

WebSphere Application Server Enterprise Edition
Component Broker



Advanced Programming Guide

Version 3.0

WebSphere Application Server Enterprise Edition
Component Broker



Advanced Programming Guide

Version 3.0

Fifth Edition (August, 1999)

This edition applies to:

VisualAge Component Development for WebSphere Application Server V3.0, Enterprise Edition for AIX, 5765–E27

VisualAge Component Development for WebSphere Application Server V3.0, Enterprise Edition for Windows NT, 5639–I07

WebSphere Application Server V3.0, Enterprise Edition for AIX, 5765–E28

WebSphere Application Server V3.0, Enterprise Edition for Solaris, 5765–E29

WebSphere Application Server V3.0, Enterprise Edition for Windows NT, 5639–I09

WebSphere Application Server V3.0, Enterprise Edition Development Runtime for Windows NT, 5639–I11

WebSphere Application Server V3.0, Enterprise Edition Development Runtime for AIX, 5765–E31

WebSphere Application Server V3.0, Enterprise Edition Development Runtime for Solaris, 5765–E30

and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable system bibliography for current information on these products.

Order publications through your IBM representative or through the IBM branch office serving your locality.

© **Copyright International Business Machines Corporation 1997, 1999. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book	ix	Event suppliers	30
Who should read this book	ix	Supplying events	31
Conventions used in this book	ix	Event consumers	33
How to send your comments.	x	Consuming events	34
 		Event channels	36
Chapter 1. Concurrency Service	1	Locating an event channel	38
The purpose of a Concurrency Service	2	Creating an event channel	39
How concurrency supports locking.	3	Configuring an event channel	41
Transactional and non-transactional locking	3	Connecting to an event channel	42
Managing updates to resources	4	Disconnecting from an event channel	44
The Concurrency Service in a transactional environment	4	Event channel samples	45
Related lock sets	4	 	
Locks and lock sets	5	Chapter 3. Notification Service	51
Lock modes	5	Communicating asynchronous events	51
Conflicting locks in a lock set	6	Communication models	54
Servicing lock requests in a lock set	7	Structured events	56
Multiple lock possession	7	Event topics	59
Programming considerations	7	Single event channel schemas	59
Deadlocks	8	Multiple event channel schemas	60
Granularity	9	Event suppliers	60
Concurrency Service tasks	10	Supplying events	62
Defining a lock set	10	Event consumers	64
Creating a lock set	10	Consuming events	65
Relating lock sets	11	Event channels	67
Obtaining and releasing locks from a lock set	12	Locating an event channel	70
Releasing locks in a transactional framework	13	Creating an event channel	71
Completing top level transactions.	14	Configuring an event channel	73
Using non-transactional locks	15	Connecting to an event channel	74
Changing the mode of a lock	17	Disconnecting from an event channel	76
Preventing deadlocks	18	FilterFactory and filters	77
Managing objects	20	Managed object-based sample	79
Handle exceptions	20	Step 1: Implementing the application server (push supplier server object)	80
Configuring run-time support	21	Step 2 : Implementing the C++ client code (PushSupplier and PullConsumer)	87
Troubleshooting.	21	Step 3: Running the sample	92
Chapter 2. Event Service	23	Chapter 4. Externalization Service	99
Communicating asynchronous events	23	Relationship to OMG Externalization Service	99
Communication models	25	Use of externalization in Component Broker	100
Events	27	 	
Event topics	28	Chapter 5. Identity Service	103
Single event channel schemas	29	Comparing objects	103
Multiple event channel schemas	29	Compare two objects.	104
		Compare multiple objects	104

Optimizations for object collections	106	Chapter 8. Security Service	221
Chapter 6. LifeCycle Service	109	Security in the distributed object system	221
Concepts of LifeCycle Service	110	Principals, credentials, and secure	
Concepts of managed objects	111	associations	222
Concepts of factories	113	Manipulating credentials	223
Concepts of factory finders	114	Access control	229
Concepts of locations	115	Other security considerations for the	
Location-based factory finding	117	server	230
Location object implementations	119	Authentication and end-users	231
Vocabulary of proximity	120	User IDs and passwords	232
FactoryFinders bound in the name space	121	Server key-tab file	232
Application factories and specialized		Logging in with environment variables	236
homes	122	Using environment variables to establish	
Managed objects and local only objects	123	authenticity	236
Creating and obtaining lifecycle objects	125	Message protection	237
Minor interfaces	126	Delegation	239
LifeCycleObject interface on managed		Enabling delegation for a secure server	239
objects	126	Security Service objects	240
GenericFactory interface on homes	127	Principal object	240
Detailed view of location-based factory		Credentials object	240
finding	128	Current object	240
Defining scope of location	128	LoginHelper object	241
Lifecycle repository structure	130	Security and servlets in WebSphere	
Factory keys	133	Enterprise	241
Factory finding specifics	135	Configuring the WebSphere Enterprise	
Default lifecycle objects	146	Servers	241
Tips for using the default lifecycle objects	152	Configuring the Java client servlet	242
Lifecycle interfaces and implementations	153	Installing and configuring servlets in	
Lifecycle object interfaces	154	WebSphere Enterprise	243
Lifecycle example	173	Example code source	243
		Java client programming note	249
Chapter 7. Naming Service	181	Chapter 9. Transaction Service	251
Naming objects in the distributed object		An example of a transaction	251
system	181	Top-level and flat transactions	252
System name space	184	Lifetime of a transaction	253
Visibility of named objects	187	Transaction scope and context	253
Local and host name tree	188	Recoverability	254
Workgroup name tree	189	Transaction outcomes	255
Cell name tree	190	The two-phase commit process	257
Navigation in the system name space	191	The one-phase commit process	258
Integration of system name spaces	192	Heuristic decisions	258
Naming contexts	193	Transaction retry limits	261
Object names	194	Transaction time limits (timeouts)	261
Local root naming context and the		Application programming using the	
bootstrap host	207	Transaction Service	262
Absolute and relative names	208	Architecture and design of a Transaction	
Summary of the naming context interface	209	Service application	262
Implementing the Naming Service	210	Visibility rules	265
		Design a Transaction Service application	266

The Transaction Service objects and interfaces	267	Form a query	326
Manage transactions in your application	269	Queries that result in an object collection	328
Controlling the Transaction Service in a running system	282	Queries that result in a data array	328
The transaction service log.	282	Queries over unnamed collections	329
Types of server start-up.	284	Coding an extendedEvaluate() method call.	329
Configuring a server to use the Transaction Service	285	Java client and Java BO example.	343
Problem determination	285	Memory management	350
Summary of the Transaction Service	286	Usage of Naming Service by query	350
Chapter 10. Session Service	287	Limit on number of query iterators per transaction	350
What is the Session Service?	287	Limit on query statement size.	351
The scope of sessions	287	Query Service tips	351
The relationship between transactions and sessions	289	Chapter 12. Query Service for OS/390 and Solaris	355
The timeout value associated with a session	289	Object-Oriented Structured Query Language	356
Resource priorities	290	Differences between OOSQL and SQL	357
Using the Session Service	290	Methods	360
Visibility rules	292	Inheritance	362
Session Service tasks.	295	Navigation	362
Setting a time limit for all new sessions	295	Collections	364
Beginning and ending a session	295	Query optimizations	364
Suspending and resuming a session.	299	The cast operator	367
Explicit and implicit propagation of session context.	300	Query over reference collections	368
Checkpoint and reset a session context	301	Data type mapping between DB2 and CORBA	369
Registering sessionable resources.	303	Query evaluators	369
Collaborating on session outcome using multiple concurrent threads	305	Default query evaluator.	370
Chapter 11. Query Service for AIX and Windows NT	307	Obtain a query evaluator	370
Object-Oriented Structured Query Language	308	Get the server name of a query evaluator	371
Differences between OOSQL and SQL	308	Topology of query evaluators and collections	374
Methods	311	Form a query	375
Inheritance	313	Queries on queryable collections.	376
Navigation	313	Queries that result in an object collection	376
Query optimizations	317	Queries that result in a data array	377
PAA pushdown rules	321	Queries over unnamed collections	378
The cast operator	321	An example using the query evaluator interface	378
Query over reference collections	322	Java clients and Java BO example	385
Data type mapping between DB2 and CORBA	323	Memory management	388
Query evaluators	323	Usage of Naming Service by query	389
Default query evaluator.	324	Limit on number of query iterators per transaction	389
Obtain a query evaluator	324	Query Service tips	389
Topology of query evaluators and collections	325	Chapter 13. Cache Service	393
		Cache Service and DB2 locking considerations	396
		Cache Service limits	396

Cache Service and Oracle locking	397	Compile the Java files	433
Chapter 14. Object Request Broker	399	Write and compile the Java client program	433
Remote method invocation.	399	Run the application	433
Conversion of objects to string form	399	Scenario: C++ client of a remote Java object	434
Code-set conversion for remote method invocations	400	Create the IDL files	435
Dynamic invocation interface (DII)	403	Produce the Java implementation bindings	435
Building a DII request	403	Write the Java implementation of getMessage()	436
Constructing a DII request	406	Compile the Java pieces.	436
Initiating a DII request	406	Create the C++ ORB adapter	436
Sample DII requests	407	Create the Java server code	436
Dynamic skeleton interface (DSI).	410	Compile and link the server	438
Chapter 15. Non-IBM ORB usage	413	Create the client program	438
The example	413	Compile and link the client program	439
Bootstrapping	415	Run the application	439
Creating the client bindings	418	Scenario: Java client of a remote C++ object	440
Running the example	419	Create the IDL file	441
Additional tips for non-IBM ORB usage	419	Produce a set of C++ implementation classes	441
Specialized homes	419	Write the C++ code to implement getMessage	442
CORBA IIOP	419	Write the C++ code for the main server program	442
Trimming client-side dependencies on Component Broker interfaces	420	Compile and link the C++ pieces to the DLL	443
Chapter 16. Interlanguage object model	421	Compile and link the DLL	443
IOM and Component Broker	421	Write and compile the Java client program	444
Defining IOM interfaces and implementations	424	Run the application	445
Communication between C++ and Java	424	Chapter 17. Workload management	447
Scenario: C++ client of a local Java object	425	Programming model	447
Create the IDL file	425	Overview	447
Run the idlc command	426	Group identity	448
Write the Java code to implement getMessage	427	Workload manageable objects	449
Compile the Java file.	428	WLM homes	450
Produce the C++ client file.	428	WLM persistent objects	450
Write the C++ client main program	429	WLM transient objects	451
Compile and link to the client	429	Scenario considerations	451
Run the application	429	Affinity management	451
Scenario: Java client of a local C++ object (NT Only)	430	Multiple activation	452
Create the IDL file	430	References to other objects	453
Produce the implementation-side C++ binding files	430	Local activation	454
Write the C++ code to implement getMessage	431	Automatic rebind	454
Compile and link the C++ piece to the DLL	431	Scenario examples	454
Produce the client stub	432	Transient application objects	454
		Business objects with DB2 application adaptor persistence	456

Application objects using remote business objects with DB2 application adaptor persistence	458	Configuring ODBC for AIX	466
Other business objects	459	Building an interface repository database	468
Application adaptors.	459	Populating the IR with Component Broker's definitions	468
BOIM application adaptors	459	Displaying the contents	468
Using Object Builder.	459	The Test.idl file	469
Adding a container	459	The makefile	469
Adding a managed object to an application	460	Running the makefile	470
Client programming model	461	Running the executable	470
Using factory finders.	461	Running irdump	470
Exceptions and recovery	462	Notices	473
Chapter 18. Interface repository	465	Trademarks and service marks	475
Using the configuration tool	465	Index	479
Creating the IR database in DB2	465		

About this book

The Component Broker Advanced Programming Guide describes the Component Broker implementation of the CORBA Object Services and the Component Broker Object Request Broker (including dynamic invocation interface (DII) procedures), Interlanguage Object Model (IOM), Interface Definition Language (IDL), and workload management.

Who should read this book

The Advanced Programming Guide is intended for application developers who use the Component Broker environment to build distributed object-oriented applications.

The examples are written in C++; therefore, programming experience in C++ and a background in object-oriented programming is required. A familiarity with Java is also helpful, but not required.

This book is a programming manual for experienced programmers who are going to use this product. It is assumed that readers of this book are familiar with concepts in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

Conventions used in this book

The following conventions distinguish different text elements:

plain Window titles, folder names, icon names, and method names.

monospace

Programming examples, user input at the command line prompt or into an entry field, user output, and directory paths.

bold Menu choices, push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combo-boxes, notebook tabs, and entry fields.

italics Programming keywords, variables, and attributes, titles of information units, initial use of unique terms, and emphasis.

The following icons are used to indicate platform-specific sections.



Denotes a section that applies only to the Windows 95 or Windows NT platform. Do not interpret this symbol to denote that an equivalent section exists for any other platform.

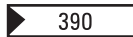
Note: The Windows 95 platform only supports the Component Broker Java client.



Denotes a section that applies only to the AIX platform. Do not interpret this symbol to denote that an equivalent section exists for any other platform.



Denotes a section that applies only to the Sun Solaris platform. Do not interpret this symbol to denote that an equivalent section exists for any other platform.



Denotes a section that applies only to the OS/390 Component Broker platform. Do not interpret this symbol to denote that an equivalent section exists for any other platform.



Denotes a section does NOT apply to OS/390 Component Broker. Do not interpret this symbol to denote that an equivalent exists in OS/390 Component Broker.

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other WebSphere Application Server Enterprise Edition documentation, send your comments by e-mail to waseedoc@us.ibm.com. Be sure to include the name of the book, the document number, the version of WebSphere Application Server Enterprise Edition, and, if applicable, the specific location of the information on which you are commenting (for example, a page number or a table number).

Chapter 1. Concurrency Service



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

The use of the Concurrency Service by user application code within the Component Broker programming model is somewhat limited. This is due to the Component Broker managed object framework already providing all the necessary resource level locking and caching to coordinate resource access by multiple transactions or threads. Further details on resource locking may be found in section “Locking and visibility concerns with Component Broker” in the *WebSphere Application Server Enterprise Edition Component Broker Planning, Performance and Installation Guide*. The Concurrency Service can be used by “application adaptor” writers to provide concurrent access to datastores currently not supported by Component Broker, however this is beyond the scope of the Component Broker programming model. This chapter is provided to document the CORBA compliant Concurrency Service provided by Component Broker for such “application adaptor” writers.

The Concurrency Service is a set of interfaces that allow an application to coordinate access by multiple transactions or threads to a shared resource. Coordinating access to a resource means that, when multiple transactions or threads try to access a single resource at the same time, any conflicting actions are reconciled so that the resource remains in a consistent state. This is known as conflict resolution.

To enable conflict resolution, the Concurrency Service supports locking using lock sets (see “Locks and lock sets” on page 5).

Component Broker for OS/390 does not support the Concurrency Service; however, you can achieve the same coordination of access to resources by configuring a Component Broker for OS/390 server with specific policies. See *OS/390 Component Broker Planning and Installation* for further information.

The purpose of a Concurrency Service

Why would you use a Concurrency Service? To answer this, consider the following typical Object Oriented business scenario. Two bank clerks want to credit the same bank account with two different amounts at the same time. Table 1 shows what could happen if a Concurrency Service is not used.

Table 1. Simultaneous updates. What can happen if a Concurrency Service is not used.

Program A: Add \$100 to a/c	Program B: Add \$20 to a/c
1. Get value of account=50	
	2. Get value of account =50
3. Add 100 internally to program	
	4. Add 20 internally to program
5. Store the result account=150	
	6. Store the result account=70

The final balance of the account is 70 dollars, rather than 170. Because the order of events is not predetermined, there is no guarantee that you will get the correct result.

To avoid these problems, you can use a Concurrency Service. Table 2 shows what happens if programs A and B use a Concurrency Service.

Table 2. Simultaneous updates. A Concurrency Service is used.

Program A: Add \$100 to a/c	Program B: Add \$20 to a/c
1. Request a write lock on the account	
2. Obtain a write lock	
	3. Request a write lock on the account
4. Get value of account =50	
5. Add 100 internally to program	
6. Store the result account=150	
7. Release the lock	
	8. Obtain a write lock
	9. Get value of account =150
	10. Add 20 internally to program
	11. Store the result account=170
	12. Release the lock

This time the result is correct. Table 2 introduces the concepts of locks and lock modes (see Locks and lock sets and “Lock modes” on page 5). A lock is

used to regulate access to a resource. A lock mode defines the way in which the resource can be accessed. For example, a “write lock” allows a program to change a file record; a “read lock” only allows the program to inspect it.

In Table 2 on page 2, transaction B is not given a lock straight away, because transaction A already holds a write lock on the account. When transaction A releases its lock, transaction B is able to get a write lock and update the account.

By using a concurrency control service in this way, you get the correct result.

How concurrency supports locking

The Concurrency Service supports both transactional and non-transactional locking (see “Transactional and non-transactional locking”). Locking prevents multiple requesters (transactions or threads that request a lock) from simultaneously accessing the same resource if their activities might conflict. This is illustrated in “The purpose of a Concurrency Service” on page 2.

The Concurrency Service is primarily intended to be used by server processes that need to manage concurrent access to resources.

The Concurrency Service manages locks on behalf of a single server process (either transactional- or thread-based locks). Shared locks between multiple server processes are not supported.

The Concurrency Service can only be used by server processes, there are no client usage bindings.

Transactional and non-transactional locking

The Concurrency Service is intended primarily for use in a transactional environment, as a complementary service to the Transaction Service (see “Chapter 9. Transaction Service” on page 251). In a transactional environment, locks are acquired and released on behalf of transactions.

It is possible to use the Concurrency Service in a non-transactional environment. In this case, locks are acquired and released on behalf of threads.

The Concurrency Service manages locks on behalf of a single server process (either transactional- or thread-based locks). Shared locks between multiple server processes are not supported.

Managing updates to resources

Although the Concurrency Service could be used by a server process to control thread execution as an alternative to semaphores, this would not be very efficient. The Concurrency Service is primarily intended to be used by server processes that need to manage concurrent access to resources. An application requiring access to a particular resource invokes a method on the server process controlling updates. Typically, the concurrency control mechanism is contained entirely within that server process, which defines the association between the locks and resources.

The Concurrency Service places no restrictions on what a resource can be. It could, for example, be a field within a record, a single file record, or a database - in the example illustrating the “The purpose of a Concurrency Service” on page 2, it is a bank account. Granularity describes the factors that you should consider when deciding what size each resource should be (see “Granularity” on page 9).

For example, consider an application `BankAccount` that controls access to resources of type `AccountBalance`. The `AccountBalance` class provides a method (`AccountBalance::get_balance`) for applications wishing to query the balance of `AccountBalance` objects. The implementation of this method might be as follows:

1. Acquire a read lock on `AccountBalance`.
2. Read value of `AccountBalance`.
3. Release the read lock on `AccountBalance`

The Concurrency Service in a transactional environment

The Concurrency Service is intended primarily for use in a transactional environment, as a complementary service to the Transaction Service (see “Chapter 9. Transaction Service” on page 251). In such an environment, a locking mechanism ensures that a transaction is unable to view the partial effects of another transaction (that is, changes that have not been committed).

Related lock sets

The Concurrency Service allows a group of lock sets to be related. This enables a server process controlling updates to drop all the locks held by a transaction in a related group by invoking just one operation, the `CosConcurrency::LockCoordinator drop_locks()` operation.

For example, a server process controlling updates might relate a group of lock sets, each of which it has created to manage access to an individual file. Within the server process's commit and rollback methods, all the locks held on any of the files by the completing transaction can be dropped by a single method call.

Locks and lock sets

A lock is associated with a single resource and a single thread or transaction. Locks can be *exclusive* or *shared*. When a transaction or thread holds an exclusive lock on a resource, it means that (in general) no other transaction or thread can obtain a lock to access the resource, until the exclusive lock is released. Write locks are exclusive. If they were not, the timing of write events could result in data corruption or outdated data could be read. Read locks, on the other hand, can be shared; because the resource is not being changed, more than one application can be given a read lock on it, with no danger of any application reading outdated data.

A lock set is a collection of all the locks associated with a single resource. A server process controlling updates creates a lock set for each resource that requires controlled access. When first created, the lock set is empty. Before a transaction or thread can access a controlled resource, it must obtain a lock, in the relevant mode, from the appropriate lock set. If the requested lock cannot be granted immediately, the request is queued. Each request for a lock, whether granted or queued, results in a new lock being created in the lock set. Each lock is associated with its requester, and exists until it is released.

A lock set is therefore the set of locks, both granted and queued, that currently exist for a controlled resource.

Lock modes

Lock modes are categories of access to resources. They define the level of concurrency that a held lock allows to other (conflicting) requesters of the lock.

Having a variety of lock modes allows more flexible conflict resolution. For example, having different modes for reading and writing means that a resource can support multiple concurrent transactions or threads that are merely reading the data of the resource. The Concurrency Service also defines *intention locks* that support locking at multiple levels of granularity.

These lock modes are available:

Read Lock Mode (R)

Obtains access to the lock for reading. This is known as a shared lock mode, because multiple read locks can be held concurrently on the same resource.

Write Lock Mode (W)

Obtains access to the lock for writing. This is known as an exclusive lock mode, because it prevents other requesters from obtaining a lock on the same resource.

Upgrade Lock Mode (U)

Only one requester can obtain this lock, but while it holds the lock other requesters can obtain read locks. Use of upgrade locks is described in more detail in “Preventing deadlocks” on page 18.

Intention Read Lock

Indicates intention to obtain a read lock. Use of the intention read lock is described in more detail in “Granularity” on page 9.

Intention Write Lock (IW)

Indicates intention to obtain a write lock. Use of the intention write lock is described in more detail in “Granularity” on page 9.

Lock Mode Capability

The following table shows the compatibility of the various lock modes. An X shows where lock modes are incompatible.

Table 3. Lock mode compatibility

	IR	R	U	IW	W
Intentional Read (IR)					X
Read (R)				X	X
Upgrade (U)			X	X	X
Intentional Write (IW)		X	X		X
Write (W)	X	X	X	X	X

Conflicting locks in a lock set

If a transaction or thread requests a lock on a resource and a lock is already held in an incompatible mode (as explained in “Lock modes” on page 5), the request is said to conflict with the held lock. The only exception to this, is when the held lock is held by the same transaction or thread. See “Multiple lock possession” on page 7.

Servicing lock requests in a lock set

If a lock request does not conflict with any held locks, and no requests are waiting to be granted in the lock set, the requested lock is granted and added to the set of held locks. If the request does conflict with a currently held lock, but no requests are waiting to be granted, it causes a queue of waiting requests to be created (containing just itself).

If a request is made for a lock, and there is already a queue of one or more waiting requests in the lock set, the request is usually added to the end of the queue (irrespective of whether or not it conflicts with any held locks).

However, if both the following apply the lock is granted immediately:

- The requester or, in the case of a transaction, one of its ancestors, already holds a lock in the lock set.
- The request does not conflict with any of the locks held.

Whenever a lock is released, the Concurrency Service automatically tries to grant the lock request that is at the front of the queue. If the request is successfully granted, the Concurrency Service removes it from the front of the queue and tries to grant the next request.

When the Concurrency Service cannot grant the lock request at the front of the queue, it searches down the queue. For each requester waiting, it checks whether that requester (or, in the case of a transaction, one of its ancestors) already holds a lock in the lock set. If this is the case and the request does not conflict with any of the locks held, the request is granted and removed from the queue.

Multiple lock possession

The Concurrency Service enables a transaction or thread to hold multiple locks in the same lock set (that is, multiple locks on the same resource) simultaneously. The locks can be of different (and possibly conflicting) modes, or of the same mode. A count is kept of the number of locks of a given mode that the transaction or thread holds. When a lock of that mode is unlocked, the count is decremented. The transaction or thread holds the lock until the count reaches zero. Therefore, to completely release a lock, the number of unlock requests must equal the number of times that the lock has been acquired in that mode.

Programming considerations

When creating a server process or an application that uses the Concurrency Service, there are a number of points you should consider:

- If a number of transactions or threads are competing for access to shared resources, “Deadlocks” on page 7 can occur. No transaction can proceed because each is waiting for a lock held by another transaction.
- When deciding on the granularity of a lock (that is, the scope of the locked resource), you need to balance the low overhead of coarse granularity with the improved concurrency of fine granularity (see “Granularity” on page 9).
- Manage objects explains how an application obtains objects from the Concurrency Service and when, if ever, it can destroy them (see “Managing objects” on page 20).
- Your server process controlling updates must be able to handle the exceptions that the Concurrency Service raises when it encounters an error (see “Handle exceptions” on page 20).

Deadlocks

When a number of transactions (or threads) are competing for shared resources, there is the risk of deadlock. This occurs when no transactions in a set can proceed because each is waiting for a lock held by another member of the set.

In the following example, deadlock occurs because programs A and B are each unable to proceed until the other releases its lock. This would be avoided if A and B both attempt to lock resource X first, then Y.

Table 4. Deadlocks

Program A	Program B
1. Request a write lock on resource X	
2. Obtain a write lock on resource X	
	3. Request a write lock on resource Y
	4. Obtain write lock on resource Y
5. Request write lock on resource Y	
6. Thread suspended waiting for program B to release its lock on Y	
	7. Request write lock on resource X
	8. Thread suspended waiting for program A to release its lock on X

Granularity

The *granularity* of a lock relates to the scope of the locked resource. For example, a lock on a resource object that represents a single file record could be described as a “fine granularity lock”; while a lock on an entire database is a “coarse granularity lock”.

Coarse granularity locks incur low overhead, because the Concurrency Service has fewer locks to manage, but reduce concurrency because conflicts are more likely to occur. Fine granularity locks improve concurrency, but result in a higher locking overhead because more locks are requested. Selecting a suitable granularity is a balance between the lock overhead and the degree of concurrency required. With the Concurrency Service, a server process controlling updates can use coarse or fine granularity by defining the associated resources appropriately.

A server process controlling updates can support variable levels of granularity on a single resource. For example, consider a collection of files each containing a number of records. A server process could associate a lock set with each of the files and a lock set with each of the records within the files. A transaction could then obtain a coarse granularity lock on a complete file or a finer granularity lock on one of the records contained within a file.

Imagine that a transaction needs to update a record in a file controlled by this server process. It would be insufficient for the transaction simply to acquire a write lock on the record, because another transaction might acquire a write lock on the complete file containing the record, and delete or modify the file.

Alternatively, the server process controlling updates could ensure that a transaction was not able to perform any updates to a record within a file unless it already held a write lock on the complete file. However, this removes the capability of finer granularity locking (on an individual record) that the server process controlling updates aimed to provide.

The Concurrency Service provides two lock modes that a server process can use to solve this problem. These modes are **intention read** (IR) and **intention write** (IW).

A transaction intending to obtain a write lock on a fine-granularity resource contained within a coarser granularity resource must first obtain an intention write lock on the coarser granularity resource. In the previous example, the coarse-granularity resource is the file that contains the record to be updated. When the intention write lock is successfully granted, the transaction can then obtain a write lock on the individual record and update it.

Notice that the ownership of an intention write lock does not prevent other transactions from obtaining an intention write lock (or an intention read lock) on the same file. However, it does prevent another transaction (which is not a descendant of the intention lock holder) from obtaining a read, upgrade, or write lock on the file. Essentially, the intention lock has restricted coarse granularity locking on the file so that multiple requesters can simultaneously hold finer granularity locks on its records.

An intention read lock is used in exactly the same way as an intention write lock, to restrict coarse-granularity locking yet allow multiple requesters to hold finer-granularity locks within a resource. An intention read lock is incompatible only with a write lock.

Concurrency Service tasks

This section describes the tasks performed when using Concurrency Services.

Defining a lock set

A server process controlling updates creates a lock set for each resource that requires controlled access.

You can define a lock set using either the LockSet interface or the TransactionalLockSet interface. The LockSet interface enables more flexibility, because the lock requests can be made on behalf of the current transaction or, if a transaction does not exist, on behalf of the current thread. There is also no need to pass an extra parameter specifying the transaction when using this interface.

Before carrying out this procedure, make sure you are familiar with: “Locks and lock sets” on page 5.

To define a lock set, define the LockSet as a private instance variable in the class header file. For example: Define the lock set as a private instance variable in the class header file as follows:

```
#include <CosConcurrency.hh>

CosConcurrencyControl::LockSet_ptr lockset;
```

only with a write lock.

Creating a lock set

A server process controlling updates creates a lock set for each resource that requires controlled access.

Before carrying out this procedure, make sure you are familiar with “Locks and lock sets” on page 5 and “Defining a lock set” on page 10.

To create a lock set, follow these steps:

1. Create a lock set for every recoverable object, using a LockSetFactory object. You can do this within the initialization code of your class.
2. Delete the LockSet factory object.

Here is an example:

To create a lock set, include the following in the initialization code for the class:

```
#include <CosConcurrency.h>
{
    ...
    CosConcurrencyControl::LockSetFactory_ptr const lockset_factory =
    CosConcurrencyControl::LockSetFactory::_create();
    this->lockset = lockset_factory->create();
    ...
    CORBA::release (lockset_factory);
    ...
}
```

The LockSetFactory object is needed only to create the LockSet. Once you have finished with a factory, you can destroy it.

Relating lock sets

The Concurrency Service allows a group of lock sets to be related (see “Related lock sets” on page 4). This enables a server process controlling updates to drop all the locks held by a transaction in a related group by invoking just one operation, the CosConcurrency::LockCoordinator drop_locks() operation.

Before carrying out this procedure, make sure you are familiar with “Creating a lock set” on page 10.

To relate lock sets, follow these steps:

1. Create the primary lock set, to which the others will be related.
2. Relate each additional lock set to the primary lock set as required.

Here is an example:

Create the primary lockset:

```
#include <CosConcurrency.hh>

CosConcurrencyControl::LockSetFactory_ptr const lockset_factory =
ConcurrencyControl::LockSetFactory::_create();
this->lockset = lockset_factory->create();
```

Create a lock set that is related to the initial one:

```
this->lockset2 = lockset_factory->create_related(lockset);
```

Create another lock set that is related to the initial one:

```
this->lockset3 = lockset_factory->create_related(lockset);
```

Note that *lockset3* is also related to *lockset2*.

The LockSetFactory object is needed only to create the LockSets. Once this has been done successfully, you can destroy it:

```
CORBA::release (lockset_factory);
```

Obtaining and releasing locks from a lock set

Before accessing data, an object must obtain a lock from the appropriate lock set. If the lock cannot be granted immediately, the thread that issued the call is blocked (suspended) until the lock can be granted. If you want the thread to continue doing useful work if the lock cannot be granted immediately, you can use the non-blocking `try_lock()` operation rather than the blocking `lock()` operation.

Before carrying out this procedure, make sure you are familiar with “Defining a lock set” on page 10, “Creating a lock set” on page 10, and “Locks and lock sets” on page 5.

To update data, follow these steps:

1. Use the `lock()` operation to obtain a write lock from the lock set that controls access to the data.
2. Update the data.
3. Release the write lock.
 - If the lock was acquired on behalf of a transaction, do this only when the transaction has completed.
 - If the lock was acquired on behalf of a thread, do this as soon as all the data has been updated.

To read data, follow these steps:

1. Use the `lock()` operation to obtain a read lock from the lock set that controls access to the data.

2. Read the data.
3. Release the read lock when it is no longer required.

The `CosConcurrency.hh` include file defines the following enums for each type of Lock Mode specified as the parameter to the `lock()/unlock()` functions:

- Read Lock - `CosConcurrencyControl::read`
- Write Lock - `CosConcurrencyControl::write`
- Upgrade Lock - `CosConcurrencyControl::upgrade`
- Intention Read Lock - `CosConcurrencyControl::intention_read`
- Intention Write Lock - `CosConcurrencyControl::intention_write`

Here is an update data example:

The following is an example of a lock request in a `BankAccount` implementation. A bank clerk updates the balance of an account using a method called `BankAccount::updateBalance`. Before it can do this, the `BankClerk` object must obtain a write lock from the `BankAccount` lockset:

```
this->lockset->lock(CosConcurrencyControl::write);
```

The `updateBalance()` method can only be used within the scope of a transaction. To avoid making public changes that might later be rolled back, the transaction's locks are not released until commit or rollback.

Here is a read data example:

The following is an example of a read only lock request in a `BankAccount` implementation. A bank clerk is performing an account query using a method called `BankAccount::retrieveBalance`. To do this, the `BankClerk` object must obtain a read lock from the `BankAccount` lockset while performing the read of the data.

```
this->lockset->lock(CosConcurrencyControl::read);  
...  
read data from datastore  
...  
this->lockset->unlock(CosConcurrencyControl::read);
```

Releasing locks in a transactional framework

In a transactional environment, a locking mechanism ensures that a transaction is unable to view the partial effects of another transaction (that is, changes that have not been committed). A transaction should only release its locks when it has completed its updates and is ready to reveal those updates to other transactions.

Although this document refers to transactions obtaining and releasing locks, this is just a convenient shorthand. It is actually the server processes controlling updates that obtain and release locks on behalf of transactions.

Before carrying out this procedure, make sure you are familiar with “The two-phase commit process” on page 257.

Locks are released as follows:

1. The transaction commits or rolls back its changes.
2. The server process controlling updates using the Concurrency Service ensures that the transaction releases its locks immediately.

You should release the locks within the server process’s commit and rollback methods. To ensure that these methods are called during the two-phase commit logic of the transaction, the resource object must register itself as a synchronization object with the transaction Coordinator using the Synchronization Interface. For further details see the “Transaction Service” section in the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*.

Important: In general, a transaction should always drop its locks during two-phase commit processing as described here. However, if a transaction acquires a lock on a resource and does not make any modifications to that resource (that is, it acquires a read lock), the transaction can drop the lock before committing or rolling back. Be careful not to release locks too early though; if a transaction requires that another transaction does not change a resource, it should hold the read lock until it completes.

When using locks:

- Ensure that a transaction never holds any locks after it has completed.
- Ensure that a transaction never attempts to acquire locks during prepare, commit or rollback processing. Following prepare, a transaction is no longer associated with the current thread, on whose behalf the lock request is made. This could result in deadlock (see “Deadlocks” on page 8).

Completing top level transactions

The Concurrency Service provides the LockCoordinator interface to ease the implementation of dropping locks prior to transaction end. One LockCoordinator object exists for each group of related lock sets for each transaction that holds a lock or has an outstanding request in one or more of the related lock sets (see “Related lock sets” on page 4). A server process

controlling updates need not concern itself with the creation or management of the LockCoordinator object because the Concurrency Service deals with this internally.

Top-level transactions are completed as follows:

1. Use the `get_coordinator()` operation to obtain the LockCoordinator object that represents the lock set and the transaction.
2. The transaction drops all the locks that it holds or has an outstanding request on. It should do this when it has committed or rolled back its changes.

Here is an example:

Suppose you are implementing a BankAccount class. You might provide a `drop_locks()` method that drops all the locks that are held by the transaction passed as an input parameter. First this method gets the LockCoordinator object representing the lock set and the transaction:

```
CosConcurrencyControl::LockCoordinator_ptr const lock_coord;  
...  
this->lockset->get_coordinator(lock_coord);
```

Next, the transaction drops all the locks that it holds or has an outstanding request on. This is done by the following call:

```
lock_coord->drop_locks();
```

If the lock set were related to other lock sets, invoking `drop_locks` against the LockCoordinator object would result in all the locks held by the transaction in the other lock sets being dropped as well.

When the transaction has completed, it must drop all its locks. In this example, you might do this in an `uninvolve_in_transaction()` method of the BankAccount object before voting read-only to prepare or before returning from commit or rollback.

Using non-transactional locks

Although the Concurrency Service is designed for use primarily in a transactional environment, it is possible to have lock sets that are only ever called outside the scope of a transaction. In such a case, each lock request in a lock set is always associated with a thread.

Before carrying out this procedure, make sure you are familiar with:

- “Defining a lock set” on page 10.
- “Creating a lock set” on page 10.
- “Obtaining and releasing locks from a lock set” on page 12.

- “Locks and lock sets” on page 5.

To use non-transactional locks, follow these steps:

1. Define the lock set.
2. Create the lock set using a LockSetFactory in the class initialization code.
3. Suspend the transaction associated with the current thread.
4. Request a lock in the lock set.
5. Read or update data.
6. Release the lock.
7. Resume the transaction associated with the current thread.

Here is an example:

Suppose in a BankAccount implementation that a class called Traninv is used for keeping track of each transaction’s involvement with a recoverable object. It does this using a hash table to implement the Coordinator-to-Resource mapping. To control access to the hash table, a lock set is defined as a private instance variable in the Traninv header file:

```
/* C++ example */

#include <CosConcurrency.hh>
{
    CosConcurrencyControl::LockSetFactory_ptr lsfact;
    CosConcurrencyControl::LockSet_ptr latch;
}
```

The latch object is then created during the initialization of the Traninv class.

```
// Get the LockSetFactory object (a new one is created if it does not
// exist)

lsfact = CosConcurrencyControl::LockSetFactory::_create();

// Create a lockset using the LockSetFactory object

latch = lsfact->create();
```

Suppose the Traninv class has a get_resource() method that returns the address of the resource associated with a transaction. It acquires this information from the hash table, which must be read locked to ensure that it does not change during access. First the Traninv::get_resource() method suspends the transaction and gets a read lock in the latch lock set on behalf of the current thread:

```
...
CosTransactions::Control_ptr const control = current->suspend();
...
this->latch->lock(CosConcurrencyControl::read);
```

When the relevant data has been accessed, the thread releases the read lock and resumes the transaction:

```
...
this->latch->unlock(CosConcurrencyControl::read);
current->resume(control);
...
release latch;
release lsfact;
```

Changing the mode of a lock

You can change the mode of a lock that has already been acquired by invoking the `change_mode()` operation on the lock set object. For example, suppose a server process controlling updates holds an upgrade lock, a special type of read lock used when the server process intends to do a subsequent write (see “Lock modes” on page 5). It would use the `change_mode()` operation to upgrade the lock to a write lock.

Before carrying out this procedure, make sure you are familiar with Lock Modes.

To change the mode of a lock, follow these steps:

1. Obtain an upgrade lock. Invoke the `lock()` operation on the appropriate lock set object.
2. Read the data.
3. If you do not need to update the data, release the lock. If you do need to update the data:
 - a. Use the `change_mode()` operation to change the mode from upgrade to write.
 - b. Update the data.
 - c. Release the write lock.

Here is an example:

In Use Non-transactional Locks, a class called `Traninv` is used for keeping track of each transaction’s involvement with a recoverable object in a `BankAccount` implementation (see “Using non-transactional locks” on page 15). `Traninv` has an `involve` method that creates a coordinator-to-resource mapping for each transaction.

The thread obtains an upgrade lock:

```
this->latch->lock(CosConcurrencyControl::upgrade);
```

Once it has successfully acquired the lock, the method looks in the hash table to see whether a coordinator-to-resource mapping exists that corresponds to the (recently suspended) transaction.

If a mapping already exists in the hash table for this transaction, there is no need to perform any additional actions. The upgrade lock is released:

```
this->latch->unlock(CosConcurrencyControl::upgrade);
```

If no mapping exists for the transaction, it must create one. It uses the `change_mode()` operation to change the mode to write:

```
this->latch->change_mode(CosConcurrencyControl::upgrade,  
    CosConcurrencyControl::write);
```

Now that the thread holds a write lock on the latch, it is safe to proceed with updates to the hash table. The hash table is updated with an entry associating the coordinator to the resource.

When all required changes have been made to the hash table, the transaction releases the write lock:

```
this->latch->unlock(CosConcurrencyControl::write);
```

Preventing deadlocks

When a number of transactions (or threads) are competing for shared resources, there is the risk of deadlock. This occurs when no transactions in a set can proceed because each is waiting for a lock held by another member of the set.

Before carrying out this procedure, make sure you are familiar with “Deadlocks” on page 8.

To prevent deadlock occurring within a single-process or application, when two transactions try to access a resource already locked by the other transaction, follow these steps:

1. Assign an arbitrary order to the resources that can be locked.
2. Ensure that all transactions or threads request locks in this order.

To prevent deadlock in an application that uses the Concurrency Service across multiple processes, specify a transaction timeout (see “Setting a time limit for all new transactions” on page 271).

If the transaction has not completed within the time specified, it automatically rolls back and its locks are released. The timeout value you should choose

depends on the application. It should be large enough to ensure that the transaction has time to complete but short enough to be an acceptable time in which to detect deadlock.

To prevent deadlock occurring when two or more requesters attempt to read and then update the same resource, obtain an upgrade lock instead of a read lock.

Here is an example:

In this example, A and B both try to read and then update the same resource:

Table 5. A and B both try to read and then update the same resource.

Program A	Program B
1. Request a read lock on resource	
2. Obtain a read lock on resource	
	3. Request a read lock on resource
	4. Obtain read lock on resource
5. Request write lock on resource	
6. Thread suspended waiting for program B to release its read lock	
	7. Request write lock on resource
	8. Thread suspended waiting for program A to release its read lock

If each registers for a single upgrade lock, followed by a write lock, the deadlock does not occur:

Table 6. Each registers for a single upgrade lock, followed by a write lock.

Program A	Program B
1. Request an upgrade lock on resource	
2. Obtain an upgrade lock on resource	
	3. Request an upgrade lock on resource
4. Request write lock on resource	
5. Obtain a write lock on resource	
6. Update resource	
7. Release write lock	
8. Release upgrade lock	
	9. Obtain an upgrade lock on resource
	10. Request a write lock on resource

Table 6. Each registers for a single upgrade lock, followed by a write lock. (continued)

	11. Obtain a write lock on resource
--	-------------------------------------

Managing objects

The first table shows how an application obtains objects from the Concurrency Service, and when, of ever, it can destroy them. If the application is not allowed to destroy an object, the second table shows when the concurrency destroys the object.

Table 7. How an application obtains and destroys objects from the Concurrency Service

Object	How the application obtains the object	When the application can destroy the object
LockSetFactory	LockSetFactory::create	At any time after the application has finished
LockSet	LockSetFactory::create or LockSetFactory::create_related	At any time after the application has finished using it
TransactionalLockSet	LockSetFactory::create_transactional or LockSetFactory::create_transactional_related	At any time after the application has finished using it
LockCoordinator	LockSet::get_coordinator or TransactionalLockSet::get_coordinator	Never (it is created and destroyed by the Concurrency Service)

Table 8. When the Concurrency Service destroys the object

Object	How the application obtains the object	When the application can destroy the object
LockCoordinator	LockSet::get_coordinator	LockCoordinator::drop_locks
	TransactionalLockSet::get_coordinator	When the last related LockSet is destroyed

Handle exceptions

When the Concurrency Service encounters an error and is unable to complete a request successfully, it raises an exception in the application environment. Your application should catch exceptions and take appropriate action.

An exception can be one of those defined in the Concurrency Service interface definition or a standard system exception. The standard system exceptions contain a minor code that identifies the nature of the problem being reported. The system exceptions are listed in the “Errors and Exceptions” section under

“Minor Codes Defined”, in the *WebSphere Application Server Enterprise Edition Component Broker Problem Determination Guide*. Exceptions are handled as follows:

1. The server process controlling updates should check the environment after every request.
2. If an exception is raised, it must clear it before issuing subsequent requests. If it does not, unpredictable results can occur.

Configuring run-time support

When you install Component Broker, the installation program sets default values for trace and log files. You can change these using the Object Editor of the System Management interface.

Troubleshooting

Problem determination is described in the *WebSphere Application Server Enterprise Edition Component Broker Problem Determination Guide*.

Chapter 2. Event Service



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

An Event Service allows objects to dynamically register or unregister their interest in specific events. An *event* is an occurrence within an object that is specified to be of interest to one or more objects. The Event Service creates a loosely joined communication channel between objects that are unfamiliar with each other.

The purpose of an Event Service is to enable objects to freely register or unregister their interest in certain events. An Event Service decouples communication between objects by defining two roles for objects: supplier objects and consumer objects. *Suppliers* produce events, while *consumers* process events.

Events are communicated among suppliers and consumers using standard CORBA requests. An Event Service contains event channels that act as supplier and consumer objects. These event channels allow multiple suppliers to communicate with multiple consumers asynchronously and without knowing about each other.

Communicating asynchronous events

The Component Broker Event Service enables you to exchange asynchronous event messages between different objects in the distributed system. This is useful for communicating about events that occur in one part of your application that another part of the application needs to know about. The asynchronous and loosely coupled nature of the Event Service allows the same event to be communicated to different parts of your application that may be interested in the same thing; neither part of your application needs to know directly of the other parts.

The Event Service introduces several key concepts:

Events

An event is a specific instance of an event message about a particular event. For instance, an event message may be generated to report that

an Insurance Agent has closed a contract for an insurance policy. This event may be monitored by one program that maintains statistics on the number of contracts that are closed on average per hour, and by another program that logs all of the activities of each agent.

Event messages are discrete, relating only to one event. However, events can be replicated to report the same event information to multiple consumers of that event.

Event topics

An event topic defines a specific event type, or a family of related event types. For instance, the events that report when agents have closed their contracts are all of the same type, even though there may be many instances of this, one for each time an agent closes a contract.

Event suppliers

An event supplier is an object that generates event messages: it is a supplier of events, or more literally, the messages that report on that event.

Event consumers

An event consumer is an object that receives event messages, making it a consumer of events.

Event channels

An event channel is a broker of event messages. Event suppliers provide their events to an event channel, and event consumers obtain those events from the same event channel. The event channel is responsible for ensuring that all events that it receives are provided to all of its connected consumers. It is also responsible for mediating between the push and pull communication models.

Each event channel is actually composed of the following seven types of objects: *EventChannel*, which anchors the set of other related event channel objects; *EventSupplierAdmin* and *EventConsumerAdmin* which are administrative objects for mediating event connections; and *PushConsumerProxy*, *PullConsumerProxy*, *PushSupplierProxy*, and *PullSupplierProxy* objects which serve to represent a specific connection between individual suppliers and consumers of the event channel.

Communication models

The Component Broker Event Service supports both a push model of communication as well as a pull model of communication. The push model allows an event supplier to push its events to the event channel. The pull model allows an event channel to pull events from its event suppliers. Likewise, the push model allows an event channel to push events on its event consumers. And the pull model allows an event consumer to pull events from its event channel. An event

channel can simultaneously support all communication models between all of its suppliers and consumers.

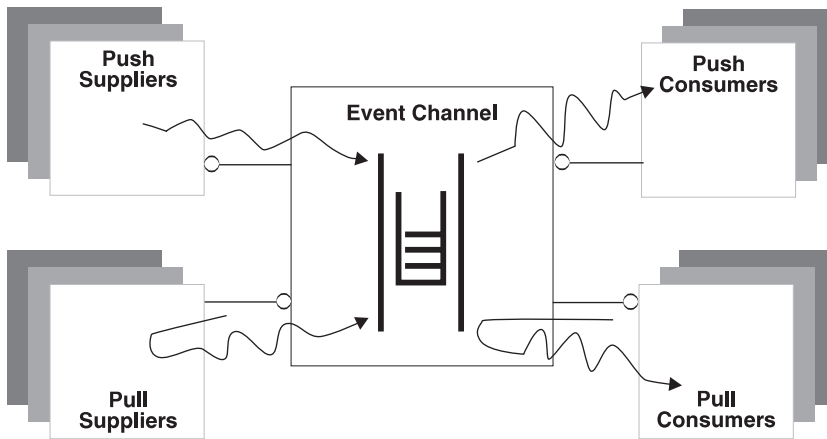


Figure 1. Event Service relationship model

Push suppliers use an event channel to push their event messages. Event channels use pull suppliers to pull their event messages. Event channels queue up all the events they receive from all of their suppliers. Event channels use consumers to push all of the events queued up in the channel. Pull consumers use event channels to pull all of the events queued up in the channel.

Communication models

The Event Service can be used with either a push model or a pull model of interaction. Event suppliers can either push or pull events with the event channel. Likewise consumers can either push or pull events with the event channel.

A push supplier, having connected with the event channel, can push an event message on the event channel whenever the event occurs, or whenever it is appropriate for the supplier to present the event message to the event channel after the occurrence of the event. The push supplier can push its event(s) using the `CosEventComm::PushConsumer::push()` method supported by the `ProxyPushConsumer` that it obtained from the `SupplierAdmin` during the event channel connection process.

A pull supplier, having connected with the event channel, will have events pulled from it periodically. The event channel does this by invoking a `pull()`

operation on the pull supplier. The pull supplier must support the `CosEventComm::PullSupplier` interface, and in particular implement the `pull()` and `try_pull()` operations. The `pull()` operation should be implemented to block and return an event message when the event has occurred. The `try_pull()` operation should never block, but rather should return an event message and a boolean return value of `TRUE` if an event has occurred, and to return a null message and a boolean return value of `FALSE` if not.

The event channel will perform its `pull()` operations on a separate thread for each pull supplier. Nonetheless, the pull supplier should be implemented to return from the `pull()` operation as quickly as possible (after an event is available) to avoid consuming resources in the event channel that could eventually affect its throughput.

Even though the event channel in Component Broker only uses the `pull()` operation, both operations should be implemented as other event channel implementations may invoke either. These methods should be implemented to work together so either method can be used interchangeably and yet only produce one event message for an event occurrence.

A push consumer, having connected with the event channel, will have events pushed on it whenever the event message is supplied to the event channel. When an event message is supplied to the event channel, the event channel iterates through its full list of connected push consumers and pushes the same event on each of those consumers.

The push consumer must support the `CosEventComm::PushConsumer` interface, and in particular implement the `push()` operation. It is up to the push consumer to decide what to do with the event when it arrives. Often, consumers are implemented to spawn a separate thread of their own to process the event. Each push occurs on a separate thread to isolate the effect that different consumers can have on each other. Nonetheless, push consumers should return from the `push()` operation as quickly as possible to avoid consuming resources in the event channel that could eventually effect its throughput.

A pull consumer, having connected with the event channel, can pull or `try_pull` an event from the event channel whenever it is ready to receive the next event message. The `pull()` operation is a blocking request and only returns when the event channel has another event message to supply to the pull consumer. The `try_pull()` operation is not blocking and returns an event-message if one is available, or simply return `FALSE` if not. The pull consumer can pull its event(s) using the `CosEventComm::PullSupplier::pull()` or `CosEventComm::PullSupplier::try_pull()` methods supported by the `ProxyPullSupplier` obtained from the `ConsumerAdmin` object during the event channel connection process.

Because the `CosEventComm::PullSupplier::pull()` operation is intended to be a blocking method, and since some events may take a long time to actually occur, it is possible that the communication network will fail or time-out before the supplier has the chance to return from the pull. This can be a significant problem for some applications. If the pull request times out, the consumer is notified through an exception to which it can respond, usually by simply re-invoking the pull request, and perhaps re-connecting to the event channel if necessary. On the other hand, the supplier is not necessarily informed of the error. Typically the supplier is programmed to present an event on return from the `pull()` method. However, if the connection from the consumer has timed out, then the return occurs without actually supplying anything to the consumer, and the event is lost.

You can reduce the chances of this happening by setting the ORB time-out value for the communication session between the consumer and supplier to zero (no time-out). However, this won't guard against other potential communication errors that could have the same effect. Thus, you should avoid the pull consumer model if it is important to you to avoid lost event messages.

Events

An event is a specific instance of an event message about a particular event. When an instance of an event occurs, an event supplier can produce an event message representing the occurrence of that event. The event message can contain as much or as little information about that occurrence as is relevant to your application.

An event has some form of abstract type, although this type is not formalized by the Event Service. The event type defines the format of the content of the event message. This is needed so that event suppliers and event consumers can know how to parse the information within the event message.

The Event Service defines an event message only as a CORBA 'any'. This means that it is entirely up to your application to define the format of the event message in a way that it can use to recognize its contents. The structure of the event message can be as elaborate or as simple as is appropriate for your application's needs. At one extreme, the event message can be a simple integer containing an event identifier, a value that can be used to uniquely identify the occurrence of the event. At this extreme, the event message does not convey anything more about the event instance except that it occurred and a handle that uniquely identifies that occurrence. Event consumers might be able to use this event identifier later to correlate back to a separate repository containing more information about the event.

When designing event messages you should consider the following:

- How much information do your event consumers need to know about the event instance, and what information will they use in the handling of that event?
- Are you likely to introduce other event consumers in the future? What information will they likely need?
- How will you evolve the schema for the event message without affecting backwards compatibility of existing event consumers and suppliers? Are you willing to modify your event suppliers and consumers whenever you change the event schema?
- How often are these events likely to occur? Will they happen so frequently that having too much information in them will likely affect system throughput?
- How often will consumers need all of the information you supply in the event? Would it be better to supply only a little information in the event message, including correlation information that can be used later to obtain the rest of the detailed information from a separate repository if and when it is actually needed?
- How do you want to group event types in an event topic, and what is the relationship of those event topics to event channels? Different event channels can be used for different event topics. Likewise, different event topics can be used to group one or more event types. All of these can be used to separate different event types, thus reducing the amount of information that you have to include in the event message itself.
- How much extra event handling information do you need? Do you need, for instance, to track how long, on average, it takes to process an event message from the time that it is produced to the time that it is consumed? Is there a maximum lifetime for the event handling, a point beyond which if the event is not handled it is no longer relevant and should be discarded? Do you need to prioritize the event? Do you want to correlate related events?

Event topics

The idea of an event topic is an abstract concept introduced by Component Broker to help you consider the relationship between event instances, event types, and event channels. Within the running system there may be many occurrences of many different types of events. Likewise, you can introduce as many different event channels as you like. Designing an effective event schema requires achieving the right balance between system performance, integrity, and programming and administrative simplicity.

Single event channel schemas

If you use only one event channel for all events supplied and consumed in your application, then administration is fairly straight-forward: you need only define one event channel and share that same event channel with all of your suppliers and consumers. However, using a single event channel raises the following concerns:

- The event channel can become a single point of failure. Since you are only dealing with one event channel, then if that event channel should fail then all of your suppliers and consumers are affected.
- Event suppliers must be able to identify the type and instance of event message they are producing.
- Event consumers need to identify the type(s) and instance(s) of event message they are interested in processing, filtering out all those that they're not interested in, and sorting through the various types and instances they are interested in, if there is more than one type.

Multiple event channel schemas

Another strategy is to have a different event channel for every type of event message, and possibly a different event channel for each set of event messages coming from different groups of either or both suppliers and consumers. This has the obvious consequence of proliferating event channels, and in the latter case significantly increasing administrative complexity for associating groups of either or both event suppliers and consumers to determine which ones should be attached to which event channel instance. By extension, event suppliers and consumers may have to connect to numerous event channels for all of the types of events they handle.

In many cases there are a set of event types that are highly related. For instance, you may have a different event type for each of the steps an insurance agent performs. Each of these event types may be related by virtue that they all pertain to the steps an agent performs. This grouping may be relevant, for instance, if you have an event consumer that tracks all of an agent's actions, as a sort of audit trail to ensure they've followed all of the right procedures or to monitor their productivity.

You can group these related event types into an event topic. This is an artificial grouping, not supported in any formal sense by the Event Service, that you can use at your discretion and for your convenience. Having identified one or more event topics, you can determine how many event channels you will need. A common practice is to define a distinct event channel for each event topic. Each topic is given a label and each event channel is registered in the system name space with this topic name. Thus a supplier can connect to a particular event channel based on the topic(s) it supplies events for. Similarly, a consumer can connect to a particular event

channel based on the topic(s) the consumer handles. Thus, if an event channel fails, only suppliers and consumers of that topic are effected. Likewise, suppliers and consumers can concentrate on the topics that they are programmed to handle, and in doing so avoid the system overhead associated with dealing with events not relevant to them.

Event suppliers

An event supplier is an object that supplies event messages to an event channel. An event supplier can use either the push or pull model of interaction with the event channel. The same object can be a supplier to more than one event channel concurrently, however if the supplier uses the pull model there is no way for it to distinguish which event channel is pulling an event. Push model event suppliers must support the `CosEventComm::PushSupplier` interface. Pull model event suppliers must support the `CosEventComm::PullSupplier` interface. It is up to you to provide an implementation for the appropriate interface if you are introducing an event supplier.

Event suppliers must create a connection to their event channel before they can begin to supply events. To do this, they must first obtain the appropriate event channel. In most cases, event channels are registered in the system name space when they are created, and therefore obtaining an event channel is a matter of resolving its appropriate name in the system name space. As discussed in “Event topics” on page 28, a common strategy is to register event channels with the name of the event topic they broker. Thus you can use the event topic name to resolve the appropriate event channel. Having located an event channel, the supplier must then connect to it. At this point, the supplier can supply any number of event messages, until the connection is terminated.

Push-model event suppliers can supply events using the `CosEventComm::PushConsumer::push()` operation on their event channel. Pull-model event suppliers can supply events when they are invoked with either the `CosEventComm::PullSupplier::pull()` or `CosEventComm::PullSupplier::try_pull()` operation. The connection with the event channel can be terminated by either the supplier or by the event channel. If the supplier is using the push-model it can disconnect by invoking the `CosEventComm::PushConsumer::disconnect_push_consumer()` operation on the event channel, and the event channel can disconnect the supplier by invoking the `CosEventComm::PushSupplier::disconnect_push_supplier()` operation on the supplier. If the supplier is using the pull-model it can disconnect by invoking the `CosEventComm::PullConsumer::disconnect_pull_consumer()` operation on the

event channel, and the event channel can disconnect the supplier by invoking the `CosEventComm::PullSupplier::disconnect_pull_supplier()` operation on the supplier.

As event channel connections are not retained persistently, if the event channel fails, the connection must be reestablished. In the case of an event channel failure, push model event suppliers are notified in an exception on their next `CosEventComm::PushConsumer::push` request. Event suppliers should catch the `CosEventComm::Disconnected` exception. This is an indication that the event channel has failed. At that point, the supplier should re-connect before re-pushing the event.

Pull model event suppliers are never notified if an event channel fails. Instead, pull model event suppliers simply stop being invoked for any further events. If this is an issue for the supplier, it can periodically attempt to reconnect with the event channel. If the event channel has not failed then the reconnect attempt raises the `CosEventChannelAdmin::AlreadyConnected` exception with no other consequences.

Event suppliers can reside in pure clients. However, since pure clients cannot export object references, event suppliers in pure clients are constrained to only use the push model of interaction. Further, pure client event suppliers are not notified if the event channel disconnects. If the event channel disconnects, the event supplier is left to detect the loss of connection with the `CosEventComm::Disconnected` exception raised on their next push request.

Supplying events

Supplying events requires you to first locate and connect to an event channel that is intended to handle your event topic. The Event Service supports two models of interaction between event suppliers and event channels: the push model and the pull model. If your event supplier resides in a pure client, you must use the push model.

Supply events using the following steps:

1. Locate an event channel. You need an event channel with which to connect. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. In addition, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.
2. Connect to the event channel. Follow the procedure described in “Connecting to an event channel” on page 42 to connect with the event channel.
3. Supply the event. As a push-model event supplier, you can use the `CosEventComm::PushConsumer::push()` method to push your event on the event channel. As a pull-model event supplier, you must implement the

`CosEventComm::PullSupplier::pull()` and `CosEventComm::PullSupplier::try_pull()` methods. The `pull()` method should block until ready to return an event. The `try_pull()` method should always return immediately; returning a boolean value of `TRUE` and the event message if one is ready, or returning a boolean value of `FALSE` and a null event message if not.

The following example demonstrates how to connect as a push supplier to the `AgentActions` event channel created in “Creating an event channel” on page 39 and supply methods using the `push()` method. This example assumes that you are your own push-supplier object, and that you are operating in a server process.

```
// Declare an intermediate Object, the AgentActions event
// channel, a supplier admin object and a push consumer
// proxy.

CORBA::Object_var intermediateObject;
CosEventChannelAdmin::EventChannel_var agentActionsEC;
CosEventChannelAdmin::SupplierAdmin_var agentActionsSA;
CosEventChannelAdmin::ProxyPushConsumer_var agentActionsPPC;

// Assuming an event structure has been declared in IDL, declare
// an instance of that event structure for reporting on agent
// actions, and a CORBA::Any for conveying it with.

ActionEventStructure_var agentActionEvent;
::CORBA::Any_var eventAny;

// Locate the AgentActions event channel, obtain the supplier
// admin, and obtain a push consumer proxy.

intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/event-channels/AgentActions");
agentActionsEC=
    CosEventChannelAdmin::EventChannel::_narrow(intermediateObject);
agentActionsSA = agentActionsEC->for_suppliers();
agentActionsPPC = agentActionsSA->obtain_push_consumer();

// Connect to the event channel proxy.

agentActionsPPC->connect_push_supplier(this);

// Do some activity for the Insurance agent

...

// Supply an event indicating the completion of that action.
// Fill in the fields of the agentActionEvent

...
```

```
// Set the event in an Any
eventAny <<= agentActionEvent;

// Push the event
agentActionsPPC->push(eventAny);
```

Event consumers

An event consumer is an object that consumes event messages from an event channel. An event consumer can use either the push or pull model of interaction with the event channel. The same object can be a consumer of more than one event channel concurrently, however, and if the consumer uses the push model there is no way for it to distinguish which event channel is pushing an event. Push model event consumers must support the `CosEventComm::PushConsumer` interface. Pull model event consumers must support the `CosEventComm::PullConsumer` interface. It is up to you to provide an implementation for the appropriate interface if you are introducing an event consumer.

Event consumers must first create a connection to their event channel before they can begin to consume events. To do this, they must first obtain the appropriate event channel. In most cases, event channels are registered in the system name space when they are created, and obtaining an event channel is simply a matter of resolving its appropriate name in the system name space. As discussed in “Event topics” on page 28, a common strategy is to register event channels with the name of the event topic they broker. Thus you can use the event topic name to resolve the appropriate event channel.

Having located an event channel, the consumer must then connect to it. At this point, the consumer can consume any number of event messages, until the connection is terminated. Push model event consumers can consume events when they are invoked with the `CosEventComm::PushConsumer::push()` operation. Pull model event consumers can consume events using either the `CosEventComm::PullSupplier::pull()` or `CosEventComm::PullSupplier::try_pull()` operation on their event channel.

The connection with the event channel can be terminated by either the consumer or by the event channel. If the consumer is using the push model, it can disconnect by invoking the `CosEventComm::PushSupplier::disconnect_push_supplier()` operation on the event channel, and the event channel can disconnect the consumer by invoking the `CosEventComm::PushConsumer::disconnect_push_consumer()` operation on the consumer. If the consumer is using the pull model it can disconnect by invoking the `CosEventComm::PullSupplier::disconnect_pull_supplier()` operation on the

event channel, and the event channel can disconnect the consumer by invoking the `CosEventComm::PullConsumer::disconnect_pull_consumer()` operation on the consumer.

As event channel connections are not retained persistently, if the event channel fails, the connection must be reestablished. If the event channel fails, pull model event consumer is notified in an exception on their next `CosEventComm::PullSupplier::pull` or `CosEventComm::PullSupplier::try_pull` request. Event consumers should catch the `CosEventComm::Disconnected` exception. This is an indication that the event channel has failed. At that point, the consumer should reconnect before re-pulling the event.

Push model event consumers are never notified if an event channel fails. Instead, they stop receiving any further events. If this is an issue for the consumer, it can periodically attempt to reconnect with the event channel. If the event channel has not failed then the reconnect attempt raises the `CosEventChannelAdmin::AlreadyConnected` exception with no other consequences.

Event consumers can reside in pure clients. However, since pure clients cannot export object references, then event consumer in pure clients are constrained to only use the pull model of interaction. Further, pure client event consumers are not notified if the event channel decides to disconnect. The event consumer is left to detect the loss of connection with the `CosEventComm::Disconnected` exception raised on their next pull or try_pull request.

Consuming events

This procedure demonstrates how to consume an event from an event channel. To consume an event, you have to first locate and connect to an event channel that is intended to handle your event topic. The Event Service supports the push and pull models of interaction between event consumers and event channels. If your event consumer resides in a pure client, you must use the pull model.

Consume an event from the event channel using the following steps:

1. Locate an event channel. You need an event channel to connect with. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. In addition, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.
2. Connect to the event channel. Follow the procedure described in “Connecting to an event channel” on page 42 to connect with the event channel.

3. Consume the event. As a pull model consumer, you can use the `CosEventComm::PullSupplier::pull()` or `CosEventComm::PullSupplier::try_pull()` method to consume an event. If you use the `pull()` method, your request blocks until an event is ready to be returned. If you use the `try_pull()` method, your request always returns immediately, and indicates in a boolean return value whether an event was available to return. As a push model consumer, you must implement the `CosEventComm::PushConsumer::push()` method. This method is invoked whenever or not an event is available for you to consume.

The following example demonstrates how to connect as a pull consumer to the `AgentActions` event channel created in “Creating an event channel” on page 39 and consume events using the `pull()` method. This example assumes you are your own pull-consumer object, and that you are operating in a server process.

```
// Declare an intermediate Object, the AgentActions event channel,  
// a consumer admin object, and a pull supplier proxy.  
  
CORBA::Object_var intermediateObject;  
CosEventChannelAdmin::EventChannel_var agentActionsEC;  
CosEventChannelAdmin::ConsumerAdmin_var agentActionsCA;  
CosEventChannelAdmin::ProxyPullSupplier_var agentActionsPPS;  
  
// Assuming an event structure has been declared in IDL, declare  
// an instance of that event structure for receiving the agent  
// actions, and a CORBA::Any for consuming it with.  
  
ActionEventStructure_var agentActionEvent;  
::CORBA::Any_var eventAny;  
  
// Locate the AgentActions event channel, obtain the consumer  
// admin, and obtain a pull supplier proxy.  
  
intermediateObject =  
CBSeriesGlobal::nameService()->resolve_with_string(  
    "/resources/event-channels/AgentActions");  
agentActionsEC=  
    CosEventChannelAdmin::EventChannel::_narrow(intermediateObject);  
agentActionsCA = agentActionsEC->for_consumers();  
agentActionsPPS = agentActionsCA->obtain_pull_supplier();  
  
// Connect to the event channel proxy.  
  
agentActionsPPS->connect_pull_consumer(this);  
  
// Consume an event.  
// Pull the event  
  
eventAny = agentActionsPPS->pull();  
  
// Obtain the event from the Any
```

```
eventAny >>= agentActionEvent;  
  
// Handle the fields of the agentActionEvent  
...
```

Event channels

Event channels have two forms. In their abstract form, an event channel is a broker of event messages, consuming them from their connected suppliers, and supplying them to their connected consumers. In their concrete form, an event channel is composed of the following object types:

CosEventChannelAdmin::EventChannel

This is the anchor point for the other event channel objects. EventChannel objects define the abstract event channel, and are the objects registered in the system name space that enable you to find an event channel.

CosEventChannelAdmin::SupplierAdmin

SupplierAdmin objects broker connections between suppliers and the event channel.

CosEventChannelAdmin::ConsumerAdmin

ConsumerAdmin objects broker connections between consumers and the event channel.

CosEventChannel::ProxyPushConsumer

ProxyPushConsumer objects define a specific connection between a supplier and the event channel. An instance of a ProxyPushConsumer is created for each supplier that connects to the event channel using the push-model of interaction.

CosEventChannel::ProxyPullConsumer

ProxyPullConsumer objects define a specific connection between a supplier and the event channel. An instance of a ProxyPullConsumer is created for each supplier that connects to the event channel using the pull-model of interaction.

CosEventChannel::ProxyPushSupplier

ProxyPushSupplier objects define a specific connection between a consumer and the event channel. An instance of a ProxyPushSupplier is created for each consumer that connects to the event channel using the push model of interaction.

CosEventChannel::ProxyPullSupplier

ProxyPullSupplier objects define a specific connection between a consumer and the event channel. An instance of a ProxyPullSupplier is created for each consumer that connects to the event channel using the pull model of interaction.

Event channels can support multiple simultaneous connections of suppliers and consumers using any mixture of push model or pull model interactions. In Component Broker, event channels are managed objects with persistent references. Thus they must reside in a Component Broker server process, and can be registered in the system name space. All suppliers and consumers that resolve the same event channel reference get to the same instance of event channel. Event channels are automatically reactivated in the Component Broker server when they are first referenced.

As brokers of event messages, event channels are in fact both event consumers and event suppliers (consumers to their connected event suppliers, and suppliers to their connected event consumers), proxies are used to mediate between consumers and suppliers, and between push and pull models of interaction.

Event channels multicast all of the event messages they receive. That is, when a supplier provides an event message to the event channel, that event message is supplied to all of the consumers that are currently connected to the event channel. The event message is not removed from the event channel queues until it has been consumed by all of its connected consumers. However, in Component Broker, event channels are transient. They do not retain either supplier or consumer connections persistently, nor do they retain event messages persistently. Connections and pending event messages are lost whenever the event channel is passivated or terminated.

Component Broker automatically creates a single *default event channel* that is registered in the system name space in `/cell/resources/event-channels/cell-default`. This event channel can be used for any purpose, but to allow for proper interoperation between all of the applications that could end up using it, all event messages using the event channel in a production environment must use the data type as `CORBA::Any`.

If you need an event channel within your application, you should introduce your own event channel for your own purposes. See “Event topics” on page 28 for more information that may be useful in deciding how to divide up your event system topology. Having done so, you should design an appropriate structure for your event messages, as discussed in “Events” on page 27. You can create a new event channel programmatically using the standard Component Broker programming model for managed objects, as described in “Creating an event channel” on page 39.

Having created the event channel you should be sure to register it in the system name space so that your suppliers and consumers can find it at run-time. This is done for you automatically if you use the

```
ExtendedEventChannelAdmin::  
    createVisibleEventChannel()
```

operation. Otherwise, you can use the `IExtendedEventChannelAdmin::IEventChannelHome::createEventChannel()` operation and manually bind the event channel in the system name space.

You can also use the Component Broker System Management facility to configure a new event channel. This is described in more detail in “Configuring an event channel” on page 41. This process automatically register the event channel in the system name space so that your suppliers and consumers can find it at run-time. The Component Broker system name space includes naming contexts where you can register event channels by name at the following locations:

- `/host/resources/event-channels/workgroup/resources/event-channels`
- `/cell/resources/event-channels`

Alternately, you can register your event channels in your own application name tree.

Locating an event channel

This procedure demonstrates how you can locate an event channel by name in the system name space. This is useful to both event suppliers and event consumers who are finding an event channel they can use to exchange event messages.

You can locate an event channel in the fashion demonstrated here only if the event channel already exists and has been bound in the system name space. You must know the name of the event channel to use this procedure.

Use the following steps to locate an event channel:

1. Determine the name of the event channel you want to use. This may be the name of the default event channel automatically created and registered in the system name space by Component Broker installation. Or it could be an event channel that your application created and bound in the system name space at some earlier point in time.
2. Resolve the event channel from the system name space. Use the Event Service operations to resolve that event channel from the system name space, by the name that you determined in the previous step.

The default event channel is created automatically during Component Broker installation. The following example shows how to obtain the default event channel:

```
// Declare the targeted event channel  
  
CORBA::Object_var intermediateObject;  
CosEventChannelAdmin::EventChannel_var defaultEC;
```

```

// Obtain the default event channel and narrow it to an event
// channel

intermediateObject=CBSeriesGlobal::nameService()
->resolve_with_string(
    " /cell/resources/event-channels/cell-default");
defaultEC = CosEventChannelAdmin::EventChannel::_narrow(
    intermediateObject);

```

Creating an event channel

An event channel is created essentially in the same way that any other managed object is created, that is, by first obtaining a factory finder with a desired location scope. By extension, such a factory finder must exist. Component Broker provides a set of default factory-finders that it creates and registers in the system name space during Component Broker installation. These are found in the following locations:

- /host/resources/factory-finders/host-scope
- /workgroup/resources/factory-finders/workgroup-scope
- /cell/resources/factory-finders/cell-scope

In addition, Component Broker automatically creates a home for event channels in every server process. However, the least granularity of scoping that any of the default factory finders supports is a host scope. Therefore, if you want the event channel created in a more specific server you must create a new factory finder with a corresponding location scope that narrows to that server.

Create an event channel using the following steps:

1. Obtain a factory finder with the desired location scope. You need a factory finder with which to create your new event channel. Alternately, you can create a non-managed factory finder to use for finding the new event channel's factory.
2. Use the factory finder to find a factory (a home) for event channels. Having obtained a factory finder, you can now use it to find a factory for factory finders. You do this with the `find_factory`, `find_factories()`, `find_factory_from_string()`, or `find_factories_from_string()` methods. The event channels interface is `IEventChannelAdminManagedClient::EventChannel`. This is the principal interface name that you is passed in to the `find_factory*` or `find_factories*` methods.
3. Create the event channel. The factory specialization for event channels introduces the `createEventChannel()` and `createVisibleEventChannel()` methods which you can use to create the new event channel. You must narrow to the `IExtendedEventChannelAdmin::IEventChannelHome` interface to use either of these new methods.

4. Register the event channel in the system name space. This step is optional. When you've created an event channel, often you want to retain this object for future use in your application to be shared by event suppliers and consumers. How you do this in your application is up to you, but the typical approach is to bind the event channel in the system name space. Again, where you bind it in the system name space is up to you. One obvious and commonly used choice is to bind it in `/host/resources/event-channels`, `/workgroup/resources/event-channels`, or `/cell/resources/event-channels` depending on what visibility you want to give it. Another choice is to bind it under your own application, contexts within `/host/applications`, `/workgroup/applications`, or `/cell/applications`. Typically, event channels are bound with the event topic name for the types of events they are intended to broker.

If you use the

`IExtendedEventChannelAdmin::IEventChannelHome::createVisibleEventChannel()` method, you can specify the relative name (the event channel's topic name), and a boolean flag for the name spaces in which you want the event channel registered. The specialized home automatically registers the new event channel into the name space in the indicated locations.

The example that follows demonstrates creating a new event channel, and binding it in the system name space under `/host/resources/event-channels` with the name `AgentActions`. The event channel is created within the local host.

```
// Declare an intermediate Object, the event-channels naming
// context, the targeted factory finder where the new event
// channel will be created, a event channel factory, and the
// new event channel.

CORBA::Object_var intermediateObject;
IExtendedNaming::NamingContext_var eventChannelsNC;
IExtendedLifecycle::FactoryFinder_var hostScopeFF;
IExtendedEventChannelAdmin::IEventChannelHome_var factoryOfEC;
CosEventChannelAdmin::EventChannel_var myNewEC;
ByteString_var key;

// Obtain the default factory finder with a host-scope and
// narrow to an IExtendedLifecycle::FactoryFinder so that we
// can use the find_factory_from_string operation.

CBSeriesGlobal::Initialize();
//Initialize the CBSeries environment

intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/factory-finders/host-scope");
hostScopeFF = IExtendedLifecycle::FactoryFinder::_narrow(
    intermediateObject);
```

```

// Find a factory for event channels and narrow to the factory
// specialization

intermediateObject = hostScopeFF->find_factory_from_string(
    "IEventChannelAdminManagedClient.object interface");
factoryOfEC =
    IExtendedEventChannelAdmin::IEventChannelHome::_narrow
    (intermediateObject);

// Use the factory to create a new event channel.

myNewEC = factoryOfEC->createEventChannel(key);

// Bind the new event channel in the system name space.

intermediateObject = CBSeriesGlobal::nameService->resolve(
    "/host/resources/event-channels");
eventChannelsNC =
    IExtendedNaming::NamingContext::_narrow(intermediateObject);
eventChannelsNC->bind_with_string("AgentActions", myNewEC);

```

The last four statements could have been combined into the following statement:

```

myNewEC = factoryOfEC->createVisibleEventChannel(key,
    "AgentActions", 0, 1, 0);

```

Configuring an event channel

This procedure demonstrates how to configure a new event channel using the Component Broker System Management facility. This is useful for when you need an event to support a new event topic. This procedure is entirely administrative and does not involve any programming.

Configure a new event channel using the following steps:

1. Navigate to the desired server. The new event channel must be configured within a specific server. Use the System Management user interface to navigate to the host and server on which you want the factory finder configured.
2. Create an event channel model. Insert an event channel model within the server.
3. Set event channel model attributes. Use the attributes notebook to fill in the event channel model attribute value. This includes the System Management name for the event channel (don't confuse this with the name of the event channel in the system name space, although these can be set to the same value), the name of the event channel in the system name space, and the home of the event channel.
4. Apply the model changes. Select the apply function for the server, host, or zone where you've created the event channel model. This results in a

corresponding event channel object being automatically created in the targeted server and registered in the system name space.

Connecting to an event channel

This procedure demonstrates how to connect to an event channel. You need to connect to an event channel before you can supply or consume events. The event channel must have already been created and registered in a way that you can find it.

Connect to an event channel using the following steps:

1. Locate an event channel. You need an event channel with which to connect. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. Otherwise, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.
2. Acquire an event channel administration object. Use the `for_suppliers()` or `for_consumers()` operation to get an event channel administration object, depending on whether you want connecting to be an event supplier or an event consumer.

If you are an event supplier, then use the `for_suppliers()` operation to get back a `CosEventChannelAdmin::SupplierAdmin` object.

If you are an event consumer, then use the `for_consumers()` operation to get back a `CosEventChannelAdmin::ConsumerAdmin` object.

3. Acquire an event channel proxy object. Use the `obtain_push_consumer()`, `obtain_pull_consumer()`, `obtain_push_supplier()`, or `obtain_pull_supplier()` operation on the corresponding event channel administration object to get an event channel proxy object, depending on whether you are an event supplier or event consumer, and depending on whether you are going to use the push-model or the pull-model of communication.

If you are a supplier using the push model, then use the `obtain_push_consumer()` operation to get back a `CosEventChannelAdmin::ProxyPushConsumer` object.

If you are a supplier using the pull model, then use the `obtain_pull_consumer()` operation to get back a `CosEventChannelAdmin::ProxyPullConsumer` object.

If you are a consumer using the push model, then use the `obtain_push_supplier()` operation to get back a `CosEventChannelAdmin::ProxyPushSupplier` object.

If you are a consumer using the pull model, then use the `obtain_pull_supplier()` operation to get back a `CosEventChannelAdmin::ProxyPullSupplier` object.

4. Connect with the proxy. Use the `connect_push_supplier()`, `connect_pull_supplier()`, `connect_push_consumer()`, or

connect_pull_consumer() method to connect to the event channel proxy, depending on whether you are a supplier or consumer, and whether you are using the push or pull model of communication.

If you are a push supplier, then use the connect_push_supplier on your ProxyPushConsumer.

If you are a pull supplier, then use the connect_pull_supplier on your ProxyPullConsumer.

If you are a push consumer, then use the connect_push_consumer on your ProxyPushSupplier.

If you are a pull consumer, then use the connect_pull_consumer on your ProxyPullSupplier.

Normally, you have to supply a reference to yourself, namely your supplier or consumer object, on the connect request. This is used either to pull or push on you, as well as to inform you when the connection is being terminated. However, if you are connecting from a pure-client you are not able to use the connect_pull_supplier() or connect_push_consumer() methods. Also, you will have to supply a NIL object reference in either the connect_push_supplier() or connect_pull_consumer() requests. This basically indicates to the event channel that you are operating from a pure-client and avoids the issuing of any disconnect requests.

The example that follows demonstrates how to connect as a push supplier to the AgentActions event channel that was created in “Creating an event channel” on page 39. This example assumes you are your own push-supplier object, and that you are operating in a server process.

```
// Declare an intermediate Object, the AgentActions event
// channel, a supplier admin object, a push consumer proxy.

CORBA::Object_var intermediateObject;
CosEventChannelAdmin::EventChannel_var agentActionsEC;
CosEventChannelAdmin::SupplierAdmin_var agentActionsSA;
CosEventChannelAdmin::ProxyPushConsumer_var agentActionsPPC;

// Locate the AgentActions event channel, obtain the supplier
// admin, and obtain a push consumer proxy.

CBSeriesGlobal::Initialize();
//Initialize the CBSeries environment

intermediateObject =CBSeriesGlobal::nameService()
->resolve_with_string(
    "cell/resources/event-channels/AgentActions");
agentActionsEC=
    CosEventChannelAdmin::EventChannel::_narrow(intermediateObject);
agentActionsSA = agentActionsEC->for_suppliers();
agentActionsPPC = agentActionsSA->obtain_push_consumer();
```

```
// Connect to the event channel proxy.  
agentActionsPPC->connect_push_supplier(this);
```

Disconnecting from an event channel

Disconnecting from an event channel is useful for when you want to stop supplying or consuming events. It essentially un-registers your event supplier or event consumer with the event channel. After disconnecting, push model suppliers can no longer push, pull model suppliers can no longer be pulled, push model consumers can no longer be pushed, and pull model consumers can no longer pull.

If you are already connected to an Event Channel, do the following to disconnect:

Use the `disconnect_push_consumer()`, `disconnect_pull_consumer()`, `disconnect_push_supplier()`, or `disconnect_pull_supplier()` method to disconnect from the event channel, depending on whether you are a supplier or consumer, and whether you are using the push model or pull model of interaction.

- As a push supplier, use the `CosEventComm::PushConsumer::disconnect_push_consumer` on your push consumer proxy.
- As a pull supplier, use the `CosEventComm::PullConsumer::disconnect_pull_consumer` on your pull consumer proxy.
- As a push consumer, use the `CosEventComm::PushSupplier::disconnect_push_supplier` on your push supplier proxy.
- As a pull consumer, use the `CosEventComm::PullSupplier::disconnect_pull_supplier` on your pull supplier proxy.

The following example demonstrates how to disconnect from the `AgentActions` event channel that was previously connected to in “Connecting to an event channel” on page 42. This example assumes this is being done from a push supplier.

```
// Assuming we've already connected to the AgentActions event  
// channel as push supplier  
...  
agentActionPPC->disconnect_push_consumer();
```


Event channel samples

A complete non system managed object sample is provided. The system managed object sample will be available when Event Services are supported by Object Builder.

A push model stock application is shown here. First, a Stock object is created. The stock price is changed by prompting for user input. The amount of stock price changing which can be either a positive or a negative number (long for simplicity) is entered by the user. When the stock price falls to 0, an exception "InvalidPrice" is thrown and the application deletes the Stock object and terminates.

The following example illustrates the Stock.cpp:

```
#include "Stock.ih"
#include <stdio.h>
void main()
{
    Stock_Impl *Stock;
    char str[10];
    long amount;

    try
    {
        // create Stock object
        Stock = new Stock_Impl;
        while (1)
        {
            cout << "Enter the amount: << endl;
            gets(str);
            amount = atoi(str); // convert string to long
            Stock->changePrice(amount); // change the stock price
        }
    }
    catch (Stock::InvalidPrice)
    {
        cout << "InvalidPrice exception !" << endl;
    }
    catch (...)
    {
        cout << "unknown exception !" << endl;
    }
    delete(Stock);
}
```

In the push module, a Customer application is also created as an ORB server. The ORB server is first initialized, the Customer object is then created and the server is waiting for some action.

The following example illustrates the Customer.cpp:

```

#include "Customer.ih"
void main(int argc, char *argv[])
{
    CORBA::ImplDef *imp;
    CORBA::ORB_var op;
    static CORBA::BOA_var bp;
    Customer_Impl *Customer;
        // Initialize the server's ImplDef, ORB, and BOA
    imp = new CORBA::ImplDef();
    imp->set_protocols("SOMD_TCPIP");
    op = CORBA::ORB_init(argc, argv, "DSOM");
    bp = op->BOA_init(argc, argv, "DSOM_BOA");
    bp->impl_is_ready(imp, 0);

    Customer = new Customer_Impl;    // create Customer

    cout << " ... Server Listening..." << endl; cout.flush();
    bp->execute_request_loop(CORBA::BOA::SOMD_WAIT);
}

```

The Stock object is inherited from the CosEventComm::PushSupplier class which has a disconnect_push_supplier() method defined. An InvalidPrice exception and a changePrice() method are introduced by the Stock interface.

The following example illustrates the Stock.idl:

```

#include <CosEventComm.idl>
interface Stock : CosEventComm::PushSupplier
{
    exception InvalidPrice {};
    void changePrice(in long amount) raises (InvalidPrice);
};

```

In the Stock object implementation header, a constructor is added. Two instance variables, *price* and *pxyPushC* are included. The instance variable *price* is used to hold the stock price. The *pxyPushC* is a CosEventChannelAdmin::ProxyPushConsumer pointer which is used to communicate to the Event Channel.

The following example illustrates the Stock.ih:

```

#include <CosEventChannelAdmin.hh>
#include <Stock.hh>
class Stock_Impl : public virtual Stock_Skeleton
{
    public:
        Stock_Impl(); // constructor
        CORBA::Void changePrice(CORBA::Long amount);
        CORBA::Void disconnect_push_supplier();

    protected:
        long price;
        CosEventChannelAdmin::ProxyPushConsumer_var pxyPushC;
};

```

In the Stock object implementation file, the “constructor” initializes the stock price to 100. The Event Channel home “ecHome” is located, and the Event Channel “ec” is created. The object reference of the Event Channel is then converted into a string and saved in a file to be used by other applications. The Supplier Admin is then obtained through the Event Channel and the ProxyPushConsumer is obtained through the Supplier Admin. Finally the Stock object is connected to the Event Channel through the ProxyPushConsumer. A NULL is passed through the connect_push_supplier() method call to indicate that the Stock object is a pure client application.

The changePrice() method updates the price change for the Stock object and throws an InvalidPrice exception when the stock price falls below 0.

The disconnect_push_supplier() method is not implemented because the Stock object is a pure client application and cannot accept any method calls.

The following example illustrates the Stock_i.cpp:

```
#include "Stock.ih"
#include <IExtendedEventChannelAdmin.hh>
#include <IExtendedLifeCycle.hh>
#include <CBSeriesGlobal.hh>
#include <fstream.h>
Stock_Impl::Stock_Impl()
{
    CORBA::Object_var intermediateObject;
    IExtendedLifeCycle::FactoryFinder_var hostScopeFF;
    IExtendedEventChannelAdmin::IEventChannelHome_var ecHome;
    ByteString_var key;
    CosEventChannelAdmin::EventChannel_var ec = NULL;
    CosEventChannelAdmin::SupplierAdmin_var sa = NULL;
    char *ec_stringref;
    ofstream fout("../ec.dat");          // file contains the ec object reference
    CORBA::ORB_var orb;
        cout << "Enter - constructor" << endl;
    price = 100;          // initial price
                        // initialize the CBSeries environment
    CBSeriesGlobal::Initialize();
                        // obtain the default factory finder with a host scope
    intermediateObject = CBSeriesGlobal::nameService()->resolve_with_string(
                        "host/resources/factory-finders/host-scope");
                        // narrow to a factory finder
    hostScopeFF = IExtendedLifeCycle::FactoryFinder::_narrow(intermediateObject);
                        // find the event channel factory
    intermediateObject = hostScopeFF->find_factory_from_string(
                        "IEventChannelAdminManagedClient::EventChannel.object interface");
                        // narrow to the event channel home
    ecHome = IExtendedEventChannelAdmin::IEventChannelHome::_narrow(
                        intermediateObject);
        cout << "ecHome is found!" << endl;
                        // use the factory in the event channel home to create
                        // a new event channel
}
```

```

    ec = ecHome->createEventChannel(key);
        // write the object reference out to a file
    ec_stringref = orbp->object_to_string(ec);
        cout << ec_stringref << endl;

    fout << ec_stringref;
    fout.close();

        // Get Supplier Admin through Event Channel
    sa = ec->for_suppliers();
        // Get ProxyPushConsumer through Supplier Admin
    pxyPushC = sa->obtain_push_consumer();
        // Connect to the Event Channel
    pxyPushC->connect_push_supplier(NULL);
}

CORBA::Void Stock_Impl::changePrice(CORBA::Long amount)
{
    CORBA::Any ev;
    char *str = CORBA::string_alloc(100);
        cout << "Enter - changeBalance: price = " << price << " amount = "
            << amount << endl;
    price = price + amount;
    if (price < 0)
    {
        price = 0;
        cout << "throw InvalidPrice exception !" << endl;
        throw Stock::InvalidPrice();
    }
    else
    {
        itoa(price, str, 10);
        ev <<= str;
        pxyPushC->push(ev);
    }
}

CORBA::Void Stock_Impl::disconnect_push_supplier()
{
}

```

The Customer object is inherited from the CosEventComm::PushConsumer class which has two methods defined: the push() method and the disconnect_push_consumer() method. No additional methods are added in the Customer object.

The following example illustrates the Customer.idl:

```

#include <CosEventComm.idl>
interface Customer : CosEventComm::PushConsumer
{
};

```

In the Customer object implementation header, only a constructor is added.

The following example illustrates the Customer.ih:

```

#include "Customer.hh"
class Customer_Impl : public virtual ::Customer_Skeleton
{
    public:
        Customer_Impl();
        CORBA::Void push (const CORBA::Any & data);
        CORBA::Void disconnect_push_consumer ();
};

```

In the Customer object implementation file, the “constructor” initializes the ORB. The Event Channel object reference created by the Stock application is obtained from the file. The Consumer Admin is then obtained through the Event Channel and the ProxyPushSupplier is obtained through the Consumer Admin. Finally the Customer object is connected to the Event Channel through the ProxyPushSupplier. The current Customer object is passed through the connect_push_consumer() method call to indicate that the Customer object is a server application. The Event Channel can communicate to the Customer server through the Customer object reference.

The push() method displays the event received from the Event Channel which is the stock price.

The disconnect_push_consumer() method displays a message to indicate that the Customer application is just disconnected by the Event Channel.

The following example illustrates the Customer_i.cpp:

```

#include "Customer.ih"
#include <CosEventChannelAdmin.hh>
#include <IExtendedLifeCycle.hh>
#include <CBSeriesGlobal.hh>
#include <fstream.h>
#include <time.h>
Customer_Impl:: Customer_Impl()
{
    int argc;
    char **argv = NULL;
    CORBA::ORB_var op;
    char objref[1024];
    ifstream fin_ec("../ec.dat");
    CORBA::Object_var optr;
    CosEventChannelAdmin::EventChannel_var ec = NULL;
    CosEventChannelAdmin::ConsumerAdmin_var ca = NULL;
    CosEventChannelAdmin::ProxyPushSupplier_var pxyPushS;
    CORBA::Object_var intermediateObject;
    IExtendedLifeCycle::FactoryFinder_var hostScopeFF;
    short i;

    cout << "Enter - constructor" << endl;
    // initial ORB
    op = CORBA::ORB_init(argc, argv, "DSOM");
    // Get Notify Channel through file ec.dat
    memset(objref, 1024, '\0');

```

```

    fin_ec >> objref;
    optr = op->string_to_object(objref);
    ec = CosEventChannelAdmin::EventChannel::_narrow(optr);
        // Get Consumer Admin through Event Channel
    ca = ec->for_consumers();
        // Get ProxyPushSupplier through Consumer Admin
    pxyPushS = ca->obtain_push_supplier();
        // Connect to the Event Channel!
    pxyPushS->connect_push_consumer(this);
        cout << "Exit - constructor" << endl;
}

CORBA::Void Customer_Impl::push(const CORBA::Any & data)
{
    char *str;
        cout << "Enter - push" << endl;
    data >>= str;
    cout << str << endl;
}

CORBA::Void Customer_Impl::disconnect_push_consumer()
{
    cout << "Enter - disconnect_push_consumer" << endl;
}

```

Chapter 3. Notification Service



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

The Component Broker Notification Service is based on the Component Broker Event Service implementation with additional capabilities like filtering and quality of services. Therefore, most of the following information is similar to the Chapter 2. Event Service chapter.

A Notification Service allows objects to dynamically register or unregister their interest in specific events. An event is an occurrence within an object that is specified to be of interest to one or more objects. The Notification Service creates a loosely joined communication channel between objects that are unfamiliar with each other.

The purpose of a Notification Service is to enable objects to freely register or unregister their interest in certain events. A Notification Service decouples communication between objects by defining two roles for objects: supplier objects and consumer objects. Suppliers produce events, while consumers process events.

Events are communicated among suppliers and consumers using standard CORBA requests. A Notification Service contains event channels that act as supplier and consumer objects. These event channels allow multiple suppliers to communicate with multiple consumers asynchronously and without knowing about each other.

In this document, the event channel and the notification channel both imply the Notification Service event channel.

Communicating asynchronous events

The Component Broker Notification Service enables you to exchange asynchronous event messages between different objects in the distributed system. This is useful for communicating about events that occur in one part of your application that another part of application needs to know about. The asynchronous and loosely coupled nature of the Event Service allows the same event to be communicated to many different parts of your application

that may be interested in the same thing; neither part of your application needs to know directly of the other parts.

The Notification Service introduces several key concepts:

Structured Events

A structured event is a well-defined data structure into which a wide variety of event types can be mapped. An event is a specific instance of an event message about a particular event. For instance, an event message may be generated to report that an Insurance Agent has closed a contract for an insurance policy. This event may be monitored by one program that maintains statistics on the number of contracts that are closed on average per hour, and by another program that logs all of the activities of each agent.

Event messages are discrete, relating only to one event. However, events can be replicated to report the same event information to multiple consumers of that event.

The Notification Service implemented uses only the structured events for communication.

Event topics

An event topic defines a specific event type, or a family of related event types. For instance, the events that report when agents have closed their contracts are all of the same type, even though there may be many instances of this, one for each time an agent closes a contract.

Event suppliers

An event supplier is an object that generates event messages: it is a supplier of events, or more literally, the messages that report on that event.

Event consumers

An event consumer is an object that receives event messages, making it a consumer of events.

Filters A Filter is an object that is used by the consumers to receive only the events that they are interested in and not all the events that are being supplied.

Event channels

An event channel is a broker of event messages. Notification suppliers provide their events to an event channel, and event consumers obtain those events from the same event channel. The event channel is responsible for ensuring that all events that it receives are provided to all of its connected consumers. It is also responsible for mediating between the push and pull communication models.

Each event channel is actually composed of the following seven types of objects: *EventChannel*, which anchors the set of other related event channel objects; *SupplierAdmin* and *ConsumerAdmin* which are administrative objects for mediating event connections; and *StructuredProxyPushConsumer*, *StructuredProxyPullConsumer*, *StructuredProxyPushSupplier*, and *StructuredProxyPullSupplier* objects which serve to represent a specific connection between individual suppliers and consumers of the event channel.

Communication models

The Component Broker Notification Service supports both a push model of communication as well as a pull model of communication. The push model allows an event supplier to push its events to the event channel. The pull model allows an event channel to pull events from its event suppliers. Likewise, the push model allows an event channel to push events on its event consumers. And the pull model allows an event consumer to pull events from its event channel. An event channel can simultaneously support all communication models between all of its suppliers and consumers.

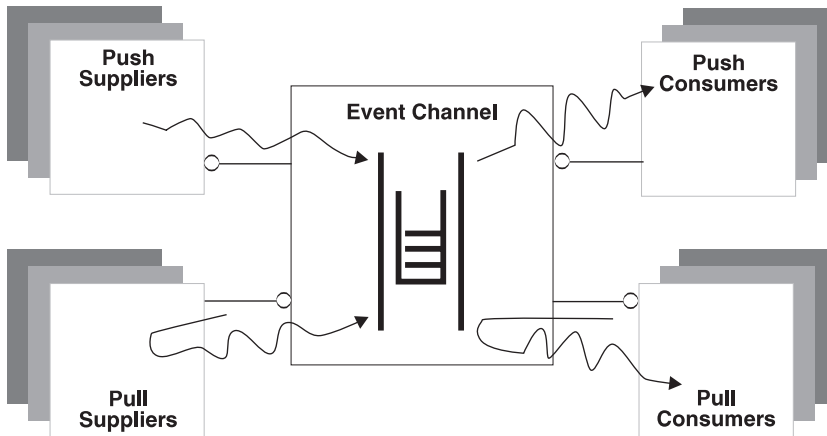


Figure 2. Notification Service relationship model

Push suppliers use an event channel to push their event messages. Event channels use pull suppliers to pull their event messages. Event channels queue up all the events they receive from all of their suppliers. Event channels use consumers to push all of the events queued up in the channel. Pull consumers use event channels to pull all of the events queued up in the channel.

Communication models

The Event Service can be used with either a push model or a pull model of interaction. Event suppliers can either push or pull events with the event channel. Likewise, consumers can either push or pull events with the event channel.

A push supplier, having connected with the event channel, can push an event message on the event channel whenever the event occurs, or whenever it is appropriate for the supplier to present the event message to the event channel after the occurrence of the event. The push supplier can push its event(s) using the `CosNotifyComm::StructuredPushConsumer::push_structured_event()` method supported by the `StructuredProxyPushConsumer` that it obtained from the `SupplierAdmin` during the event channel connection process.

A pull supplier, having connected with the event channel, will have events pulled from it periodically. The event channel does this by invoking a pull operation on the pull supplier. The pull supplier must support the `CosNotifyComm::StructuredPullSupplier` interface, and in particular implement the `pull_structured_event` and `try_pull_structured_event` operations. The `pull_structured_event()` operation should be implemented to block and return an event message when the event has occurred. The `try_pull_structured_event()` operation should never block, but rather should return an event message and a boolean return value of `TRUE` if an event has occurred, and to return a null message and a boolean return value of `FALSE` if not.

The event channel will perform its pull operations on a separate thread for each pull supplier. Nonetheless, the pull supplier should be implemented to return from the pull operation as quickly as possible (after an event is available) to avoid consuming resources in the event channel that could eventually affect its throughput.

Even though the event channel in Component Broker only uses the pull operation, both operations should be implemented as other event channel implementations may invoke either. These methods should be implemented to work together so either method can be used interchangeably and yet only produce one event message for an event occurrence.

A push consumer, having connected with the event channel, will have events pushed on it whenever the event message is supplied to the event channel. When an event message is supplied to the event channel, the event channel iterates through its full list of connected push consumers and pushes the same event on each of those consumers.

The push consumer must support the `CosNotifyComm::StructuredPushConsumer` interface, and in particular implement the `push_structured_event` operation. It is up to the push consumer to decide what to do with the event when it arrives. Often, consumers are implemented to spawn a separate thread of their own to process the event. Each push occurs on a separate thread to isolate the effect that different consumers can have on each other. Nonetheless, push consumers should return from the push operation as quickly as possible to avoid consuming resources in the event channel that could eventually effect its throughput.

A pull consumer, having connected with the event channel, can pull or `try_pull` an event from the event channel whenever it is ready to receive the next event message. The pull operation is a blocking request and will only return when the event channel has another event message to supply to the pull consumer. The `try_pull` operation is not blocking and will return an event-message if one is available, or simply return `FALSE` if not. The pull consumer can pull its event(s) using the `CosNotifyComm::StructuredPullSupplier::pull_structured_event` or `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event` methods supported by the `StructuredProxyPullSupplier` obtained from the `ConsumerAdmin` object during the event channel connection process.

Since the `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` operation is intended to be a blocking method, and since some events may take a long time to actually occur, it is possible that the communication network will fail or time-out before the supplier has the chance to return from the pull. This can be a significant problem for some applications. If the pull request times out, the consumer will be notified through an exception that it can respond to, usually by simply re-invoking the pull request, and perhaps re-connecting to the event channel if necessary. On the other hand, the supplier is not necessarily informed of the error. Typically the supplier is programmed to present an event on return from the pull method. However, if the connection from the consumer has timed out, then the return will occur without actually supplying anything to the consumer, and the event will be lost.

You can reduce the chances of this happening by setting the ORB time-out value for the communication session between the consumer and supplier to zero (no time-out). However, this does not guard against other potential communication errors that could have the same effect. Thus, you should avoid the pull consumer model if it is important to avoid lost event messages.

Structured events

A structured event is an event that is encapsulated into a well-defined data structure. An event is a specific instance of an event message about a particular event. When an instance of an event occurs, an event supplier can produce an event message representing the occurrence of that event. The event message can contain as much or as little information about that occurrence as is relevant to your application.

The Notification Service defines an event message as a structured event. Each structured event consists of two main components: a header and a body. The header part mainly consists information about the name of the event, the type of the event and the various (optional) Quality of Services (QoS) that are applicable to the event message. The body consists of the actual contents of the event instance upon which the consumer is most likely to base filtering decisions.

The suppliers and the consumers in the Notification Service are defined such that the structured events can be transmitted directly, without any repackaging needed.

The structure of the event message can be as elaborate or as simple as is appropriate for your application's needs provided it follows the format for the structured event. At one extreme, the event message can be a simple integer containing an event identifier, a value that can be used to uniquely identify the occurrence of the event with no QoS related information. At this extreme, the event message does not convey anything more about the event instance except that it occurred and a handle that uniquely identifies that occurrence. Event consumers might be able to use this event identifier later to correlate back to a separate repository containing more information about the event.

At the other extreme, the event message may contain a number of QoS properties like the priority of the event, the time when the channel should discard the event, and so on.

The following IDL represents an example of a more sophisticated event message structure:

```
typedef CosTrading::PropertySeq FilterableEventBody;
typedef struct Event_s {

    // Part 1: Header
    // Fixed Header
    string                domain_type;
    string                event_type;
    string                event_name;
    // Variable Header
    short                 Priority;
```

```

    TimeBase::UtcT          StopTime;
    TimeBase::UtcT          Timeout;

    // Part 2: Filterable Event Data

    FilterableEventBody     filterable_data;

    // Part 3: Un-filterable Event Data

    any                     unfilterable_data;
}
StructuredEvent;
};

```

This event structure has provisions in it for defining the type and name of the event instance; a set up quality-of-service controls: Priority, StopTime and Timeout. In addition, this event structure can include event data (in the filterable_data field) that can be used to filter the event message.

This type of structure might be useful in situations where event processing is central to the overall architecture of your application structure and processes, where not only being able to distinguish essential information about the event occurrence itself is important, but also where being able to monitor the handling of the event message is important, and where having sufficient robustness to ensure the longevity of the event handling even in the presence of further evolutions in your application is important too. However, this structure may be overly complicated for other application situations.

In this implementation three types of QoS properties are supported. They are Priority, StopTime and Timeout.

Priority indicates the relative priority of the event compared to other events in the channel in terms of delivery. According to the OMG Notification Service specification it should take a value between -32,767 and 32,767 with -32,767 being the lowest priority, 32,767 being the highest, and 0 being the default. Internally, in our implementation we treat all the priorities less than 1 (that is, 0 to -32,767) to be 1, priorities greater than 9 (10 to 32,767) to be 10. All other priorities (1 to 9) have their original priority numbers assigned.

StopTime is an absolute time (for example, 12/12/99 at 23:59) when the channel should discard the event.

Timeout is a relative time (for example, 10 minutes from time received) when the channel should discard the event.

One can set the QoS at the following levels of scope: notification channel, admin objects, individual proxy objects and the event itself. Accessor operations (set_qos and get_qos) are available at each of these levels (except

for the event where the QoS is set explicitly in its message as shown in the Structured Event example above) to set and get the various QoS properties. These levels of scope form a simple hierarchy, reflecting the ability to override QoS at various levels. The QoS properties set at the event level take higher precedence, followed by the QoS properties set at the individual proxy levels, followed by the QoS properties set at the admin objects level, followed by the QoS properties set at the notification channel level.

The samples provided at the end of this document illustrates a Structured Event (`define_event()` method) and the use of the QoS settings on a per-event basis in the header field of this event.

This event structure has provisions in it for defining the type and name of the event instance; a set up quality-of-service controls, including priority. In addition, this event structure can include event data that can be used to filter the event message.

This type of structure might be useful in situations where event processing is central to the overall architecture of your application structure and processes, where not only being able to distinguish essential information about the event occurrence itself is important, but also where being able to monitor the handling of the event message is important, and where having sufficient robustness to ensure the longevity of the event handling even in the presence of further evolutions in your application is important too. However, this structure may be overly complicated for other application situations.

The `define_event()` method in the sample at the end of this document provides an example of a structured event.

When designing the contents for your event messages you should consider the following:

- How much information do your event consumers need to know about the event instance, and what information will they use in the handling of that event?
- Are you likely to introduce other event consumers in the future? What information will they likely need?
- How will you evolve the schema for the event message without affecting backwards compatibility of existing event consumers and suppliers? Are you willing to modify your event suppliers and consumers whenever you change the event schema?
- How often are these events likely to occur? Will they happen so frequently that having too much information in them will likely affect system throughput?
- How often will consumers need all of the information you supply in the event? Would it be better to supply only a little information in the event

message, including correlation information that can be used later to obtain the rest of the detailed information from a separate repository if and when it is actually needed?

- How do you want to group event types in an event topic, and what is the relationship of those event topics to event channels? Different event channels can be used for different event topics. Likewise, different event topics can be used to group one or more event types. All of these can be used to separate different event types, thus reducing the amount of information that you have to include in the event message itself.
- How much extra event handling information do you need? Do you need, for instance, to track how long, on average, it takes to process an event message from the time that it is produced to the time that it is consumed? Is there a maximum lifetime for the event handling, a point beyond which if the event is not handled it is no longer relevant and should be discarded? Do you need to prioritize the event? Do you want to correlate related events?
- Which data is filterable and which is not?

Event topics

The idea of an event topic is an abstract concept introduced by Component Broker to help you consider the relationship between event instances, event types, and event channels. Within the running system there may be many occurrences of many different types of events. Likewise, you can introduce as many different event channels as you like. Designing an effective event schema requires achieving the right balance between system performance, integrity, and programming and administrative simplicity.

Single event channel schemas

If you use only one event channel for all events supplied and consumed in your application, then administration will be fairly straight-forward: you need to define one event channel and share that same event channel with all of your suppliers and consumers. However, using a single event channel raises the following concerns:

- The event channel can become a single point of failure. Because you are only dealing with one event channel, if that event channel should fail then all of your suppliers and consumers are affected.
- Event suppliers must be able to identify the type and instance of event message they are producing.
- Event consumers need to identify the type (or types) and instance (or instances) of the event message they are interested in processing, filtering

out all those in which they are not interested, and sorting through the various types and instances in which they are interested, if there is more than one type.

Multiple event channel schemas

Another strategy is to have a different event channel for every type of event message, and possibly a different event channel for each set of event messages coming from different groups of either or both suppliers and consumers. This has the obvious consequence of proliferating event channels, and in the latter case significantly increasing administrative complexity for associating groups of either or both event suppliers and consumers to determine which ones should be attached to which event channel instance. By extension, event suppliers and consumers may have to connect to numerous event channels for all of the types of events they handle.

In many cases, you will find there are a set of event types that are highly related. For instance, you may have a different event type for each of the steps an insurance agent performs. Each of these event types may be related by virtue that they all pertain to the steps an agent performs. This grouping may be relevant, for instance, if you have an event consumer that tracks all of an agent's actions, as a sort of audit trail to ensure they've followed all of the right procedures or to monitor their productivity.

You can group these related event types into an event topic. This is an artificial grouping, not supported in any formal sense by the Event Service, that you can use at your discretion and for your convenience. Having identified one or more event topics, you can determine how many event channels you will need. A common practice is to define a distinct event channel for each event topic. Each topic is given a label and each event channel is registered in the system name space with this topic name. Thus a supplier can connect to a particular event channel based on the topic (or topics) for which it supplies events. Similarly, a consumer can connect to a particular event channel based on the topic (or topics) the consumer handles. Thus, if an event channel fails, only suppliers and consumers of that topic are affected. Likewise, suppliers and consumers can concentrate on the topics that they are programmed to handle and in doing so avoid the system overhead associated with dealing with events not relevant to them.

Event suppliers

An event supplier is an object that supplies event messages to an event channel. An event supplier can use either the push or pull model of interaction with the event channel. The same object can be a supplier to more than one event channel concurrently, however if the supplier uses the pull model there is no way for it to distinguish which event channel is pulling an

event. Push model event suppliers must support the `CosNotifyComm::StructuredPushSupplier` interface. Pull model event suppliers must support the `CosNotifyComm::StructuredPullSupplier` interface. It is up to you to provide an implementation for the appropriate interface if you are introducing an event supplier.

Event suppliers must create a connection to their event channel before they can begin to supply events. To do this, they must first obtain the appropriate event channel. In most cases, event channels are registered in the system name space when they are created, and therefore obtaining an event channel is a matter of resolving its appropriate name in the system name space. As discussed in “Event topics” on page 59, a common strategy is to register event channels with the name of the event topic they broker. Thus you can use the event topic name to resolve the appropriate event channel. Having located an event channel, the supplier must then connect to it. At this point, the supplier can supply any number of event messages, until the connection is terminated.

Push-model event suppliers can supply events using the `CosNotifyComm::StructuredPushConsumer::push_structured_event()` operation on their event channel. Pull-model event suppliers can supply events when they are invoked with either the `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` or `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event()` operation. The connection with the event channel can be terminated by either the supplier or by the event channel. If the supplier is using the push-model it can disconnect by invoking the `CosNotifyComm::StructuredPushConsumer::disconnect_structured_push_consumer()` operation on the event channel, and the event channel can disconnect the supplier by invoking the `CosNotifyComm::StructuredPushSupplier::disconnect_structured_push_supplier()` operation on the supplier. If the supplier is using the pull-model it can disconnect by invoking the `CosNotifyComm::StructuredPullConsumer::disconnect_structured_pull_consumer()` operation on the event channel, and the event channel can disconnect the supplier by invoking the `CosNotifyComm::StructuredPullSupplier::disconnect_structured_pull_supplier()` operation on the supplier.

As event channel connections are not retained persistently, if the event channel fails, the connection will have to be reestablished. In the case of an event channel failure, push model event suppliers will be notified in an exception on their next `CosNotifyComm::StructuredPushConsumer::push_structured_event()` request. Event suppliers should catch the `CosEventComm::Disconnected` exception. This is an indication that the event channel has failed. At that point, the supplier should re-connect before re-pushing the event.

Pull model event suppliers are never notified if an event channel fails. Instead, pull model event suppliers will simply stop being invoked for any further events. If this is an issue for the supplier, it can periodically attempt to reconnect with the event channel. If the event channel has not failed then the reconnect attempt will simply raise the `CosEventChannelAdmin::AlreadyConnected` exception with no other consequences.

Event suppliers can reside in pure clients. However, since pure clients cannot export object references, event suppliers in pure clients are constrained to only use the push model of interaction. Further, pure client event suppliers will not be notified if the event channel disconnects. If the event channel disconnects, the event supplier is left to detect the loss of connection with the `CosEventComm::Disconnected` exception raised on their next push request.

Supplying events

Supplying events requires you to first locate and connect to an event channel that is intended to handle your event topic. The Event Service supports two models of interaction between event suppliers and event channels: the push model and the pull model. If your event supplier will reside in a pure client, you must use the push model.

Supply events using the following steps:

1. Locate an event channel. You will need an event channel with which to connect. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. In addition, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.
2. Connect to the event channel. Follow the procedure described in “Connecting to an event channel” on page 74 to connect with the event channel.
3. Supply the event. As a push-model event supplier, you can use the `CosNotifyComm::StructuredPushConsumer::push_structured_event()` method to push your event on the event channel. As a pull-model event supplier, you must implement the `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` and `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event()` methods. The pull method should block until ready to return an event. The `try_pull` method should always return immediately; returning a boolean value of `TRUE` and the event message if one is ready, or returning a boolean value of `FALSE` and a null event message if not.

The following example demonstrates how to connect as a push supplier to the `AgentActions` event channel created in “Creating an event channel” on

page 71 and supply methods using the push method. This example assumes that you are your own push-supplier object and that you are operating in a server process.

```

// Declare an intermediate Object, the AgentActions event channel,
// a supplier admin object and a push consumer proxy.

CORBA::Object_var intermediateObject;
CosNotifyChannelAdmin::EventChannel_var agentActionsEC;
CosNotifyChannelAdmin::SupplierAdmin_var agentActionsSA;
CosNotifyChannelAdmin::StructuredProxyPushConsumer_var agentActionsSPPC;

// Declare the ID that the StructuredProxyPushConsumer object is assigned
// when it is created.
CosNotifyChannelAdmin::ProxyID SPPCid;

// Define an event of type Structured Event that needs to be pushed.
CosNotification::StructuredEvent structuredEvent_var =
    new CosNotification::StructuredEvent;

// Locate the AgentActions event channel, obtain the supplier admin,
// and obtain a structured push consumer proxy.

intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/notify-channels/AgentActions");
agentActionsEC=
    INotifyChannelAdminManagedClient::EventChannelFactory::_narrow(
        intermediateObject);
agentActionsSA = CosNotifyChannelAdmin::SupplierAdmin::_narrow(
    agentActionsEC->default_supplier_admin());
agentActionsPPC =
    CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
        agentActionsSA->obtain_notification_push_consumer(
            CosNotifyChannelAdmin::STRUCTURED_EVENT, SPPCid));

// Connect to the event channel proxy.

agentActionsPPC->connect_structured_push_supplier(this);

// Do some activity for the Insurance agent

...

// Supply an event indicating the completion of that action.
// Fill in the fields of the structured event
// (structuredEvent) Shown in the sample program at the
// end of the document.

...

// Push the event
agentActionsPPC->push_structured_event(structuredEvent);

```

Event consumers

An event consumer is an object that consumes event messages from an event channel. An event consumer can use either the push or pull model of interaction with the event channel. The same object can be a consumer of more than one event channel concurrently, however, and if the consumer uses the push model there is no way for it to distinguish which event channel is pushing an event. Push model event consumers must support the `CosNotifyComm::StructuredPushConsumer` interface. Pull model event consumers must support the `CosNotifyComm::StructuredPullConsumer` interface. It is up to you to provide an implementation for the appropriate interface if you are introducing an event consumer.

Event consumers must first create a connection to their event channel before they can begin to consume events. To do this, they must first obtain the appropriate event channel. In most cases, event channels are registered in the system name space when they are created, and obtaining an event channel is simply a matter of resolving its appropriate name in the system name space. As discussed in “Event topics” on page 59, a common strategy is to register event channels with the name of the event topic they broker. Thus you can use the event topic name to resolve the appropriate event channel.

Having located an event channel, the consumer must then connect to it. If the consumer is interested in filtering events then it should also create a filter object and use it to connect to the supplier proxies so that only the events that match the constraints specified in the filter objects are delivered to it. At this point, the consumer can consume any number of event messages, until the connection is terminated. Push model event consumers can consume events when they are invoked with the

`CosNotifyComm::StructuredPushConsumer::push_structured_event()` operation. Pull model event consumers can consume events using either the `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` or `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event()` operation on their event channel.

The connection with the event channel can be terminated by either the consumer or by the event channel. If the consumer is using the push model, it can disconnect by invoking the

`CosNotifyComm::StructuredPushSupplier::disconnect_structured_push_supplier()` operation on the event channel, and the event channel can disconnect the consumer by invoking the

`CosNotifyComm::StructuredPushConsumer::disconnect_structured_push_consumer()` operation on the consumer. If the consumer is using the pull model it can disconnect by invoking the

`CosNotifyComm::StructuredPullSupplier::disconnect_structured_pull_supplier()` operation on the event channel, and the event channel can disconnect the

supplier by invoking the `CosNotifyComm::StructuredPullConsumer::disconnect_structured_pull_consumer()` operation on the consumer.

As event channel connections are not retained persistently, if the event channel fails, the connection will have to be reestablished. If the event channel fails, pull model event consumer will be notified in an exception on their next `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` or `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event()` request. Event consumers should catch the `CosEventComm::Disconnected` exception. This is an indication that the event channel has failed. At that point, the consumer should reconnect before re-pulling the event.

Push model event consumers are never notified if an event channel fails. Instead, they will simply stop receiving any further events. If this is an issue for the consumer, it can periodically attempt to reconnect with the event channel. If the event channel has not failed then the reconnect attempt will simply raise the `CosEventChannelAdmin::AlreadyConnected` exception with no other consequences.

Event consumers can reside in pure clients. However, since pure clients cannot export object references, then event consumer in pure clients are constrained to only use the pull model of interaction. Further, pure client event consumers will not be notified if the event channel decides to disconnect. The event consumer is left to detect the loss of connection with the `CosEventComm::Disconnected` exception raised on their next pull or `try_pull` request.

Consuming events

This procedure demonstrates how to consume an event from an event channel. To consume an event, you have to first locate and connect to an event channel that is intended to handle your event topic. The Event Service supports the push and pull models of interaction between event consumers and event channels. If your event consumer will reside in a pure client, you must use the pull model.

Consume an event from the event channel using the following steps:

1. Locate an event channel. You will need an event channel to connect with. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. In addition, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.

2. If the consumer is interested in filtering events then it should create a filter object, add its required constraints to the filter object and use it to connect to the event channel.
3. Connect to the event channel. Follow the procedure described in “Connecting to an event channel” on page 74 to connect with the event channel.
4. Consume the event. As a pull model consumer, you can use the `CosNotifyComm::StructuredPullSupplier::pull_structured_event()` or `CosNotifyComm::StructuredPullSupplier::try_pull_structured_event()` method to consume an event. If you use the pull method, your request will block until an event is ready to be returned. If you use the `try_pull` method, your request will always return immediately, and will indicate in a boolean return value whether an event was available to return. As a push model consumer, you must implement the `CosNotifyComm::StructuredPushConsumer::push_structured_event()` method. This method will be invoked whenever an event is available for you to consume.

The following example demonstrates how to connect as a pull consumer to the `AgentActions` event channel created in “Creating an event channel” on page 71 and consume events using the pull method. This example assumes you are your own pull-consumer object, and that you are operating in a server process.

```
// Declare an intermediate Object, the AgentActions event channel,
// a consumer admin object and a pull supplier proxy.

CORBA::Object_var intermediateObject;
CosNotifyChannelAdmin::EventChannel_var agentActionsEC;
CosNotifyChannelAdmin::ConsumerAdmin_var agentActionsCA;
CosNotifyChannelAdmin::StructuredProxyPullSupplier_var agentActionsPPS;

// Declare the ID that the StructuredProxyPullSupplier object is assigned
// when it is created.
CosNotifyChannelAdmin::ProxyID SPPSid;

// Declare an event of type Structured Event that needs to be pulled.
CosNotification::StructuredEvent structuredEvent_var;

// Locate the AgentActions event channel, obtain the consumer admin,
// and obtain a pull supplier proxy.

intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/notify-channels/AgentActions");
agentActionsEC=
    INotifyChannelAdminManagedClient::EventChannelFactory::_narrow(
        intermediateObject);
agentActionsCA = CosNotifyChannelAdmin::SupplierAdmin::_narrow(
    agentActionsEC->default_consumer_admin());
```

```

agentActionsPPS =
    CosNotifyChannelAdmin::StructuredProxyPullSupplier::_narrow(
        agentActionCA->obtain_notification_pull_supplier(
            CosNotifyChannelAdmin::STRUCTURED_EVENT, SPPSid));

// Optional: create Filter object, define constraints in the Filter object,
//           add the filter object to the structuredproxyPullSupplier.
// Shown in the sample program at the end of the document.
.....

// Connect to the event channel proxy.

agentActionsPPS->connect_structured_pull_consumer(this);

// Consume an event.
// Pull the event

structuredEvent = agentActionsPPS->pull_structured_event();

// Handle the fields of the structuredEvent
...

```

Event channels

Event channels have two forms. In their abstract form, an event channel is a broker of event messages, consuming them from their connected suppliers, and supplying them to their connected consumers. In their concrete form, an event channel is composed of the following object types:

CosNotifyChannelAdmin::EventChannel

This is the anchor point for the other event channel objects. EventChannel objects define the abstract event channel, and are the objects registered in the system name space that enable you to find an event channel.

CosNotifyChannelAdmin::SupplierAdmin

SupplierAdmin objects broker connections between suppliers and the event channel. An event channel can have more than one SupplierAdmin.

CosNotifyChannelAdmin::ConsumerAdmin

ConsumerAdmin objects broker connections between consumers and the event channel. An event channel can have more than one ConsumerAdmin.

CosNotifyChannelAdmin::StructuredProxyPushConsumer

StructuredProxyPushConsumer objects define a specific connection between a structured supplier and the event channel. An instance of a

StructuredProxyPushConsumer is created for each structured supplier that connects to the event channel using the push-model of interaction.

CosNotifyChannelAdmin::StructuredProxyPullConsumer

StructuredProxyPullConsumer objects define a specific connection between a structured supplier and the event channel. An instance of a StructuredProxyPullConsumer is created for each structured supplier that connects to the event channel using the pull-model of interaction.

CosNotifyChannelAdmin::StructuredProxyPushSupplier

StructuredProxyPushSupplier objects define a specific connection between a structured consumer and the event channel. An instance of a StructuredProxyPushSupplier is created for each structured consumer that connects to the event channel using the push model of interaction.

CosNotifyChannelAdmin::StructuredProxyPullSupplier

StructuredProxyPullSupplier objects define a specific connection between a structured consumer and the event channel. An instance of a StructuredProxyPullSupplier is created for each structured consumer that connects to the event channel using the pull model of interaction.

Event channels can support multiple simultaneous connections of suppliers and consumers using any mixture of push model or pull model interactions. In Component Broker, event channels are managed objects with persistent references. Thus they must reside in a Component Broker server process, and can be registered in the system name space. All suppliers and consumers that resolve to the same event channel reference will get to the same instance of event channel. Event channels are automatically reactivated in the Component Broker server when they are first referenced.

As brokers of event messages, event channels are in fact both event consumers and event suppliers (consumers to their connected event suppliers, and suppliers to their connected event consumers), proxies are used to mediate between consumers and suppliers, and between push and pull models of interaction.

The transmission of events from the event channel to the consumers depend on the filter objects that are attached by the consumers to supplier proxies. Only the events that satisfy the constraints or conditions in the filter object will be transmitted to that consumer (which attached the filter object). If a consumer does not attach a filter object to the supplier proxy or if the filter object attached does not contain any constraints then all the events received by the event channel will be transmitted to that consumer. Also, the delivery of the event also depends on the quality of services (QoS) attached to the

message and/or to the proxy. For example, if StopTime is set in the QoS then the event channel will discard the event if it has not been delivered before the StopTime.

Also, in Component Broker, event channels are transient. They do not retain either supplier or consumer connections persistently, nor do they retain event messages persistently. Connections and pending event messages will be lost whenever the event channel is passivated or terminated.

Component Broker automatically creates a single default event channel that is registered in the system name space in /cell/resources/notify-channels/cell-default. This event channel can be used for any purpose, but to allow for proper interoperability between all of the applications that could end up using it, all event messages using the event channel in a production environment must follow the following structure:

```
module DefaultEventStructure {
    typedef struct Event_s {

        // Part 1: Header

        const unsigned short    version=1;

        // structure version

        string                  domain_type;;
        string                  event_type;
        string                  event_name;
        CosTrading::PropertySeq quality_of_services;
        CosTrading::PropertySeq filterable_data;

        // Part 2: Un-filterable Event Data

        any                     unfilterable_data;
    } StructuredEvent;
};
```

If you need an event channel within your application, you should introduce your own event channel for your own purposes. See “Event topics” on page 59 for more information that may be useful in deciding how to divide up your event system topology. Having done so, you should design an appropriate structure for your event messages, as discussed in “Structured events” on page 56. You can create a new event channel programmatically using the standard Component Broker programming model for managed objects, as described in “Creating an event channel” on page 71.

Having created the event channel you should be sure to register it in the system name space so that your suppliers and consumers can find it at run-time. This will be done for you automatically if you use the `INotifyChannelAdminManagedClient::EventChannelFactory::createVisibleEventChannel`

operation. Otherwise, you can use the `INotifyChannelAdminManagedClient::EventChannelFactory::createEventChannel()` operation and manually bind the event channel in the system name space.

You can also use the Component Broker System Management facility to configure a new event channel. This is described in more detail in “Configuring an event channel” on page 73. This process automatically registers the event channel in the system name space so that your suppliers and consumers can find it at run time. The Component Broker system name space includes naming contexts where you can register event channels by name at the following locations:

- `/host/resources/notify-channels/workgroup/resources/notify-channels`
- `/cell/resources/notify-channels`

Alternately, you can register your event channels in your own application name tree.

Locating an event channel

This procedure demonstrates how you can locate an event channel by name in the system name space. This is useful to both event suppliers and event consumers who are finding an event channel they can use to exchange event messages.

You can locate an event channel in the fashion demonstrated here only if the event channel already exists and has been bound in the system name space. You must know the name of the event channel to use this procedure.

Use the following steps to locate an event channel:

1. Determine the name of the event channel you want to use. This may be the name of the default event channel automatically created and registered in the system name space by Component Broker installation. Or it could be an event channel that your application created and bound in the system name space earlier.
2. Resolve the event channel from the system name space. Use the Naming Service operations to resolve the event channel from the system name space by the name that you determined in the previous step.

The default event channel is created automatically during Component Broker installation. The following example shows how to obtain the default event channel:

```
// Declare the targeted event channel

CORBA::Object_var intermediateObject;
CosNotifyChannelAdmin::EventChannel_var defaultEC;
```

```

// Obtain the default event channel and narrow it to an event
// channel

intermediateObject=CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/cell/resources/notify-channels/cell-default");
defaultEC=
    CosNotifyChannelAdmin::EventChannel::_narrow(intermediateObject);

```

Creating an event channel

An event channel is created essentially in the same way that any other managed object is created, that is, by first obtaining a factory finder with a desired location scope. By extension, such a factory finder must exist. Component Broker provides a set of default factory-finders that it creates and registers in the system name space during Component Broker installation.

These are found in the following locations:

- /host/resources/factory-finders/host-scope
- /workgroup/resources/factory-finders/workgroup-scope
- /cell/resources/factory-finders/cell-scope

In addition, Component Broker automatically creates a home for event channels (called notify—channels in order to differentiate from the Event Service event-channels) in every server process. However, the least granularity of scoping that any of the default factory finders supports is a host scope. Therefore, if you want the event channel created in a more specific server you must create a new factory finder with a corresponding location scope that narrows to that server.

Create an event channel using the following steps:

1. Obtain a factory finder with the desired location scope. You will need a factory finder with which to create your new event channel. Alternately, you can create a non-managed factory finder to use for finding the new event channel's factory.
2. Use the factory finder to find a factory (a Home) for event channels. Having obtained a factory finder, you can now use it to find a factory for factory finders. You do this with the `find_factory()`, `find_factories()`, `find_factory_from_string()`, or `find_factories_from_string()` methods. The event channels interface is `INotifyChannelAdminManagedClient::EventChannel.object`. This is the principal interface name that you will pass in to the `find_factory*` or `find_factories*` methods.
3. Create the event channel. The factory specialization for event channels introduces the `createEventChannel` and `createVisibleEventChannel` methods which you can use to create the new event channel. You will

have to narrow to the `INotifyChannelAdminManagedClient::EventChannelFactory` interface to use either of these new methods.

4. Register the event channel in the system name space. This step is optional. When you've created an event channel, often you will want to retain this object for future use in your application to be shared by event suppliers and consumers. How you do this in your application is up to you, but the typical approach is to bind the event channel in the system name space. Again, where you bind it in the system name space is up to you. One obvious and commonly used choice is to bind it in `/host/resources/notify-channels`, `/workgroup/resources/notify-channels`, or `/cell/resources/notify-channels` depending on what visibility you want to give it. Another choice is to bind it under your own application, contexts within `/host/applications`, `/workgroup/applications`, or `/cell/applications`. Typically event channels are bound with the event topic name for the types of events they're intended to broker.

If you use the

```
INotifyChannelAdminManagedClient::EventChannelFactory::  
    createVisibleEventChannel
```

method, you can specify the relative name (the event channel's topic name), and a boolean flag for the name spaces in which you want the event channel registered. The specialized home will automatically register the new event channel into the name space in the indicated locations.

The following example demonstrates creating a new event channel, and binding it in the system name space under `/host/resources/notify-channels` with the name `AgentActions`. The event channel is created within the local host.

```
// Declare an intermediate Object, the event-channels naming  
// context, the targeted factory finder where the new event  
// channel will be created, a event channel factory, and the  
// new event channel.  
  
CORBA::Object_var intermediateObject;  
IExtendedNaming::NamingContext_var eventChannelsNC;  
IExtendedLifeCycle::FactoryFinder_var hostScopeFF;  
INotifyChannelAdminManagedClient::EventChannelFactory_var factoryOfEC;  
CosNotifyChannelAdmin::EventChannel_var myNewEC;  
ByteString_var key;  
  
// Declare the ID for the event channel being created.  
CosNotifyChannelAdmin::ChannelID cid;  
  
// Obtain the default factory finder with a host-scope and  
// narrow to an IExtendedLifeCycle::FactoryFinder so that we  
// can use the find_factory_from_string operation.
```

```

CBSeriesGlobal::Initialize();
//Initialize the CBSeries environment

intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/factory-finders/host-scope");
hostScopeFF = IExtendedLifeCycle::FactoryFinder::_narrow(
    intermediateObject);

// Find a factory for event channels and narrow to the factory
// specialization

intermediateObject = hostScopeFF->find_factory_from_string(
    "INotifyChannelAdminManagedClient::EventChannel.object interface");
factoryOfEC =
    INotifyChannelAdminManagedClient::EventChannelFactory::_narrow
    (intermediateObject);

// Create the required default quality of services (QoS)
// Shown in the sample program at the end of the document.
CosTrading::PropertySeq qos;
.....

// Use the factory to create a new event channel.

myNewEC = factoryOfEC->createEventChannel(key, qos, NULL, cid);

// Bind the new event channel in the system name space.

intermediateObject = CBSeriesGlobal::nameService->resolve(
    "/host/resources/notify-channels");
eventChannelsNC =
    IExtendedNaming::NamingContext::_narrow(intermediateObject);
eventChannelsNC->bind_with_string("AgentActions", myNewEC);

```

The last four statements could have been combined into the following statement:

```

myNewEC = factoryOfEC->createVisibleEventChannel(key,
    "AgentActions", 0, 1, 0);

```

Configuring an event channel

This procedure demonstrates how to configure a new event channel using the Component Broker System Management facility. This is useful for when you need an event to support a new event topic. This procedure is entirely administrative and does not involve any programming.

Configure a new event channel using the following steps:

1. Navigate to the desired server. The new event channel must be configured within a specific server. Use the System Management user interface to navigate to the host and server on which you want the factory finder configured.
2. Create an event channel model. Insert an event channel model within the server.
3. Set event channel model attributes. Use the attributes notebook to fill in the event channel model attribute value. This will include the System Management name for the event channel (do not confuse this with the name of the event channel in the system name space, although these can be set to the same value), the name of the event channel in the system name space, and the Home of the event channel.
4. Apply the model changes. Select the apply function for the server, host, or zone where you've created the event channel model. This results in a corresponding event channel object being automatically created in the targeted server and registered in the system name space.

Connecting to an event channel

This procedure demonstrates how to connect to an event channel. You need to connect to an event channel before you can supply or consume events. The event channel must have already been created and registered in a way that you can find it.

Connect to an event channel using the following steps:

1. Locate an event channel. You will need an event channel with which to connect. The easiest approach is to simply obtain an already existing event channel that has been registered in the system name space. Otherwise, you can create a new event channel, or use the Component Broker System Management facilities to configure a new event.
2. Acquire an event channel administration object. Use the appropriate method from the following methods to get the required administrative object:
 - If you are an event supplier, use the `default_supplier_admin()` method or the `new_for_suppliers()` operation to get back a `CosNotifyChannelAdmin::SupplierAdmin` object.
 - If you are an event consumer, use the `default_consumer_admin()` method or the `new_for_consumers()` operation to get back a `CosNotifyChannelAdmin::ConsumerAdmin` object.
3. Acquire an event channel proxy object. Use the appropriate method from the following methods to get the required proxy object:
 - If you are a structured supplier using the push model, then use the `obtain_notification_push_consumer()` operation to get back a `CosNotifyChannelAdmin::StructuredProxyPushConsumer` object.

- If you are a structured supplier using the pull model, then use the `obtain_notification_pull_consumer()` operation to get back a `CosNotifyChannelAdmin::StructuredProxyPullConsumer` object.
 - If you are a structured consumer using the push model, then use the `obtain_notification_push_supplier()` operation to get back a `CosNotifyChannelAdmin::StructuredProxyPushSupplier` object.
 - If you are a structured consumer using the pull model, then use the `obtain_notification_pull_supplier()` operation to get back a `CosNotifyChannelAdmin::StructuredProxyPullSupplier` object.
4. Connect with the proxy. Use the appropriate method from the following to connect with the proxies:
- If you are a structured push supplier, then use the `connect_structured_push_supplier()` method on your `StructuredProxyPushConsumer`.
 - If you are a structured pull supplier, then use the `connect_structured_pull_supplier()` method on your `StructuredProxyPullConsumer`.
 - If you are a structured push consumer, then use the `connect_structured_push_consumer()` method on your `StructuredProxyPushSupplier`.
 - If you are a structured pull consumer, then use the `connect_structured_pull_consumer()` method on your `StructuredProxyPullSupplier`.

Normally, you have to supply a reference to yourself, namely your supplier or consumer object, on the connect request. This is used either to pull or push on you, as well as to inform you when the connection is being terminated. However, if you are connecting from a pure-client you are not able to use the `connect_structured_pull_supplier()` or `connect_structured_push_consumer()` methods. Also, you will have to supply a NULL object reference in either the `connect_structured_push_supplier()` or `connect_structured_pull_consumer()` request. This basically indicates to the event channel that you are operating from a pure-client and avoids the issuing of any disconnect requests.

The following example demonstrates how to connect as a push supplier to the `AgentActions` event channel that was created in “Creating an event channel” on page 71. This example assumes you are your own push-supplier object, and that you are operating in a server process.

```
// Declare an intermediate Object, the AgentActions event
// channel, a supplier admin object, a push consumer proxy.

CORBA::Object_var intermediateObject;
CosNotifyChannelAdmin::EventChannel_var agentActionsEC;
CosNotifyChannelAdmin::SupplierAdmin_var agentActionsSA;
CosNotifyChannelAdmin::StructuredProxyPushConsumer_var agentActionsSPPC;
```

```

// Declare the ID that the StructuredProxyPushConsumer object is assigned
// when it is created.
CosNotifyChannelAdmin::ProxyID SPPCid;

// Define an event of type Structured Event that needs to be pushed.
CosNotification::StructuredEvent structuredEvent_var =
    new CosNotification::StructuredEvent;

//Initialize the CBSeries environment
CBSeriesGlobal::Initialize();

// Locate the AgentActions event channel, obtain the supplier
// admin, and obtain a structured push consumer proxy.
intermediateObject = CBSeriesGlobal::nameService()
    ->resolve_with_string(
        "/host/resources/notify-channels/AgentActions");
agentActionsEC=
    INotifyChannelAdminManagedClient::EventChannelFactory::_narrow(
        intermediateObject);
agentActionsSA =
    CosNotifyChannelAdmin::SupplierAdmin::_narrow(
        agentActionsEC->default_supplier_admin());
agentActionsPPC =
    CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(
        agentActionSA->obtain_notification_push_consumer(
            CosNotifyChannelAdmin::STRUCTURED_EVENT, SPPCid));

// Connect to the event channel proxy.

agentActionsPPC->connect_structured_push_supplier(this);

```

Disconnecting from an event channel

Disconnecting from an event channel is useful for when you want to stop supplying or consuming events. It essentially un-registers your event supplier or event consumer with the event channel. After disconnecting, push model suppliers can no longer push, pull model suppliers will no longer be pulled, push model consumers will no longer be pushed, and pull model consumers can no longer pull.

If you are already connected to an Event Channel, use the appropriate method from the following methods to disconnect:

- As a structured push supplier, use the


```

CosNotifyComm::StructuredPushConsumer::
    disconnect_structured_push_consumer()

```

method on your structured push consumer proxy.

- As a structured pull supplier, use the


```
CosNotifyComm::StructuredPullConsumer::  
    disconnect_structured_pull_consumer()
```

method on your structured pull consumer proxy.

- As a structured push consumer, use the

```
CosNotifyComm::StructuredPushSupplier::  
    disconnect_structured_push_supplier()
```

method on your structured push supplier proxy.

- As a structured pull consumer, use the

```
CosNotifyComm::StructuredPullSupplier::  
    disconnect_structured_pull_supplier()
```

method on your structured pull supplier proxy.

The following example demonstrates how to disconnect from the AgentActions event channel that was previously connected to in “Connecting to an event channel” on page 74. This example assumes that this is being done from a push supplier.

```
// Assuming we've already connected to the AgentActions event  
// channel as push supplier  
...  
agentActionPPC->disconnect_structured_push_consumer();
```

FilterFactory and filters

The FilterFactory creates the filter objects. The Filter encapsulates the constraints which will be used by a proxy object associated with a notification channel in order to make decisions about which events to forward, and which to discard. Each object supporting the Filter interface can encapsulate a sequence of any number of constraints. Each event received by a proxy object which has one or more objects supporting the Filter interface associated with it, must satisfy at least one of the constraints associated with one of its associated Filter objects in order to be forwarded (either to another proxy object or to the consumer, depending on the type of proxy the filter is associated with), otherwise it will be discarded.

Each constraint encapsulated by a filter object is a structure comprised of two main components. The first component is a sequence of data structures, each of which indicates an event type comprised of a domain and a type name. The second component is a boolean expression over the properties of an event, expressed in the constraint grammar.

For a given constraint, the sequence of event type structures in the first component nominates a set of event types to which the constraint expression in the second component applies. Each element of the sequence can contain

strings which will be matched for equality against the `domain_name` and `type_name` fields of each event being evaluated by the filter object when determining if the boolean expression should be applied to the event, or the event should simply be discarded without even attempting to apply the boolean expression.

The constraint expressions associated with a particular object supporting the Filter are expressed as strings which obey the syntax of a particular constraint grammar (that is, a BNF). This implementation supports constraint expressions expressed in the constraint grammar shown in the “Appendix A. Default Filter Constraint Language” of the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*.

The Filter interface supports the operations required to manage the constraints associated with an object instance which supports the interface, along with a readonly attribute which identifies the particular constraint grammar in which the constraints encapsulated by this object have meaning. In addition, the Filter supports the `match_structured()` operation which can be invoked by an associated proxy object upon receipt of an event to determine if the event should be forwarded or discarded, based on whether or not the event satisfies at least one criteria encapsulated by the filter object.

Create, define and attach a Filter to a proxy using the following steps:

1. Obtain a Filter Factory. This is similar to the Event Channel Factory creation process as explained before. The following steps illustrate this process.

```
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var pxyPushS;  
CORBA::Object_var intermediateObject;  
IExtendedLifeCycle::FactoryFinder_var hostScopeFF;  
INotifyFilterManagedClient::FilterFactory_var fFactory = NULL;  
CosNotifyFilter::Filter_var fi = NULL;  
CosNotifyFilter::ConstraintExpSeq *cl;  
CosNotifyFilter::ConstraintInfoSeq *cis;  
CosNotifyFilter::FilterID fid;  
    // initial ORB  
CBSeriesGlobal::Initialize();  
    // obtain the default factory finder with a host scope  
intermediateObject =  
    CBSeriesGlobal::nameService()->resolve_with_string(  
        "host/resources/factory-finders/host-scope");  
    // narrow to a factory finder  
hostScopeFF = IExtendedLifeCycle::FactoryFinder::_narrow(  
    sintermiateObject);  
    // find the filter factory  
intermediateObject = hostScopeFF->find_factory_from_string(  
    "INotifyFilterManagedClient::Filter.object interface");  
    // narrow to the filter factory  
fFactory = INotifyFilterManagedClient::FilterFactory::_narrow(  
    intermediateObject);
```

2. Create a Filter from the Filter Factory. The `create_filter` method is used to create a filter. The constraint grammar is passed as the input string parameter.

```
fi = fFactory->create_filter("IBM_NTF_CTG");
```

3. Define the contents (constraints) of the filter and add them to the filter. The `add_constraints` method is used to add the constraints to the filter. Events will be delivered to the consumers only when these constraints of the filter match with that of the event.

```
CosNotifyFilter::ConstraintExpSeq *c1 =  
    new CosNotifyFilter::ConstraintExpSeq;  
(*c1).length(1);  
(*c1)[0].constraint_expr = CORBA::string_alloc(100);  
    // define constraint  
strcpy((*c1)[0].constraint_expr, "($COMPANY == IBM) and (  
    $PRICE > 150)");  
    // add constraint to the filter  
cis = fi->add_constraints(*c1);
```

4. Add the filter to the proxy which is expected to deliver the events. The client after creating the filter with the necessary constraints should use the `add_filter` method of the appropriate proxy suppliers to attach the filter to the proxy. Once attached only the events whose filterable data matches the constraints in the filter will be delivered to the client by the proxy.

```
id = pxyPushS->add_filter(fi);
```

The `Customer_Impl()` method in the following sample illustrates the procedure for creating, defining and attaching a Filter to a proxy.

Managed object-based sample

In this scenario a Push Supplier is created as a Component Broker server. The Push Supplier object is called `Account`. This object contains two attributes. One is the `accountNumber` to hold an account number. This will be the primary key. The other attribute in this `Account` object is called `balance` which holds the current content of this `accountNumber` in the bank. (We discuss only one account in this sample.) The `Account` object also has a method called `changeBalance` which when invoked will update the `balance` and push the new `balance` to the event channel (The default event channel is considered in this sample. For a user created event channel, see “User-defined event channel” on page 96 for further information.) The `changeBalance` method will be invoked by a pure C++ client (called `PushSupplier`) which prompts the user with the amount to be added/subtracted from the `balance`.

On the consumer side, a Pull Consumer is implemented as a pure C++ client (called `PullConsumer`). This consumer creates a filter with the necessary requirements (in this case to notify it if the `balance` is less than \$1,000) and pulls the events from the event channel using the `try_pull` method.

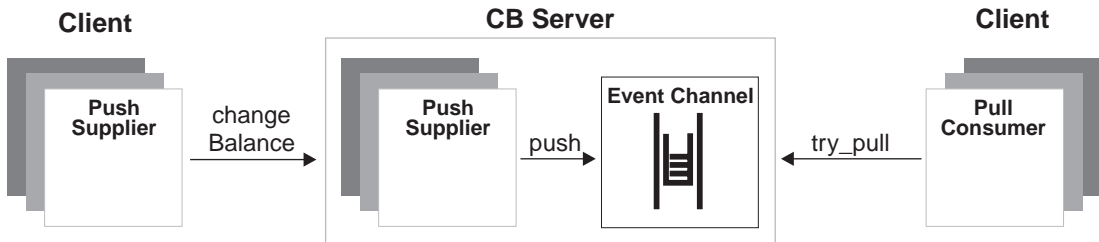


Figure 3. Notification Service managed object-based sample

We will go through the above process in three steps. In step one, we implement and build the server. In step two, we implement and build the PushSupplier and the PullConsumer clients. In step three, we deploy and run the sample.

If you want your model to be transparent across different platforms, you can create your own containers instead of using the defaults. To create a Cachetransient Container sample, follow these steps:

1. On the Container Definition, select **Add Container Instance**
2. Name the container PushsupplierContainer with the description Notification Push Supplier sample Container and click **Next**.
3. Accept the Service Details page defaults and click **Next**.
4. On the Service page, select **Use no Object Services** and click **Next**.
5. On the Data Access Patterns page, select **Caching** for the business object, and select **Local copy** for the data object. These two selections must match the selection when the business object and data object were created.


The following instructions are for Windows NT and AIX. Some platform-specific notes are included.

Step 1: Implementing the application server (push supplier server object)

Implementing the server application consists of two major steps:

1. Specifying all the interfaces and implementations of the business object and the data object, including the helper objects using the Component Broker Object Builder (OB).
2. Building the application by running the code emitter, compiler and linker.

Creating the model using the Object Builder

 The DLL files are called shared library files and are in the format lib*.so. For any reference to a DLL file, substitute shared library file.

Note: The following instructions assume that the Object Builder is up and running.

1. **Create a new file Account.**
 - a. Select **User-Defined Business Objects > Add File**.
 - b. Type Account.
 - c. Click **Finish**.
2. **Add a new interface to the file.**
 - a. Select **Account > Add Interface**.
 - b. Type AccountInterface.
 - c. Click **Next**.
 - d. Add an exception, **NotEnough** using the Constructs page of the AccountInterface.
 - 1) On the Constructs page, select **Constructs > Add Exception**.
 - 2) Type NotEnough.
 - 3) Click **Refresh**.
 - 4) Click **Next**.
 - e. Add a second parent interface, **CosNotifyComm::StructuredPushSupplier**, using the Interface Inheritance page of the AccountInterface wizard.
 - 1) On the Interface Inheritance page, select **Parents > Add**.
 - 2) Type **CosNotifyComm CosNotifyComm::StructuredPushSupplier** for Parent Interface.
 - 3) Click **Refresh**.
 - 4) Click **Finish**.
3. **Add properties to the interface.**
 - a. Select **AccountInterface > Properties**.
 - b. Go to the Attributes page, select **Attributes > Add**.
 - 1) Add the attribute *accountNumber* using the following procedure:
 - a) Type *accountNumber* in the **Attribute Name** field.
 - b) Select **String** in the **Type** list.
 - c) Type 10 in the **Size** field.
 - d) Click **Add Another**.
 - 2) Add the attribute *balance* using the following procedure:
 - a) Type *balance* in the **Attribute Name** field.
 - b) Select **double** in the **Type** list.
 - c) Click **Refresh**.
 - d) Click **Next**.
 - c. Add the method *changeBalance* using the following procedure:

- 1) On the Methods page, select **Method > Add**.
- 2) Type `changeBalance` in the **Method Name** field.
 - a) Select **Void** in the **Return Type** list.
 - b) Click **Refresh**.
 - c) Select **Parameters > Add**.
 - d) Type `anAmount` in the **Parameter Name** field.
 - e) Select **Double** from the **Type** list.
 - f) Click **Refresh**.
 - g) Select **Exceptions > Add**.
 - h) Select **Account AccountInterface NotEnough** for Exception Name.
 - i) Click **Refresh**.
 - j) Click **Finish**.
4. **Create a Key helper using the `accountNumber` as the Primary Key.**
 - a. Select **AccountInterface > Add Key**.
 - b. Click on `accountNumber` on the left.
 - c. Click **>>**.
 - d. Click **Finish**.
5. **Create a Copy helper with the attribute of `accountNumber`.**
 - a. Select **AccountInterface > Add Copy Helper**.
 - b. Select `accountNumber`.
 - c. Click **>>**.
 - d. Click **Finish**.
6. **Add Business Object implementation.**
 - a. Select **AccountInterface > Add Implementation**.
 - b. Click **Finish**.
7. **Add the StructuredProxyPushConsumer to the Business Object.**
 - a. Select **AccountInterfaceBO > Properties**.
 - 1) On the Attributes page, select **Attributes > Add**.
 - 2) Type `ProxyPushConsumer` in the **Attribute Name** field.
 - 3) Type `CosNotifyChannelAdmin`
`CosNotifyChannelAdmin::StructuredProxyPushConsumer` in the **Type** field.
 - 4) Click **Refresh**.
 - 5) Click **Next**.
 - b. Override methods `disconnect_structured_push_supplier` and `subscription_change`.
 - 1) Go to the Methods to Override page.

- 2) Select `disconnect_structured_push_supplier` and `subscription_change` in the **Method** list on the left and move them to the right by clicking `>>`.
 - 3) Click **Next**.
- c. Add attributes `accountNumber` and `balance` to the data object interface.
- 1) Select `accountNumber` and `balance` in the Business Object Attributes list on the left and move them to the right by clicking `>>`.
 - 2) Click **Finish**.
- d. Add implementation to `changeBalance()`, `initForCreation()`, and `uninitForDestruction()` methods. The implementation of these three methods are shown below. Copy this code into three different files in the Model directory and add them to the Model using the following procedure:
- 1) Select **changeBalance** in the Methods window > **Properties**
 - 2) Select **Use an external file**.
 - 3) Click **Browse**, and select the file that implements the `changeBalance()` method in the Model directory.
 - 4) Repeat for `initForCreation()` and `uninitForDestruction()` methods.

changeBalance()

```

CORBA::Double newBalance;
CosNotification::StructuredEvent *event;
CORBA::Short Priority;
time_t Timeout;
tm t1;

// change the balance
newBalance = iDataObject->balance() + anAmount;

if (newBalance < 0)
    throw NotEnough();
else // update the balance and generate event
{
    iDataObject->balance(newBalance);
    event = new CosNotification::StructuredEvent;
        // Define the event
    event->header.fixed_header.domain_type = CORBA::string_alloc(100);
    strcpy(event->header.fixed_header.domain_type, "dt");
    event->header.fixed_header.event_type = CORBA::string_alloc(100);
    strcpy(event->header.fixed_header.event_type, "et");
    event->header.fixed_header.event_name = CORBA::string_alloc(100);
    strcpy(event->header.fixed_header.event_name, "en");
        // Set the Quality of services (QoS)
    event->header.variable_header.length(3);
        // Set Priority
    event->header.variable_header[0].name = CORBA::string_alloc(100);
    strcpy(
        event->header.variable_header[0].name, CosNotification::Priority);
    Priority = 6;
    event->header.variable_header[0].value <= Priority;

```

```

        // Set Timeout = 2 days in 100 nanoseconds
event->header.variable_header[1].name = CORBA::string_alloc(100);
strcpy(
    event->header.variable_header[1].name, CosNotification::Timeout);
Timeout = 2*24*60*60*100000000;
event->header.variable_header[1].value <<= Timeout;
        // Set StopTime = 2005 Dec 4, 12.01
event->header.variable_header[2].name = CORBA::string_alloc(100);
strcpy(
    event->header.variable_header[2].name, CosNotification::StopTime);
t1.tm_year = 99;
t1.tm_mon = 11;    // month is zero-based
t1.tm_mday = 4;    // day is one-based
t1.tm_hour = 12;
t1.tm_min = 1;
t1.tm_sec = 0;
CORBA::ULong utmp = mktime(&t1);
event->header.variable_header[2].value <<= utmp;
        // Set filterable data
event->filterable_data.length(2);
event->filterable_data[0].name = CORBA::string_alloc(20);
strcpy(event->filterable_data[0].name, "accountNumber");
event->filterable_data[0].value <<= "123456";
event->filterable_data[1].name = CORBA::string_alloc(20);
strcpy(event->filterable_data[1].name, "balance");
event->filterable_data[1].value <<= newBalance;
        // Push the event to the event channel
iProxyPushConsumer->push_structured_event(*event);
}

```

initForCreation()

```

CosNotifyChannelAdmin::EventChannel_var ec;
CosNotifyChannelAdmin::SupplierAdmin_var sa;
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxyConsumer_var proxyConsumerVar;
CORBA::Object_var it;

iDataObject = AccountInterfaceDO::_narrow(theDO);
AccountInterfaceBO_Impl::initializeState();
        // initialize the CBSeries environment
CBSeriesGlobal::Initialize();
        // get default event channel
it = CBSeriesGlobal::nameService()->resolve_with_string(
    "cell/resources/notify-channels/cell-default");
ec = CosNotifyChannelAdmin::EventChannel::_narrow(it);
        // get default SupplierAdmin
sa = ec->default_supplier_admin();
        // get ProxyPushConsumer
proxyConsumerVar = sa->obtain_notification_push_consumer(
    CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);
iPushConsumer =
    CosNotifyChannelAdmin::StructuredProxyPushConsumer::_narrow(

```



```

        proxyConsumerVar);
        // connect to the ProxyPushConsumer
        iProxyPushConsumer->connect_structured_push_supplier(this);

```

Note: Use Object Builder generated code and add changes on top of it.

uninitForDestruction()

```

        iProxyPushConsumer->disconnect_structured_push_consumer();

```

Note: Use Object Builder generated code and add changes on top of it.

8. Add additional include files for AccountInterfaceBO.

- a. Select **AccountInterfaceBO**
- b. At the bottom of the Methods window, select **File Adornments > Prologue**.
- c. Type the following in the bottom window:

```

#include <CBSeriesGlobal.hh>
#include <INotifyChannelAdminManagedClient.hh>
#include <IExtendedLifeCycle.hh>
#include <time.h>

```

9. Add a Data Object Implementation.

- a. Select **AccountInterfaceDO > Add Implementation**.
- b. Go to the Behavior page.
- c. Select the following: **BOIM with any key, Transient, Home name and key**.
- d. Click **Finish**.

10. Add a Managed Object to the AccountInterfaceBO.

- a. Select **AccountInterfaceBO > Add Managed Object**.
- b. Click **Finish**.

11. Generate the code.

Select **User-Defined Business Objects > Generate > All**.

12. Build the Server Application. In the Build Configuration folder, implement the following steps.

AIX only: The DLL files are called shared library files and are in the format lib*.so. For any reference to a DLL file, substitute shared library file.

- a. Add a client DLL named accountC and include all account objects in the DLL.
 - 1) Select **Build Configuration > Add Client DLL**.
 - 2) Type accountC in the **Name** field.
 - 3) Click **Next**.

- b. On the Client Source Files page, select the following and move from **Items Available** (on left) to **Items Chosen** (on right) by clicking >>.
 - 1) **AccountCopy**
 - 2) **AccountKey**
 - 3) **Account**

Click **Finish**.

- c. Add a server DLL named accountS and include all account objects to the DLL. Also, link the client library accountC to it.
 - 1) Select **Build Configuration > Add Server DLL**.
 - a) Type accountS in the **Name** field.
 - b) Click **Next**.
 - 2) On Server Source Files page, perform the following tasks:
 - a) Click **All>>** to select all Account* objects to move.
 - b) Click **Next**.
 - 3) On the Libraries to Link With page, select the **accountC**, and click >>.
 - 4) Click **Finish**.
 - d. Generate the makefiles. Select **Build Configuration > Generate > All > C++ Default Targets**.
 - e. Build the server application. Select **Build Configuration > Build > Out-of-Data Targets > C++**.
13. **Generate the DDL using procedures from the *WebSphere Application Server Enterprise Edition Component Broker Application Development Tools Guide*.** The following instructions were used to generate the DDL for our test. However, since these instructions can change you need to always refer to the *WebSphere Application Server Enterprise Edition Component Broker Application Development Tools Guide* to get the latest information.
- a. Add Application Family.
 - 1) Select **Application Configuration > Add Application Family**.
 - 2) In the **Name** field, type AccountAppFam.
 - 3) Click **Finish**.
 - b. Define a Server Application.
 - 1) Select **AccountAppFam > Add Application**.
 - 2) In the **Application Name** field, type accountS.
 - 3) Click **Finish**.
 - c. Configuring the managed object.
 - 1) Open AccountAppFam > Select **accountS > Add Managed Object > In the Managed Object** field to ensure that **AccountInterfaceMO** is selected.

- 2) Click **Next**.
 - 3) In Data Object Implementations page > **Add Another**.
 - 4) Select **AccountDOImpl AccountInterfaceDOImpl** in the **Data Object Implementation** field if not already selected.
 - 5) Select **accountS** in the **DLL for Data Object Implementation** field, if not already selected.
 - 6) Click **Refresh**.
 - 7) Click **Finish**.
- d. Generate the DDL File. Open AccountAppFam > **Generate**.

Note: The DDL files should be generated in the Working/NT/PRODUCTION/AccountAppFam directory for Windows NT and Working/AIX/AccountAppFam directory for AIX.

Step 2 : Implementing the C++ client code (PushSupplier and PullConsumer)

This section provides sample code for the PushSupplier and the PullConsumer.

PushSupplier C++ client code

In this sample, the Account object in the server is going to generate events on the event channel each time its changeBalance method is invoked. As mentioned earlier, a pure C++ client (PushSupplier) is used to invoke the changeBalance method on the Account object. The PushSupplier program prompts the user for the amount of the *balance* to be changed. The *balance* is then passed as a parameter to the changeBalance method. The following code illustrates the implementation of this client.

The PushSupplier client code creates an object using AccountKey and uses the object to call the changeBalance method in the Account application server that then pushes the event (*balance* in this case) to the Event Channel. When this code is run it prompts for the user to input an amount to be added to the *balance* of the Account.

```
#include <CBSeriesGlobal.hh>
#include <IExtendedLifeCycle.hh>
#include <IManagedClient.hh>
#include "Account.hh"
#include "AccountKey.hh"

int main(int argc, char *argv[])
{
    CORBA::ORB_ptr op;
    IExtendedLifeCycle::FactoryFinder_var myFinder;
    IManagedClient::IHome_var accountHome;
    AccountInterfaceKey_var theKey;
```

```

ByteString* theKeyString;
IManagedClient::IManageable_var moPtr;
AccountInterface_var account;
CORBA::Double anAmount;
CORBA::Object_var it;
// Global Initializer
CBSeriesGlobal::Initialize();
op = CBSeriesGlobal::orb();
// find a Factory
it = CBSeriesGlobal::nameService()->resolve_with_string(
    "host/resources/factory-finders/host-scope");
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow(it);
it = myFinder->find_factory_from_string(
    "AccountInterface.object interface");
accountHome = IManagedClient::IHome::_narrow(it);
// create an instance of a AccountInterfaceKey
theKey = AccountInterfaceKey::_create();
// Set the key to ("123456789");
theKey->accountNumber("123456789");
// create a account from the Key class
theKeyString = theKey->toString();
// Checking to see if the object already exists
try
{
    cout << "Checking to see if the object already exists on the Server"
        << endl;
    moPtr = accountHome->findByPrimaryKeyString(*theKeyString);
}
catch (IManagedClient::INoObjectWKey &nowk)
{
    cout << "Expected ERROR in findByPrimaryKeyString call: "
        << nowk.id() << endl;
}
if (!moPtr)
    moPtr = accountHome->createFromPrimaryKeyString(*theKeyString);
// Narrow to the Account interface
account = AccountInterface::_narrow(moPtr);

try
{
    while (1)
    {
        cout << "Enter the amount of Balance to be changed" << endl;
        cin >> anAmount;
        cout << "anAmount = " << anAmount << endl;
        if (anAmount != 0)
            account->changeBalance(anAmount);
    }
}
catch (AccountInterface::NotEnough)
{
    cout << "NotEnough exception thrown" << endl;
}
catch (...)
{

```

```

        cout << "Unknown Exception thrown" << endl;
    }

    CORBA::release(op);
    delete theKeyString;
    return 0;
}

```

PullConsumer C++ client code

The consumer client code is also implemented in C++. The consumer in this case is only interested in the event the *balance* in the account falls below \$1000. A filter is created to have this requirement so that the consumer client will be notified in case the *balance* is less than \$1000. In this sample the consumer uses the `try_pull` method in a loop to continuously monitor the events from the event channel (*balance* in this case). The following code illustrates the implementation of this client.

The PullConsumer client code pulls events from the event channel by invoking the `try_pull_structured_event` method. A filter is created with the constraints such that it receives events only when the *accountNumber* is 123456 and the *balance* in the account is less than \$1000.

```

#ifdef _WIN32
#include <windows.h>
#endif
#include <CBSeriesGlobal.hh>
#include <CosNotifyChannelAdmin.hh>
#include <INotifyFilterManagedClient.hh>
#include <IExtendedLifeCycle.hh>
#include <IExtendedNaming.hh>

int main( )
{
    CORBA::ORB_ptr orb;
    IExtendedNaming::NamingContext_var rootNC;
    CosNotifyChannelAdmin::EventChannel_var ec = NULL;
    CosNotifyChannelAdmin::ConsumerAdmin_var ca;
    CosNotifyChannelAdmin::ProxyID proxy_id;
    CosNotifyChannelAdmin::StructuredProxyPullSupplier_var PxyPullS;
    IExtendedLifeCycle::FactoryFinder_var hostScopeFF;
    INotifyFilterManagedClient::FilterFactory_var ff = NULL;
    CosNotifyFilter::Filter_ptr fi;
    CosNotifyFilter::ConstraintExpSeq *cl =
        new CosNotifyFilter::ConstraintExpSeq;
    CosNotifyFilter::ConstraintInfoSeq *cis =
        new CosNotifyFilter::ConstraintInfoSeq;
    CORBA::Boolean has_event;
    CosNotification::StructuredEvent* event;
    CORBA::Double newbalance;
    CORBA::Object_var it;
    CosNotifyChannelAdmin::ProxyConsumer proxyConsumerVar
        // call the Global initializer


```

```

CBSeriesGlobal::Initialize();
orb = CBSeriesGlobal::orb();
// get the root Naming Context
rootNC = CBSeriesGlobal::nameService( );
// get the default event channel
it = rootNC->resolve_with_string(
    "cell/resources/notify-channels/cell-default");
ec = CosNotifyChannelAdmin::EventChannel::_narrow(it);
// get consumer admin
ca = ec->default_consumer_admin();
proxyConsumerVar = ca->obtain_notification_pull_supplier(
    CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);
PxyPullS = CosNotifyChannelAdmin::StructuredProxyPullSupplier::_narrow(
    proxyConsumerVar);
// get factory finder
it = rootNC->resolve_with_string(
    "host/resources/factory-finders/host-scope");
hostScopeFF = IExtendedLifeCycle::FactoryFinder::_narrow(it);
// get filter factory
it = hostScopeFF->find_factory_from_string(
    "INotifyFilterManagedClient::Filter.object interface");
ff = IManagedClient::FilterFactory::_narrow(it);
// create a filter
fi = ff->create_filter("IBM_NTF_CTG");
// define the constraint
(*c1).length(1);
(*c1)[0].constraint_expr = CORBA::string_alloc(100);
strcpy((*c1)[0].constraint_expr, "($accountNumber == '123456') and (
    $balance < 1000)");
// define the filter
cis = fi->add_constraints(*c1);
// add the filter to the Proxy
PxyPullS->add_filter(fi);
// connect the PullConsumer to the event channel
PxyPullS->connect_structured_pull_consumer(NULL);
cout << "About to pull events" << endl;
while (1)
{
    event = PxyPullS->try_pull_structured_event(has_event);
    if (has_event)
    {
        event->filterable_data[1].value >= newbalance;
        cout << "The Account Balance is: " << newbalance << endl;
    }
    delete (event);
}
}

```

Makefile for C++ client's code on Windows NT

 This makefile will compile and link the client applications on Windows NT. Make sure you set the ACCOUNTDIR below to the Object Builder Working directory.

Note: Both AccountC.dll and AccountS.dll need to be copied into the bin path.

```
ACCOUNTDIR = ..\Working\NT\PRODUCTION
```

```
LLIBS = \  
$(ACCOUNTDIR)\AccountC.lib \  
somasali.lib \  
sompngli.lib \  
sompncii.lib \  
somorori.lib  
  
.all: \  
.\PullConsumer.exe .\PushSupplier.exe  
  
.\PullConsumer.obj: \  
.\PullConsumer.cpp  
@echo " Compile "  
icc.exe /Fo".%\fF.obj" /Gm+ /Ti+ /I $(ACCOUNTDIR) /C .\PullConsumer.cpp  
  
.\PushSupplier.obj: \  
.\PushSupplier.cpp  
@echo " Compile "  
icc.exe /Fo".%\fF.obj" /Gm+ /Ti+ /I $(ACCOUNTDIR) /C .\PushSupplier.cpp  
  
.\PullConsumer.exe: .\PullConsumer.obj  
@echo " Link "  
icc.exe @<<  
/B" /de" /Gm+ /Ge+ /Gd+  
/FePullConsumer.exe  
.\PullConsumer.obj  
$(LLIBS)  
<<  
  
.\PushSupplier.exe: \  
.\PushSupplier.obj  
@echo " Link "  
icc.exe @<<  
/B" /de"  
/FePushSupplier.exe  
.\PushSupplier.obj  
$(LLIBS)  
<<  
  
clean:  
erase *.exe *.obj
```

Makefile for C++ client's code on AIX



This makefile will compile and link the client applications on AIX. Make sure you set the ACCOUNTDIR below to the Object Builder Working directory.

Note: Both libaccountC.so and libaccountS.so need to be copied into the \$HOME/lib directory from the Working/AIX directory.

```
# The actions included in this make file are:
# Compile and Link

# include make rules file
include /usr/lpp/CBToolkit/lib/samples.mk

ACCOUNTDIR      = ../Working/AIX
LOCAL_INCDIRS   = -I$(ACCOUNTDIR)
LOCAL_LIBPATH    = -L$(ACCOUNTDIR)

LIBS             = $(LOCAL_LIBPATH) $(LIBPATH)
INCS             = $(INCDIRS) $(LOCAL_INCDIRS)

#-----
# tools and build environment
# make sure CLASSPATH is defined in the environment variable
#-----

CC_FLAGS        = $(INCS) -D__IBMCPP__ -qlanglvl=ansi
LDFLAGS         = -brtl

CLEAN_FILES = PushSupplier.o PushSupplier PullConsumer.o PullConsumer
.SUFFIXES: .o .cpp

.cpp.o:
$(CCC) -c -o$@ $(CC_FLAGS) $<

all: PushSupplier PullConsumer

PushSupplier.o: PushSupplier.cpp
PullConsumer.o: PullConsumer.cpp

PushSupplier: PushSupplier.o
$(CCC) $(LDFLAGS) $(CONST_LD_FLAGS_DEBUG) -o$@ PushSupplier.o \
$(LIBS) -lsompmci -lsompmg1 -lsomos1 -laccountC -lsomoror

PullConsumer: PullConsumer.o
$(CCC) $(LDFLAGS) $(CONST_LD_FLAGS_DEBUG) -o$@ PullConsumer.o \
$(LIBS) -lsompmci -lsompmg1 -lsomos1 -laccountC -lsomoror

clean:
$(REMOVE) -f $(CLEAN_FILES)
```

Step 3: Running the sample

The first process in running the sample is to load and configure the application. The *WebSphere Application Server Enterprise Edition Component Broker Planning, Performance and Installation Guide* provides a sample (for Claim object) to do this. Follow the procedures to load and configure the AccountAppFam application.

The following instructions were used to load and configure the application in our test scenario. However, since these instructions can change always refer to the *WebSphere Application Server Enterprise Edition Component Broker Planning, Performance and Installation Guide* for the latest information.

Loading the application onto System Management

1. Start the System Manager User Interface.
2. From the System Manager User Interface menu, select **View > User Level > Expert**.
3. Expand Host Images, and select **myhost** (that is, your hostname).
4. Open the menu for **myhost**, and select **Load Application**.
5. Click **Browse**, and select **AccountAppFam.ddl**.

Configuring the application

1. **Configure the application with a management zone.**
 - a. **For accountS:**
 - 1) Expand Available Applications and select **accountS**.
 - 2) From the pop-up menu of the accountS application, select **Drag**.
 - 3) Expand Management Zones > **Sample Application Zone > Configurations**, and select **Sample Configuration**.
 - 4) From the pop-up menu of Sample Configuration, select **Add Application**.
 - b. **For iDefaultCellNotifyChannel:**
 - 1) Expand Available Applications and select **iDefaultCellNotifyChannel**.
 - 2) From the pop-up menu of the iDefaultCellNotifyChannel application, select **Drag**.
 - 3) Expand Management Zones > **Sample Application Zone > Configurations**, and select **Sample Configuration**.
 - 4) From the pop-up menu of Sample Configuration, select **Add Application**.
 - c. **For iNotificationService:**
 - 1) Expand Available Applications and select **iNotificationService**.
 - 2) From the pop-up menu of the iNotificationService application, select **Drag**.
 - 3) Expand Management Zones > **Sample Application Zone > Configurations**, and select **Sample Configuration**.
 - 4) From the pop-up menu of Sample Configuration, select **Add Application**.

Note: The `accountS`, `iDefaultCellNotifyChannel` and the `iNotificationService` applications were added to the Applications folder within the Configurations folder.

2. **Configure the server.**

- a. Expand Management Zones > **Sample Application Zone > Configurations**, and select **Sample Configuration**.
- b. From the pop-up menu of Sample Configuration, select **New > Server Group**. A dialog box is displayed.
- c. Type `AccountServerGroup` as the name for the server group.
- d. Click **OK**. The `AccountServerGroup` is displayed under Server Groups, which is under sample configurations.
- e. From the pop-up menu of `AccountServerGroup`, select **New > Server member of group**. A dialog box is displayed.
- f. Type `AccountServer` as the name for the server.
- g. Click **OK**. The `AccountServer` is displayed under `AccountServerGroup > servers` (member of the group).

3. **Associate the configured application with the server.**

a. **For `accountS`:**

- 1) Expand Management Zones > **Sample Application Zone > Configurations > Sample Configuration > Applications** and select `accountS`.
- 2) From the pop-up menu of `accountS`, select **Drag**.
- 3) Expand Management Zones > **Sample Application Zone > Configurations > Sample Configuration > Server Groups** and select `AccountServerGroup`.
- 4) From the pop-up menu of `AccountServerGroup`, select **Configure Application**.

b. **For `iDefaultCellNotifyChannel`:**

- 1) Expand Management Zones > **Sample Application Zone > Configurations > Sample Configuration > Applications** and select `iDefaultCellNotifyChannel`.
- 2) From the pop-up menu of `iDefaultCellNotifyChannel`, select **Drag**.
- 3) Expand Management Zones > **Sample Application Zone > Configurations > Sample Configuration > Server Groups** and select `AccountServerGroup`.
- 4) From the pop-up menu of `AccountServerGroup`, select **Configure Application**.

c. **For `iNotificationService`:**

- 1) Expand Management Zones > **Sample Application Zone > Configurations > Sample Configuration > Applications** and select `iNotificationService`.

- 2) From the pop-up menu of `iNotificationService`, select **Drag**.
- 3) Expand Management Zones > **Sample Application Zone** > **Configurations** > **Sample Configuration** > **Server Groups** and select **AccountServerGroup**.
- 4) From the pop-up menu of `AccountServerGroup`, select **Configure Application**.

Note: A Configured Applications folder is displayed under `AccountServer`. You can expand the folder to display the entries for `accountS`, `iDefaultCellNotifyChannel` and `iNotificationService`.

4. **Configure the server with the host.**
 - a. Expand Management Zones > **Sample Application Zone** > **Configurations** > **Sample Configuration** > **Server Groups** > **AccountServerGroup** > **Servers** (member of group) and select **AccountServer**.
 - b. From the pop-up menu of `AccountServer`, select **Drag**.
 - c. Under Hosts, select **myhost** for your current system.
 - d. From the pop-up menu of `myhost`, select **Configure Server (member of group)**.

Note: Under the `myhost` folder, there is now a folder called `Configured Servers` (members of group) that contains an entry for the `AccountServer` server.

5. **Activate the configuration.**
 - a. Expand Management Zones > **Sample Application Zone** > **Configurations** and select **Sample Configuration**.
 - b. From the pop-up menu of `Sample Configuration`, select **Activate**.

Running the application

Make sure the client code is built.

After activating the application, run the `PushSupplier` client and add some *balance* into the account (start with > \$1000). Then, start the `PullConsumer` client in another window. As soon as the *balance* is changed by the `PushSupplier` to be below \$1000 the following statement should appear on the `PullConsumer` client window followed by the amount remaining in the account. The Account Balance is:

1. **To run PushSupplier:**
 - a. From a command prompt, change directory to where the `PushSupplier` is stored.
 - b. Type `PushSupplier`

- c. The program will prompt you to add *balance* to the account with the following statement.
 - Enter the amount of Balance to be changed
 - d. Type 3000
 - e. Use another command prompt window to run the PullConsumer.
2. **To run PullConsumer:**
- a. From a command prompt, change the directory to where the PullConsumer is stored.
 - b. Type PullConsumer
 - c. Wait for the following output: About to pull events.
 - d. Go to the PushSupplier window.
3. **Remove money from the account.**
- a. In the PushSupplier window at the following prompt: Enter the amount of Balance to be changed
 - Type -1500
 - b. You should not see any event in the PullSupplier window since the amount is still > 1000.
 - c. Next Type -1000 at the prompt.
 - d. You should see the following statement on the PullConsumer window.
 - The Account Balance is: 500

User-defined event channel

The sample presented above uses the default event channel. If one needs to use a user-defined event channel, they need to create a visible event channel with a name and use it instead. For example if the name of the event channel is "newev1" then the path name to obtain it is cell/resources/notify-channels/newev1. Change the following code snippet in the sample above:

```
// obtain the default event channel
tmp = rootNC->resolve_with_string(
"cell/resources/notify-channels/cell-default");
```

Change to:

```
// obtain the user-defined event channel
tmp = rootNC->resolve_with_string(
"cell/resources/notify-channels/newev1");
```

Create a user-defined event channel: The following code can be used to create a new user-defined event channel (called newev1).

```
#include <INotifyChannelAdminManagedClient.hh>
#include <IExtendedLifeCycle.hh>
#include <CBSeriesGlobal.hh>
#include <time.h>
```

```

void main(int argc, char *argv[])
{
    CORBA::Object_var intermediateObject;
    IExtendedLifeCycle::FactoryFinder_var hostScopeFF;
    INotifyChannelAdminManagedClient::EventChannelFactory_var ecHome;
    CosTrading::PropertySeq qos;
    CORBA::Short Priority;
    time_t Time_out;
    ByteString_var key;
    CosNotifyChannelAdmin::ChannelID cid;
    CosNotifyChannelAdmin::EventChannel_var ec;
        // initialize the CBSeries environment
    CBSeriesGlobal::Initialize();
        // obtain the default factory finder with a host scope
    intermediateObject = CBSeriesGlobal::nameService()->
        resolve_with_string(
            "host/resources/factory-finders/host-scope");
        // narrow to a factory finder
    hostScopeFF = IExtendedLifeCycle::FactoryFinder::_narrow(
        intermediateObject);
        // find the event channel factory
    intermediateObject = hostScopeFF->find_factory_from_string(
        "INotifyChannelAdminManagedClient::EventChannel.object interface");
        // narrow to the event channel home
    ecHome =
        INotifyChannelAdminManagedClient::EventChannelFactory::_narrow(
            intermediateObject);
        // Create Quality of Services (QoS)
    qos.length(2);
    qos[0].name = CORBA::string_alloc(20);
    strcpy(qos[0].name, "Priority");           // Priority = 6
    Priority = 6;
    qos[0].value <=< Priority;

    qos[1].name = CORBA::string_alloc(20);
    strcpy(qos[1].name, "Timeout");           // timeout = two days
    Time_out = 2 * 24 * 60 * 60 * 100000000; // in 100 nanoseconds
    qos[1].value <=< Time_out;
        // use the factory in the event channel home
        // to create a new event channel
    ec = ecHome->createVisibleEventChannel(
        key, "newev1", 1, 1, 1, qos, NULL, cid);
}

```

Chapter 4. Externalization Service

The Externalization Service provides a mechanism by which objects are able to save and restore their state in a non-object form. This allows the object's state to exist independent of the existence of the object itself. The state can be maintained for an indefinite amount of time without regard to the continued existence of the original object or the ORB process in which it existed.

Typically, when the state of an object is saved, it is later restored into a different instance of an object of the same type, rather than into the same object instance. The new instance of the object is independent of the original object in regard to location, and may:

- exist in the same or a different process
- exist on the same or different physical machine
- exist on machines of the same or different hardware
- exist on machines of the same or different operation system
- be implemented in the same or a different programming language

The state of an externalized object is contained in a stream of bytes. This stream of bytes could be:

- contained in a memory buffer
- stored in a file
- transported in a network independent of an ORB

Relationship to OMG Externalization Service

The Externalization Service in Component Broker is a relatively small subset of the Externalization Service defined by OMG. OMG defined interfaces for externalization are grouped by module as follows:

CosExternalization

This module contains the client usage interfaces and consists of the Stream, StreamFactory and FileStreamFactory interfaces.

CosStream

This module contains the service construction interfaces and consists of the StreamIO, Streamable and StreamableFactory interfaces.

CosCompoundExternalization

This module contains the interfaces for handling a graph of related objects and contains the Node, Role, Relationship, PropagationCriteriaFactory interfaces.

Component Broker only makes use of the StreamIO and Streamable interfaces defined in the CosStream module. It does not make use of the CosStream::StreamableFactory interface or any of the CosExternalization or CosCompoundExternalization modules.

Use of externalization in Component Broker

Component Broker does not make use of the interfaces defined in the OMG CosExternalization module, that are the interfaces for client usage of Externalization. The implication is that Component Broker does not provide a model for clients to directly use externalization in a general manner. Component Broker does use and implement the CosStream::Streamable and CosStream::StreamIO interfaces which are part of what OMG refers to as the service construction interfaces. These are used internally by Component Broker in specific usage scenarios. How externalization is used is important for developers of managed objects to understand as this will affect some of the code that they are required to provide. The interfaces used by Component Broker have the following purposes:

Streamable

Objects that can be externalized inherit the CosStream::Streamable interface. They implement the `externalize_to_stream` and `internalize_from_stream` methods which are used to write and read the essential state of the object.

StreamIO

A StreamIO object is responsible for transforming data between the format understood by an object and a format which can exist outside of an object. A StreamIO is passed to a Streamable object on the `externalize_to_stream` and `internalize_from_stream` methods. The Streamable makes calls on this StreamIO to perform the externalization and internalization of individual elements of its state.

StreamIO objects are local only objects and they are not created by client code or by your managed object implementations. They are only created internally by Component Broker. The implication of this is that only Component Broker internals ever request a Streamable object to externalize or internalize itself. The CosStream::Streamable interface is introduced in two places by Component Broker.

1. It is inherited by the `IManagedClient::IManageable` interface, thus making all managed objects Streamable. This capability may be used by Component Broker when moving an instance of a managed object to another location.

2. It is inherited by `IManagedLocal::INonManageable` which means that Primary Key and Copy Helper objects are Streamable. Externalization's primary use in Component Broker is to support externalizing Primary Key and Copy Helper objects.

For Primary Key and Copy Helper objects, the externalization methods `externalize_to_stream` and `internalize_from_stream` provide the underlying support for the `INonManageable::toString` and `INonManageable::fromString` methods. The `ByteString` type used by these methods is actually the externalized form of the object state. Managed object developers should see the "MOFW Server Programming Model" chapter in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide* for more information on how these methods are used.

Chapter 5. Identity Service

Component Broker derives an object identity from relative information that positions the object within its Container, Server, Host, and ultimately Domain. This information can be used within the Component Broker Managed Object Framework to uniquely identify it from any other object in the distributed system. The `CosObjectIdentity::IdentifiableObject::is_identical()` method is implemented to use this information to precisely determine whether two objects are the same object.

If you want to compare whether two objects are the same object, you can use the `is_identical()` method on one of the objects, passing in the other object. This returns a boolean value indicating whether the two objects are the same object.

Comparing objects

Object systems, and particularly distributed object systems tend to obscure object identities. It is natural for object oriented designs to define objects which are in fact veneers to other objects. This is the case, for instance, where proxies are introduced to a client process, and the proxies “stand in” for the real object referred to by the client. The proxy appears to be the real object, but in fact the real object is hidden behind and is obscured by the proxy.

This condition makes it difficult to know whether two objects are in fact the same object. If your client program has two object references, you cannot simply compare the pointers to those references to determine whether they are referring to the same object. It is possible for two distinct references to refer to the same object, even though the reference objects themselves (the proxies) are two distinct objects. The references have different memory locations, and they have different pointers. For these reasons, you cannot reliably use the pointer values to determine whether the two objects are the same object.

Further, it is possible that both references contain different state even though they refer to the same object. Thus, you can’t reliably compare the values of the references to determine whether the two objects are the same object, either. To resolve this problem, Component Broker introduces an Identity Service. As part of the service, all managed objects support the `CosObjectIdentity::IdentifiableObject` interface.

This interface introduces two methods, `is_identical()` and `constant_random_id()`, which can be used to determine whether two objects are the same object. The `is_identical()` method is implemented on the managed object. It uses information intrinsic to the object to determine accurately whether the object is the same as the object to which it is being compared.

Compare two objects

This procedure is used to determine whether two objects are the same or different objects. This is useful for avoiding duplicate or circular references. You can only perform object comparisons on objects that support the `CosObjectIdentity::IdenticalObject` interface.

To perform object comparisons, invoke the `is_identical()` method. Assuming you have two object references, invoke the `is_identical()` method on one object, passing in the other object. This returns a boolean value indicating whether the two objects are the same or different.

The following example compares two objects:

```
// Declare two arbitrary business objects --
// this assumes both objects are derived from the CBC
// managed object framework.

SomeBusinessObject_var myB01, myB02;

// Create or obtain instances of these objects some how
...
// Compare the two objects.
if (myB01->is_identical(myB02))
    // Do something ...
```

Compare multiple objects

This procedure demonstrates how you can compare an object to a collection of objects. This is useful when you maintain a number of objects and want to compare them against another object to determine if the object is already a member of your collection. This procedure only works for objects that support the `CosObjectIdentity::IdentifiableObject` interface.

Perform an optimized object comparison using the following steps:

1. Get and cache the `constant_random_id` for the collection members. As each member is added to the collection, get the object's `constant_random_id` and cache in the collection.
2. Obtain the `constant_random_id` for the additional object. Obtain the `constant_random_id` for the object you want to compare.
3. Compare `constant_random_ids`. Compare the `constant_random_id` of the additional object with the cached `constant_random_id` for each member of the collection.

4. Invoke the `is_identical()` method on matched objects. If the `constant_random_id` values are not the same, then the objects are not the same. However, if the `constant_random_id` values are the same, then you must invoke the `is_identical()` method on the two objects to determine precisely whether they are the same.

The following example demonstrates comparing objects using `constant_random_ids`. This example presumes the existence of a sequence of objects, and their corresponding `constant_random_id`'s, as defined by the following IDL:

```
struct CollectionMember {
    CosObjectIdentity::ObjectIdentifier approximateIdentity;
    CORBA::Object theObject;
};
typedef sequence<CollectionMember> Collection;
```

The remainder of the example compares a passed object with all of the members in the `Collection`:

```
// Declare the collection (presuming it is set elsewhere), an
// index, a termination flag, and the constant_random_id of the
// passed object, and the passed object itself (presuming it is
// a managed object and is passed in from elsewhere).

Collection_var theCollection;
unsigned long index;
boolean matchingObjectFound = 0;
SomeBusinessObject_var myB01;
CosObjectIdentity::ObjectIdentifier myB01ApproximateIdentity;

// Get the constant_random_id for the passed object.

myB01ApproximateIdentity = myB01->constant_random_id();

// Iterate through the collection, testing each approximate
// identity, and invoking is_identical on those that appear
// to be the same to determine exactly whether they are the
// same object.

for (index=0;
     (matchingObjectFound == 0) &&
     (index < theCollection.length(); index++)
     {
     if (myB01ApproximateIdentity ==
         theCollection[index].approximateIdentity)
     {
     // constant_random_ids match, so test for actual identity
     if (myB01->is_identical
         (theCollection[index].theObject))
     {
     // A match was found, do something and terminate
     // further searching.
     ...
     }
     }
     }
```

```
        matchingObjectFound = 1;
    }; // endif
}; // endif
}; // endfor
```

Optimizations for object collections

If you implement a collection of objects, and want to compare whether some object passed in to you is already in your collection, you can use the `CosObjectIdentity::IdentifiableObject::is_identical()` operation to compare the passed object against each object in your collection. However, since the `is_identical()` operation is implemented on managed objects, this implies that you would be invoking the operation on the managed objects themselves.

If the objects in your collection are potentially located on other servers, then invoking the `is_identical()` method on each object can get expensive in terms of the system resources it consumes. The method request has to be communicated between the servers, potentially over the network, the object has to be reactivated, and the response returned. The expense associated with invoking the `is_identical()` method increases as more objects are added to the collection. Furthermore, in unique collections, the comparison is destined to return false in all cases, but, at most, one.

To improve the performance of performing object comparisons, the Identity Service introduces the `CosObjectIdentity::Identifiable::constant_random_id` attribute. This attribute is defined to return a constant value with the following properties:

- The attribute always returns the same constant value throughout the lifetime of the object.
- The attribute is a unsigned long hash-value that approximates the identity of the object.

In your collection, you can obtain the object's `constant_random_id` when each object is added to the collection caching the value within your collection. Later, when you want to compare two objects, you can obtain the passed object's `constant_random_id` and compare it to each of the `constant_random_ids` that you have already cached in your collection.

Since the values remain constant throughout time, you can be certain that if the two values are not the same, then the two object are not the same. However, if two values are the same, you can't be certain that the two objects are the same object. In this case, you can then invoke the `is_identical()` operation on just those few objects whose values are the same.

The `constant_random_id` value is a full unsigned long value with good distribution properties. For most collections, the `constant_random_id` values have a high probability of being unique. Thus, the set of objects on which you actually have to perform `is_identical` on is very small, perhaps just one.

Chapter 6. LifeCycle Service

A LifeCycle Service provides operations for creating, copying, moving, and deleting objects in a distributed environment. Because of the distributed nature of the environment, clients need to perform lifecycle operations on objects in various locations. Suppose client programs were required to understand the specific configuration of the distributed environment in order to determine where to create a new object. This would pose a serious problem, because development of client code and the configuration of the environment in which that code runs are separate tasks, normally done by different people at different points in time. In addition, the configuration of the environment is likely to change over time, and these changes should not necessitate changes in client code. The LifeCycle Service in Component Broker is designed to address these issues by providing a level of abstraction between the client program wishing to create an object and the determination of the location where that new object will exist. With Component Broker LifeCycle Service, the location of a new object can be:

- Controlled by the client's specific knowledge of the configuration
- Controlled by the client's knowledge of some logical aspects of the configuration
- Controlled by administrative policy defined outside of the client

The LifeCycle Service concerns the following types of objects and how they are related and used by clients and administrators:

Managed Objects

The objects being created, copied, moved and deleted

Factories

The objects used by clients to create managed objects

Factory Finders

The objects used by clients to locate factories capable of creating particular types of managed objects

Locations

The objects used by factory finders to define the location in which managed objects will be created by the factories it finds

Although concerned with all of these objects, the major focus of the Component Broker LifeCycle Service is factory finding through use of the factory finder and location objects. Specifically, how these objects can be used by client programs and administrators to abstract location concerns out of client programs. The interfaces for the LifeCycle Service are based on the

CosLifeCycle module defined by the OMG CORBA services LifeCycle Service Specification. The Component Broker implementation provides extensions to the OMG defined service.

This chapter is organized to take the reader progressively deeper into the details of the LifeCycle Service. There are the following major sections:

- Concepts of LifeCycle Service provides the high level abstract concepts of lifecycle and introduces details relevant to the Component Broker implementation of lifecycle.
- Minor interfaces provides information about OMG defined CosLifeCycle interfaces which are part of Component Broker but do not play an important role in the LifeCycle Service.
- Detailed view of location-based factory finding provides the details needed to fully understand the semantics of factory finding in Component Broker.
- Lifecycle interfaces and implementations provides a detailed look at the FactoryFinder and Location interfaces and the mechanisms for accessing and creating these objects, including code snippets.
- Lifecycle example is a full code example of the creation and use of LifeCycle objects.

Concepts of LifeCycle Service

This section presents the high level concepts associated with the LifeCycle Service. The concepts are introduced at a rather high, abstract level and then a flavor for how these concepts apply to Component Broker is given. To start with, the explanations are segmented by the major object types, specifically addressing the nature and characteristics of each object type and it's relationship to the other object types. The major object types presented are:

Managed Objects

See Concepts of managed objects.

Factories

See Concepts of factories.

Factory Finders

See Concepts of factory finders.

Locations

See Concepts of locations.

Then we proceed to look at some other concepts, including:

- Location-based factory finding
- Location object implementations
- Vocabulary of proximity
- FactoryFinders bound in the name space
- Application factories and specialized homes

- Managed objects and local only objects
- Creating and obtaining lifecycle objects

An understanding of these concepts will enable you to better understand and appreciate the specifics of the Component Broker LifeCycle Service which are presented in detail in subsequent sections.

Concepts of managed objects

Managed objects are the business objects which you implement that provide solutions to your business problems. Examples of these would be objects such as Policy, Claim and Customer which are presented in the “Personal Life Insurance Application Example” described in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*. These objects are the whole reasons for your use of Component Broker, and the Component Broker LifeCycle Service only has value to you as it addresses issues related to your use of these business objects.

Typically, your client applications that use these objects are interested in the business interface these objects provide to them. Applications are not generally interested in specifics about implementation classes nor specific location information regarding where the object exists in the distributed environment. However, when it is time for your client application to create a new business object, implementation and location information must be known. Component Broker, with the assistance of the LifeCycle Service, lets you create business objects anywhere in the distributed environment without being encumbered by the complexity of the distributed system and the details of the implementation of the business objects. Let’s examine why isolating the client applications from these considerations is important.

Although you know the interface of your business object when you create your application, that interface may be different when you create an instance of that object in a production system. The interface may have been specialized and the implementation changed to add new behaviors.

The preceding may be even more true in a distributed system where the business object is used and shared among several different applications, and where each application may apply its own set of requirements to the business object. This is certainly true in the Component Broker programming model where business objects become managed objects between the time the business objects were originally developed and when they’re finally deployed in a production system.

The implementation of a business object is also likely to change over time to consider additional business requirements in the form of added behavior, to address new business conditions, priorities, policies, or constraints, or to fix bugs in the original object implementation.

Different enterprises have different needs as to where their business objects are located. In some institutions, life insurance policy objects are created at the insurance agent's branch office. In other institutions, those life insurance policy objects would have been created at the central office. You may not know exactly where (geographically) objects should be created in the distributed system. An enterprise may require that different kinds of objects should be created in different places.

As the enterprise grows it may add new servers to accommodate that growth. In doing so, it may need to redistribute the location of its business objects to maintain some balance over its computational resources. Although life insurance policy objects continue to be created at the central office, they may actually be created on a different server at that office.

As you can see, it would be very difficult to maintain a client application if all of these factors necessitated changes in that application.

To summarize the key points about managed objects in Component Broker

- Managed objects (business objects) provide solutions to business problems.
- An application which creates a managed object is principally interested in the interface that addresses the business need of the application.
- An instance of the managed object may have additional and extended interfaces, or any combination of the preceding that are not known by the client application.
- The implementation details of the managed object are not known by the client application.
- Location in the distributed system is not known by the client application.
- Interface, implementation and location details are likely to change over time.
- Client applications need to be isolated from these characteristics of managed objects to avoid consistently needing to be updated.

The Component Broker lifecycle model separates the concern for when to create a given kind of object from where and of what type (specific type and implementation) to create. It does so through the introduction of *factories*, *factory finders*, and *locations*. While the application is responsible for defining the kind of object it wants created and when, the factory object is responsible for encoding the specific type and implementation class for the kind of object or objects it supports. It is also responsible for encoding the specific details for how to create an object of that type. The location object is responsible for encoding a scope of location, which is something like a geographic area representing where objects should be created. Typically this is supplied or configured by the enterprise. Factory finders contain a location object which define the scope of location to be considered by that factory finder. The factory finder is responsible for finding a factory that supports creating the kind of specified object within that scope of location. Thus the factory finder,

in collaboration with a location object, is responsible for establishing where to create the object. The following sections look at these concepts in more detail.

Concepts of factories

The term *factory* is used loosely to refer to any object that is responsible for creating other objects. For example, a family may decide they need a life insurance policy. They would contact their insurance agent, who would then create their policy. From the perspective of this family, the agent can be thought of as a factory, or perhaps more accurately, a factory representative.

From the perspective of the insurance agent, creating the insurance policy for the family means filling out paperwork (to open the customer account and policy, and to declare the beneficiary), and adding the family to a list of clients. The insurance agent's insurance application form is the agent's factory for the new policy.

A data entry operator might then enter the information from the form into an application program. From the perspective of the insurance application program, creating the insurance policy means creating a PolicyHolder object, Beneficiary object, and Policy object, and adding the PolicyHolder as a Customer to the Agent object. The insurance application needs factories for each of the new objects that it creates.

Because PolicyHolder, Beneficiary, and Policy are all managed objects they each have, by definition of the Component Broker programming model, a corresponding home. Homes are in fact managed object factories, responsible for creation of a single managed object type and implementation in a particular location. This is the most rudimentary form of a factory, being directly responsible for creating the low level implementation instances that make up the managed object and allocating system resources required by the managed object, such as the memory the object occupies and its persistent storage. It defines what constructors or initialization methods to invoke and in what order, as well as any other activities that need to be performed for the enterprise or in the distributed system for that kind of object.

A managed object may be dependent upon the existence of another object or objects. When it is, it may need these other objects created at the same time it is created. To assure the consistency and integrity of the system, all managed objects must be created by a corresponding managed object factory, their home. Consequently, when one managed object depends on another managed object being created at the same time, then the subordinate object should also be created through its corresponding managed object factory. One way to accomplish this is for the superior object to interact with the subordinate object's factory itself, perhaps within superior object's initialization method. Another approach would be for the superior object's factory to interact with

the subordinate objects factory to create the subordinate object. In Component Broker, this might be accomplished through the implementation of a specialized home.

Factories other than managed object factories can exist also. When the life insurance application creates the PolicyHolder, Beneficiary and Policy objects, it is in effect acting as a factory. It may collect all of this logic into an object of its own that understands the mechanics of creating managed objects (through their corresponding homes), thus becoming what can be referred to as an application factory. It is common for application factories to deal with the creation of several interrelated objects as a single operation, thus abstracting object creation to be viewed more as a business process.

To summarize the key points about factories in Component Broker

- All managed objects must have a managed object factory (a home)
- A home creates managed objects of a single type and implementation in a very specific location
- A home isolates the client from needing to know specifics about the implementation of the managed object
- A managed object which has dependencies on the existence of other objects may:
 - Interact directly with the home of the other object itself
 - Have a specialized home which interacts with the home of the other object
 - Be created by an application factory which interacts with the homes of both objects

Concepts of factory finders

As we have seen, managed objects are created by managed object factories. When a client (client program, an application factory or a business object) needs to create a managed object, it must locate the managed object factory (the home) which it will use to create the managed object. A client interacts with a factory finder in order to locate an appropriate managed object factory. Locating the appropriate managed object factory involves two considerations:

What What kind of managed object is created by the factory

Where Where should the factory create the managed object

The answer to “what” must be provided by the client of the factory finder. The client is required to provide the factory finder with the type of managed object the requested factory will create. The client does not need to know the specific implementation of that type of managed object. In fact, if managed object factories exist for more than one implementation of that managed object type, either of the factories can be returned. However, if the underlying implementation of the managed object is important to the client, he can

provide additional information to the factory finder that will limit the factories returned to those creating managed objects of the requested type and implementation.

The answer to “where” is not provided by the client, but is specified through how the factory finder is configured. Each factory finder is configured with a location object which defines what parts of the distributed environment will be considered when looking for a factory that satisfies the “what”. For example, one factory finder may only look for factories in server process A on host X whereas another factory finder may look for factories in all server processes in host X, Y and Z. Truly understanding factory finders requires an understanding of the location objects which are used to define the “where”. This will be covered in Concepts of locations.

Within a distributed environment, there may be many managed object factories which create objects of a particular type. Clients can ask a factory finder to return one factory or all factories which satisfy the request. The most common usage of factory finders is to ask for a single factory. However, there may be some situations where a client wants to examine all the managed object factories for a particular object type, and then select for himself an appropriate one to use from the set of factories returned.

The key points to remember about factory finders are:

- The client tells the factory finder “what” type of objects the returned factories should create.
- The factory finder determines “where” the returned factories create the objects.

Because the factory finder determines where the returned factories create objects, choice of which factory finder a client uses should be a consideration if the location where the managed object is to be created is important.

Concepts of locations

Component Broker strives to make managed objects local or remote transparent. This makes it possible for you to program your application without regard to whether the objects you are using are local to your application or remote in another process, or even in another host somewhere across the network. The proximity of the objects you are using only becomes evident when you consider side effects, such as the latency associated with invoking requests on the object.

However, there are times when location does matter, most notably when the object is being created. For a variety of reasons, including performance in the overall system, enterprises want certain kinds of objects created in certain places. Since managed objects do get created by their home in a specific location (for example, in a particular container in a particular server process

in a specific host machine), control of where a managed object gets created is possible through proper selection of a home. However, container, server and host is most often not the terms in which an enterprise wants to talk about location and proximity of objects. Considering this, it is desirable to retain a degree of ambiguity in the way that proximity is communicated, while at the same time being able to translate that into the precise information needed to determine exact location, such as a particular server process on a particular host. To enable this, Component Broker introduces the concept of a location object. A location object is intended to be an abstraction for proximity. As an abstraction, location objects help in establishing a vocabulary with which proximity can be communicated, and yet they encapsulate the precision that is needed to make concrete decisions in the distributed computing system. The following categories represent possible scopes of location and types of proximity, or any combination of the preceding, that could be introduced into the enterprise's vocabulary through the use of location objects.

Infrastructural

homes, containers, and servers

Topological

hosts, workgroups, and cells

Physical

rooms, floors, buildings, sites

Political

departments, projects, companies, governments

Geographical

cities, time zones, countries, continents

Proximal

here, near, far

Temporal

now, soon, later

Compound

combining any of the above

Compound-conditional

representing different scopes based on some condition

Compound-temporal-conditional

representing different scopes at different times

Component Broker location objects specifically identify "infrastructural" (home, container, server) and "topological" (host, workgroup, cell) information to specify location within the distributed environment. It is this very specific information that location objects provide to factory finders during a search for a factory. The other forms of proximity in the above list are realized through

encapsulating the appropriate infrastructural and topological information in a location object and then treating that location object as an abstract representation of an alternate type of location. This idea will be expanded upon in Vocabulary of proximity.

The key points to remember about location objects are:

- They deal with specific location information, such as hosts and servers.
- They are used by factory finders to define where to look for a factory.
- They provide an abstraction by which location can be communicated and discussed in terms which are more likely to align with the enterprise's needs and application concerns about location.

Location-based factory finding

Now that we have examined the concepts of managed objects, factories, factory finders and location objects it is time to take a more in depth look at how factory finders and location objects work together to find appropriate factories for applications to use to create managed objects. To understand this, we need to look at:

- The infrastructural and topological elements provided by a location object
- The interaction between factory finders and location objects
- The use of the lifecycle repository by factory finders

Infrastructural and topological elements of location objects

Location objects encapsulate very specific information about location, which is referred to as a scope of location. The scope of location provides topological scope boundaries and infrastructural scope boundaries. This information is contained in a Scope structure with six elements, three elements for each of the topological and infrastructural scope boundaries, as follows:

- Topological scope boundaries (cell, workgroup and host) relate to the configuration of hosts within your distributed environment. Hosts are addressable machines and cell and workgroup are logical groupings of these machines, as have been defined in your Component Broker System Manager configuration.
- Infrastructural scope boundaries (server, container, home) relate to the deployment of managed object applications within a host machine. The container and home are defined as part of your application developed with Component Broker Object Builder and the server is where you have installed that application with Component Broker System Manager.

Topological scope boundary elements can be specified by name, by reference to a local default or be ignored. Infrastructural scope boundary elements can be specified by name or specified to include all possible instances. By combining values for the six elements into a Scope structure, a scope of

location can be defined which is very broad, very narrow or anywhere in between. The use of these scope boundaries will be explained in “Defining scope of location” on page 128.

Interaction between FactoryFinder and location objects

The Location interface is defined to return a sequence of Scope structures, each of which contain the six scope of location boundary values (cell, workgroup, host, server, container, home). Factory finder objects are configured to contain a Location object which defines the scope of location boundaries for that factory finder. The FactoryFinder is defined to look for factories by taking each Scope structure in the sequence and searching for factories registered in the factory repository according to the values of these six items. Since a sequence has order, the order in which these Scope structures appear in the sequence is the order in which the factory finder will proceed to search for factories. The Location interface has a `get_scopes` method which returns this sequence. Therefore, the actual logic in the FactoryFinder is to do a `get_scopes` on it's location object and then proceed to use that sequence of scopes to do the factory finding, a scope at a time.

How a Location object determines what the sequence of scopes is that it returns is up to the implementation of that particular location object. The Location interface itself is abstract. In Component Broker we provide two specific implementations of this interface, `SingleLocation` and `OrderedLocation`, which will be explained in detail in “Lifecycle interfaces and implementations” on page 153.

The lifecycle repository and factory finding

To assist in finding factories, the Component Broker Lifecycle Service maintains a factory repository. Managed object factories (homes) are automatically registered in the factory repository when they're created, The factory repository is partitioned over all three portions (cell, workgroup and host branches) of the System name space. This partitioning reflects the topological elements within a scope of location.

When homes register with the Lifecycle Repository, they provide information about themselves and the objects they create, including server name, container name and home name that will contain the managed objects they create. This information is retained in the repository and reflects the infrastructural elements within a scope of location. The home also provides information which is used to control within which of the topological branches of the tree the home's information will be maintained.

The client of the factory finder provides a factory key (see “Factory keys” on page 133) with information about the type of object it wants the factory to

create. The scope information obtained from the location object controls where in the lifecycle repository the factory finder will look for the factory. For each individual Scope structure returned by the location object, the factory finder will look in one branch of the lifecycle repository, based on the values for the topological elements. Within that branch of the repository, the information provided by the client and the infrastructural elements of the scope of location control what the factory finder searches for.

Within each of the topological branches of the lifecycle repository, there are two sub branches. The first, called the FactoryBranch, is only used for searches where the scope of location has indicated that infrastructural elements are not limited (that is, all servers, containers and homes are to be considered). The second, called the ServerBranch, is only used when the infrastructural elements indicate server by name (and possibly container and home name), indicating they are to be specifically considered in the search. Homes will be registered in both of these branches. This separation is partly for efficiency reasons, but as we will see in the Lifecycle repository structure and Factory finding specifics sections, this does affect under what conditions a home will be found.

Summary of location-based factory finding

By using the infrastructural and topological location scope information provided by the Location object, the factory finder limits where it searches for factories, and by controlling where they are registered, homes and servers control their visibility to the factory finder. If you consider your entire Component Broker distributed network as the universe within which factories can be found, the location object defines a scope of location which is a subset of that universe. The subset it defines can be as large as the entire cell, as small as identifying a specific home, or anywhere in between. Therefore, you can think about factory finding as an operation looking for a factory which satisfies what the client has specified in the factory key, where that factory exists in the subset of the Component Broker distributed network defined by the scope of location specified by the FactoryFinders contained Location object.

Location object implementations

So far we have only looked at location objects in terms of the abstract Location interface. We know that objects supporting the Location interface return a sequence of scope structures. How a Location object determines what the sequence of scopes is that it returns it up to the implementation of that particular Location object. Because the Location interface is abstract, there is no implementation of an object that is just a Location. The approach it is define a new interface which inherits from Location and implement that interface to have some mechanism for determining the sequence of scopes it

will return for the `get_scopes` method. In Component Broker we provide two such object implementations, `SingleLocation` and `OrderedLocation`.

The `SingleLocation` implementation is defined to be configured with specific values for the six items of a single scope structure. As such, when a `get_scopes` is called on it, it simply returns a sequence of scopes with just one element, the scope structure containing the six values it is configured with.

The `OrderedLocation` implementation is defined to be configured with a sequence of `Location` objects. What the `OrderedLocation` does for `get_scopes` is call `get_scopes` on each of the `Location` objects in it's sequence of `Locations` and builds a sequence of scopes that is essentially the concatenation of the sequence of scopes returned by each of the other `Location` objects. Since an `OrderedLocation` is only dependent upon it's contained objects being of type `Location`, it can contain `SingleLocations`, other `OrderedLocations` or other implementations of the `Location` interface.

In addition to these two implementations, a user of Component Broker can take the `Location` interface, inherit it into some other interface (for example, `MyLocation`) and then implement `MyLocation` to return something for `get_scopes`, based on some useful algorithm. A `MyLocation` object can now be used by a `FactoryFinder` or be contained within an `OrderedLocation`.

Vocabulary of proximity

In the Concepts of locations section, we introduced the idea that a location object is an abstraction which provides a way to establish a vocabulary by which proximity can be communicated. We have seen how the infrastructural and topological categories are directly specified by a location object and how they are used within Component Broker to assist in the factory finding process. Several different categories of proximity were also defined, such as physical, political and geographical.

You can use the Component Broker supplied implementations to create instances of location objects which satisfy some of these other categories of proximity within your environment. For example, suppose you wanted to have a location object which represented all of the hosts within a department (political proximity). This can be done by having a single object for each host within the department and then have an ordered `Location` object which contained all of these single location objects. You can now refer to this ordered location as being the department location and could use it whenever you wanted to find a factory within the departments computing resources. As the mix of hosts change within the department, the ordered location can be updated to reflect the changes. This same approach could be used for representing physical proximity (all the hosts in the 3rd floor lab) and geographical proximity (all the hosts in Texas).

There are times when you might want to create your own implementation of a location object. Suppose you wanted a compound-conditional location object which found factories within the departmental resources of the user for whom the request is being made. You could start by having several of the departmental locations described above. Then provide an implementation of the location interface which contained these departmental location objects along with a table relating user's principal IDs to the department they were in. Your implementation could then determine which departmental location to use, based on the principal ID contained within the context of the current request.

In the Concepts of locations section it was also stated that a proximal proximity could represent things like here, near and far. These things have different meanings in different contexts. For example, suppose I wanted two objects to be "near" each other. This might mean they reside in the same server, or that they reside in servers which are on the same host, or that they reside in servers on hosts attached to the same LAN, and so on. An interesting use of locations objects would be to enable an object to be created "near" another object, where near is interpreted by the creator of the second object. A possible implementation of this would be to have managed objects know about location objects which represented "nearness" to themselves (for example, the location object scoped to the server they reside in, the location object scoped to the host they reside in and a location object scoped to all the hosts in the LAN they reside in). You can then imagine the ability to have an operation on the managed object such as `get_location` which took a parameter of "server", "host" or "LAN". I can now get an appropriate location object which I can use with a factory finder to enable me to get a factory which can create a second object "near" the first object, where I define if "nearness" is the same server, same host or same LAN.

FactoryFinders bound in the name space

Although you can imagine various uses for location objects, the primary use of them is to enable factory finding. As we discussed earlier, a client asks a factory finder for a factory which creates a certain type of object, but where in the distributed environment the factory finder looks for that factory is determined by the location object the factory finder is configured with. Therefore, it is often very important for a client creating an object to have a way to get a reference to a specific factory finder. The most common way for this to be done is to have factory finders bound into the System name space.

We have seen how location objects enable us to introduce the vocabulary of proximity. It is therefore reasonable to bind a factory finder into the name space using a name that is an indicator of the type of proximity it represents.

For example, you could bind factory finders configured with the departmental locations described previously into `/cell/resources/locations/cell/resources/factory-finders` with names like `accounting-dept`, `purchasing-dept`, and so on. You can now make statements about creating an object within the accounting department and have a specific programming mechanism to find the appropriate factory (that is, code your application to use the factory finder bound at `/cell/resources/factory-finders/accounting-dept`). In addition, when new hosts are added to the accounting department, the single-location objects defining those hosts can be added to the departmental ordered location. The result is that no changes need to be made to any code to accommodate the configuration changes within the department, yet all the factories on those new hosts are available for use by your applications.

Component Broker provides a variety of default location objects and default factory finder objects which are directly associated with the topological and infrastructural views of proximity. Both the default location and factory finder objects are bound into the system name space. (See Default lifecycle objects for a complete list of these objects and where they are bound). Applications will often use these factory finders directly rather than creating their own. Also, the default locations can be used as building blocks for other locations. For example, if you needed to create an accounting department factory finder, that can be done by your creating the factory finder and ordered location representing the accounting department, but initializing the ordered location with the default host scoped single location objects for the hosts in the accounting department. The binding of a factory finder into the name space enables a separation of development concerns from configuration concerns. An example of this might be a software vendor who is developing a Component Broker application which will be used by several different companies, all with different distributed system configurations. A way to isolate the application from the configuration concerns is through an appropriately defined factory finder. The application might be coded to always use a factory finder obtained from the name space at `/host/resources/factory-finders/xyzApplication`. The software vendor does not supply this factory finder, but instead makes its existence a responsibility of each company installing the application. Each company configures this factory finder to reflect their environment as it should be seen by the `xyzApplication`. With this approach, the software vendors application can be isolated from the configuration differences of the various environments.

Application factories and specialized homes

In the Concepts of factories section, we discussed managed object factories (homes). Also introduced was the idea of using specialized homes or application factories when a managed object had a dependency on another object's existence. In actuality, specialized homes and application factories

could be introduced for a variety of reasons (see the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide* for information about specialized homes). Whatever the reason for introducing a specialized home or application factory, you need to understand and take into consideration differences in how they would be handled by your code.

Any application can introduce its own application factories for its own purposes. As an example, let's look at the case where a Policy object must always have a corresponding Beneficiary object. This can be done with an application factory that finds a Policy home, finds a Beneficiary home, creates both objects and establishes their relationship to each other. It would be appropriate for you to introduce this as a managed object and to bind it in the system name space under your own application naming context (for example, bind it at `/host/applications/myApplication/myApplicationFactory`). As such you can create one instance of the factory and continue to use that in each invocation of your application. Notice that this approach does not make use of factory finding by your application, but instead you directly do a naming resolve to access the application factory. Component Broker does not yet provide any support or assistance for distinctly managing application factories with the LifeCycle Service.

Alternately, you could use a specialized home and embed the logic for creating the Beneficiary as a side-effect of creating the Policy object. Doing so requires that you specialize the home for the Policy object to find the Beneficiary home, create the Beneficiary object and establish its relationship with the Policy object. A specialized home is distinguished from an application factory only in that a customized home continues to benefit from the distinct support provided to homes by Component Broker. In particular, a factory finder can be used to find a specialized home, but cannot be used to find an application factory.

Managed objects and local only objects

Component Broker enables development, deployment and use of managed objects. However, Component Broker actually allows for two fundamental kinds of objects, managed objects and local only (non-managed) objects. Generally, the local only objects have some specialized use or purpose related to the enabling of managed objects (for example, PrimaryKey objects). For complete information on managed objects and local only objects, see the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

There are differences between these kinds of objects in terms of their lifecycle (creation, existence and destruction) and therefore they are worth comparing here. More importantly, however, is the fact that the LifeCycle Service provides both managed and local only versions of FactoryFinder,

SingleLocation and OrderedLocation objects. Because of this, the nature of the differences between these kinds of objects is important to understanding details that will be presented in later sections.

We have already covered most aspects of the creation of managed objects. They are created by managed object factories, called homes. The home is responsible for allocating system resources needed by the managed object. Typically this includes the memory it occupies when active, and any storage used to retain its state persistently. The default implementation of home provided by Component Broker can be specialized and extended to perform any other functions that need to occur when the object is created, such as registering the object with other objects and resource managers as needed by the system. Alternately, this function can be embedded in the initialization of the object, or it can be performed by an application factory. Introducing an application factory provides more relevance to applications that use it to perform a business function. However, it still ultimately results in invoking one of the creation methods on a home for creating the actual managed object instance.

local only objects are created directly from their native language class. Most often this is done using the mechanisms provided with your programming language for creating native language objects, for instance using the new operator in C++. However, to preserve some language neutrality, local only objects provided by Component Broker support a more general approach. Normally this involves using the static class function `_create` for the class of the local only object. There may in fact be multiple `_create` methods which take different combinations of initialization parameters, similar in concept to overloaded C++ constructors. The specifics for creating instances of local only objects supplied by Component Broker is detailed in the documentation for each of those types of objects.

Similar differences exist for the destruction of objects. Managed objects all support the “remove” operation. When a managed object is asked to remove itself, it cooperates with its home to ensure that all resources allocated to it during creation are appropriately released. For local only objects, the native language delete method is used.

Finally, the existence of these two kinds of objects varies. Managed objects can only exist in Component Broker server processes, are accessible for use by objects in other processes and generally do not cease to exist when the server process is stopped and restarted. On the other hand, local only objects can exist in either Component Broker server or client processes, are only available to other objects in that process and will not continue to exist upon termination of the process in which they were created.

From a LifeCycle Service perspective, the service itself only relates to providing support of lifecycle concerns of managed objects. However, for factory finding purposes, local only versions of factory finder objects and location objects are provided. These can be useful when there is no existing managed factory finder with the scope of location you need, and you happen to know the specifics of what the scope of location should be. An example of this need would be within the Component Broker bootstrapping mechanism, where the home of factory finders must be found in order to create the first managed factory finder. Details of the use of local only factory finder and location objects is provided in “Local only creation mechanisms” on page 163.

Creating and obtaining lifecycle objects

When your application has need to find a factory, you need to obtain a reference to a factory finder. There are generally two ways that this is done:

- Create a local only factory finder.
- Obtain a managed factory finder which is bound in the name space.

When getting a managed factory finder by doing a resolve to the name space, there is always the question of how did that factory finder get created and placed in the name space. There are basically three ways in which this happens:

- It is a default factory finder created by the Component Broker LifeCycle Service.
- It was created by an application (your application or some other application on which your application depends).
- It was configured using the System Manager (by a system administrator).

Factory finders have the capability to bind themselves into the name space when they are created, so the binding itself is normally a part of the creation process.

When you are running code during development, you will most likely make use of the default factory finders for most things. However, when an application is deployed, you are more likely to need a factory finder whose scope of location is different than that provided by the default factory finders. There are generally two approaches to the definition of these factory finders:

- Your application defines the need for one or more factory finders which reflect certain things about the configuration relevant to your application. You either provide a setup application which creates them or you provide instructions to a system administrator on how they should be created using the System Manager when he is installing your application, and where they should be bound into the name space.
- The enterprise within which your application is being installed has policies regarding the creation of objects and the factory finders that should be used. In this case, you were probably given the information as to which factory finders your application should use and where they will be bound

in the name space. These factory finders will already exist prior to the installation of your application. The enterprise may have created these with an application program or configured them with the System Manager.

These ideas also apply to location objects. However, there is generally not a need for the location objects to be bound into the name space and it is only under certain limited application scenarios that you would need to directly access a location object from the name space. However, when a setup application creates (or system administrator configures) a factory finder, they generally also have to create appropriate location objects for use by the factory finder.

The “Lifecycle interfaces and implementations” on page 153 section provides details about creating local only and managed lifecycle objects.

Minor interfaces

The interfaces documented in this section are, from an interface definition standpoint, part of the Lifecycle Service. However, implementation of these interfaces occurs outside of the Lifecycle Service. The documentation here is intended to provide some background information and identify where these interfaces are implemented within Component Broker.

LifeCycleObject interface on managed objects

The OMG CosLifecycle module introduces an interface called LifeCycleObject which provides the operations needed to move, copy and delete an object. It introduces the operations:

- copy
- move
- remove

All managed objects in Component Broker are lifecycle objects. The CosLifecycle::LifeCycleObject interface is inherited by the programming model interface IManagedClient::IManageable, which is the base interface for all managed objects.

Other than definition of the interface, the Component Broker Lifecycle Service does not provide any implementation support for this interface. Rather, implementations are provided by the application adaptors which support the infrastructure in which managed objects are created and exist.

In general, the application adaptors do not support implementations of the two operations, move and copy. The remove operation is supported and is the way defined by the programming model to delete a managed object.

A definition of this interface and its operations can be found in the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* in the CosLifeCycle module. Discussion of usage can be found in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

GenericFactory interface on homes

The OMG CosLifeCycle module introduces a interface called GenericFactory which provides operations used to create objects. It introduces the operations:

- supports
- create_object

All homes in Component Broker are generic factory objects. The CosLifeCycle::GenericFactory interface is inherited by the programming model interface IManagedClient::IHome, which is the base interface for all homes. However, this interface is not the interface typically used for creating an object on a home. The IHome interface introduces the operation createFromPrimaryKeyString which is the preferred creation operation to create_object.

The create_object interface as defined by OMG provides a mechanism for passing input parameters to the factory which are name value pairs, where the value can be of any type. However, the OMG does not define what any of the name value pairs should be. Therefore this is a standard method that can be completely different in usage based on the name value pairs a particular generic factory chooses to accept as input.

Other than definition of the interface, the Component Broker LifeCycle Service does not provide any implementation support for this interface that is used by all homes. Rather, implementations are provided by the application adaptor implementations of the IHome interface, which also define the appropriate name value pairs that they accept.

Any implementation of a specialized home may provide implementation support for additional name value pairs as input parameters. LifeCycle does introduce some specialized home implementations for its own objects (FactoryFinder, SingleLocation, OrderedLocation) which define additional parameters that can be passed on create_object.

A definition of this interface and its operations can be found in the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* in the CosLifeCycle module. Implementations of homes described in the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* should provide the information about the name value pairs they support.

Detailed view of location-based factory finding

Concepts of LifeCycle Service has provided much of the background information needed to understand the basics of factory finding based on scopes of location defined in location objects. Assuming that the reader has read that section, we now go deeper and deal with many of the specifics in the Component Broker implementation. This knowledge is essential to be able to effectively use the LifeCycle Service in Component Broker.

The following section discusses these topics:

- Defining scope of location
- Lifecycle repository structure
- Factory keys
- Factory finding specifics
- Default lifecycle objects
- Tips for using the default lifecycle objects

Defining scope of location

Starting with a quick review, location objects define a scope of location within the distributed environment. This scope of location is more specifically defined by a sequence of Scope structures which the location object returns. This section takes a detailed look at the specific values that can be used in the six elements of a Scope structure.

A Scope structure contains six state variables, three of which represent infrastructure scope boundaries, *home*, *container*, and *server* and three of which represent topology scope boundaries, *host*, *workgroup*, and *cell*. The rules and meaning for the infrastructure scope boundaries are different than those for the topological scope boundaries, as described in the following sections.

Infrastructure scope boundaries

Any of the infrastructure boundary-elements can be set with either a name or the keyword **ANY*, with the following rule: Once **ANY* has been specified for a higher level scope boundary, each of the lower level boundary elements must be set to **ANY* as well. The following table defines valid values and the resulting scope of location that can be specified for the infrastructure boundary elements of a Scope structure:

Table 9. Infrastructure boundary-element values

Server	Container	Home	Resulting Infrastructure Scope
name	name	name	A specific home creating objects in a specific container in a specific server or server -group

Table 9. Infrastructure boundary-element values (continued)

Server	Container	Home	Resulting Infrastructure Scope
name	name	*ANY	Any home creating objects in a specific container in a specific server or server-group
name	*ANY	*ANY	Any home creating objects in any container in a specific server or server-group
*ANY	*ANY	*ANY	Any home creating objects in any container in any server or server-group

The name specified for home and container is the name given to these when they were defined using the Component Broker Object Builder. The name specified for server is the server or server group name as defined for it when it was configured with the Component Broker System Manager.

Although it is possible to have a location object which specifies a specific container or home by name, in actual practice this is unusual. In most instances, these two boundary elements are specified as *ANY.

Topology scope boundaries

Any of the topology boundary elements can be set with either a name or one of the keywords *LOCAL, or *IGNORE, with the following rules:

- Once *LOCAL has been specified for a lower-level scope boundary, each of the higher-level boundary elements must be set to *LOCAL as well.
- Because Component Broker currently only supports a single cell, the cell boundary element must always be set to *LOCAL.

The topology boundary-element values are used to form a name path. The resulting name path is used to resolve to a portion of the system name space representing a different portion of the distributed system topology. In this way, when used with factory finding, the location object can be used to narrow or expand the scope over which factories are found; finding only factories which have been registered with the corresponding level of visibility.

The following table defines the valid values and resulting scope of location that can be specified for the topology boundary-elements of a location object:

Table 10. Topology boundary-element values

Cell	Workgroup	Host	Resulting Topology Scope (As a Name Path)
*LOCAL	name	name	/cell/workgroups/<workgroup-name>/hosts/<host-name>
*LOCAL	name	*IGNORE	/cell/workgroups/<workgroup-name>

Table 10. Topology boundary-element values (continued)

Cell	Workgroup	Host	Resulting Topology Scope (As a Name Path)
*LOCAL	*IGNORE	name	/cell/hosts/<host-name>
*LOCAL	*IGNORE	*IGNORE	/cell
*LOCAL	*LOCAL	name	/workgroup/hosts/<host-name>
*LOCAL	*LOCAL	*IGNORE	/workgroup
*LOCAL	*LOCAL	*LOCAL	/host

The name specified in each case in the above table is the name for the corresponding element in the topology as defined for it in the system name space. The resulting name paths represent where resources are located in the system name space, and by extension implying where they are located in the system topology.

Lifecycle repository structure

In review, we have seen that factory finding utilizes a lifecycle repository that is constructed in the System name space. This repository has three major branches, one for each of the topological elements of location (cell, workgroup and host). Each major branch has two sub branches, a factory branch and a server branch, within which information about registered homes is maintained. Both homes and servers have some control over where in the lifecycle repository information about them appears.

For a complete understanding of the factory finding process, it becomes important to know specifics about how homes are registered, where they appear in the repository and how that affects the conditions under which they will be found. This section provides the details of this structure. It is important to note that the structure of the lifecycle repository is really an implementation detail of the LifeCycle Service and is not part of the public interfaces provided by the service. The structure presented here is logical view and is not meant to imply that what is documented are actual name paths in the System name space. However, we go into the high level details of the structure in this section as it becomes an important piece of the puzzle in understanding factory finding.

There is another point to bring up before getting started. It is important to point out that workload managed server groups and homes which are workload management enabled are handled differently than servers and homes which do not participate in workload management. We will address the workload management specific issues after covering all the other ground.

There are six branches in repository, three major branches, each of which has two minor or sub branches. We can represent these branches of the tree as

follows:

BRANCH	SUB BRANCH
cell	factory
	server
workgroup	factory
	server
host	factory
	server

We will start with the registration of servers. All servers are registered in the host/server branch. Therefore we can envision server Abc appearing as:

BRANCH	SUB BRANCH	ENTRY
host	server	Abc

Next, all homes which are registered with lifecycle appear in both the host/factory branch and the host/server branch. Therefore, we would see the home of Policy objects which is deployed in the Abc server to be registered as:

BRANCH	SUB BRANCH	ENTRY
host	factory	Policy
	server	Abc Policy

Now, homes have the ability to say that they want to be visible within the workgroup and cell lifecycle repositories, or any combination of the preceding, as well. This is specified for the home when it is defined in Component Broker Object Builder. So, if the above Policy home had also specified visible in the cell lifecycle repository, it would appear in the cell/factory branch, as follows:

BRANCH	SUB BRANCH	ENTRY
cell	factory	Policy

Note that this did not result in the home being placed in the cell/server branch. That is only accomplished by having the server request that it be placed in the cell/server branch. This can be specified for the server when it is configured with the Component Broker System Manager. To make the example interesting, let's assume that the server requested to be visible in the workgroup lifecycle repository but not the cell. The resulting repository, considering all the above, would look like this:

BRANCH	SUB BRANCH	ENTRY
cell	factory	Policy
	server	
workgroup	factory	
	server	Abc Policy
host	factory	Policy
	server	Abc Policy

Notice that the visibility of the Policy home in the cell and workgroup branch is not entirely under it's own control. It does control it's appearance in the factory branch at these levels, but it is the server that controls it's visibility in the server branch at these levels. We will see in the section on Factory finding specifics how this affects the conditions under which the Policy home will be found.

So now it is time to introduce workload management enabled homes in workload managed server groups. When you configure a server group using Component Broker System Manager, it may or may not be workload managed. If it is a workload managed server group, it is placed into the lifecycle repository in a similar manner to a server, except that it is not placed at a host level, is always placed at a workgroup level and can optionally be placed at the cell level. So, if we add the Xyz workload managed server group it would appear as:

BRANCH	SUB BRANCH	ENTRY
workgroup	server	Xyz

A home can be enabled for workload management when it is defined using Component Broker Object Builder. When a home is so enabled, and when that home is deployed in a workload managed server group, the home is registered at the workgroup level rather than the host level and is registered as part of the server group rather than part of the individual servers within that server group. So, a workload managed home for Agent objects configured in the Xyz server group would appear in the lifecycle repository as follows:

BRANCH	SUB BRANCH	ENTRY
workgroup	factory	Agent
	server	Xyz Agent

Considering all of the above, the repository would now look like this:

BRANCH	SUB BRANCH	ENTRY	
cell	factory	Policy	
	server		
workgroup	factory	Agent	
	server	Abc	Policy
		Xyz	Agent
host	factory	Policy	
	server	Abc	Policy

Notice that there is no differentiation between the Abc server and the Xyz server group at the workgroup level of the repository. From a factory finding perspective, other than where they are initially registered, server groups are handled exactly as servers are for factory finding purposes.

This covers the structure of the lifecycle repository. The Factory finding specifics section will have an expanded example of the lifecycle repository and will describe how factory finding is affected by where homes, servers and server groups register themselves.

Factory keys

As you recall, the process of factory finding involves both “what” factory to find and “where” in the distributed environment to look for that factory. It is the client of a factory finder that provides the “what” and the location object contained by the factory finder that determine the “where”. This section takes a look at factory keys, the mechanism by which the client specifies “what” factory they want to find.

A factory key is specified as an argument in each of the factory-finder operations. This key is used to provide information about the factory that you want to get. With Component Broker, you can identify the desired factory based on the following parameters:

- The type of object that the factory creates
- The home in which the objects created by the factory exist
- The implementation of the object that the factory creates
- The interface name of an interface supported by the factory itself

object interface

Specification of this parameter is always required for a factory finding operation. It is defined to be the fully qualified IDL name given to the principal interface for your business object. This is what we have referred to up to this point as the objects type. For instance, in the

Personal Life Insurance Application Example (in *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*), PolicyHolder is the principal interface name for the policy holder object, which is derived from the Customer and Person interfaces. Note that this interface name is the abstract interface inherited by your business object and not the name of the business object itself, which would be PolicyHolderBO. If the PolicyHolder interface had been embedded in the LifeInsurance module, its fully qualified principal interface name would have been LifeInsurance::PolicyHolder. This is the value by which the home is registered with the lifecycle repository.

object home

Specification of this parameter is optional. It is defined to be the name given to the home when you defined it using Component Broker Object Builder, specifically the “Name as Factory” value. This can be used to distinguish between implementations of the same principal interface which have differing characteristics. For example, if you defined an interface called MyCollection and then provided both a transient and persistent implementation of the interface, you might call the homes MyCollectionPersistFactory and MyCollectionTransFactory. When a client needs to create a MyCollection and cares about the persistence or lack of persistence, he can qualify the request with this parameter to get the appropriate home.

object implementation

Specification of this parameter is optional. It is defined to be the name given to the managed object class generated by Component Broker Object Builder. This can be used to distinguish between specific implementations of the same principal interface. To continue the example from above, suppose there was a DB2 and an Oracle implementation of the persistent MyCollection. It would be reasonable to name both homes MyCollectionPersistFactory. For example, if a client wanted to create a persistent MyCollection, and specifically wanted it to be the DB2 implementation, he would use this parameter with a value such as MyModuleDB2MO::MyCollectionDB2MO to accomplish finding the right home.

factory interface

Specification of this parameter is optional. It allows you to ensure that the home returned supports a particular IDL interface. For example, you might have Policy objects configured with a standard IHome implementation, and then you develop a specialized home for Policy objects with an interface of MyPolicyHome that introduces new creation methods. In your distributed environment, you convert to the new home over time, so there is a period of time when some Policy

object homes may be only IHome homes and others are MyPolicyHome homes. Old client code that finds Policy object homes can use either, because MyPolicyHome is also an IHome. However, new client code may be written that takes advantage of the new creation methods on MyPolicyHome. When getting a Policy home, this new client code needs to be assured that the home he gets supports the MyPolicyHome interface. That is accomplished with this parameter.

As you can see, only “object interface” is required and thus the client code requesting the factory is isolated from having to know implementation specifics. However, the use of the other parameters gives the client control over the home returned when he does have concerns about the implementation of the objects created by the home or the interface supported by the home.

Factory finding specifics

This section presents detailed examples of factory finding. It relies heavily on an understanding of the information provided in the previous three sections (Defining scope of location, Lifecycle repository structure and Factory keys). Each example starts by presenting a LifeCycle Repository with a variety of homes registered. Based on the contents of this LifeCycle Repository, examples are walked through, showing which factories would be found when a FactoryFinder processes a request with a specific factory key, given a particular scope of location defined by the Location object contained in the Factory Finder.

Before we start with the examples, there is another point to be made about the branches in the lifecycle repository. There is a host branch for each host within your Component Broker environment, and there is a workgroup branch for each workgroup in your Component Broker environment. In the following examples, we will name the hosts and workgroups as there are cases where we show multiple host or workgroup branches to illustrate location scope capabilities of using a specific name rather than *LOCAL.

Example 1- one application on two servers on two hosts

In this example, we have the following situation:

- A Policy application which is not workload managed. The Policy home is specified to be visible at the workgroup level.
- Two hosts, H1 and H2, and each host is part of the same cell and workgroup (W1).
- Two free standing servers configured with the Policy application. S1 is on H1 and S2 is on H2. S2 is specified to be visible at the workgroup and at the cell level, but S1 is not.

In the example, the actual Policy homes in S1 and S2 will be called PH1 and PH2, respectively. The resulting repository is as follows:

BRANCH	SUB BRANCH	ENTRY		ACTUAL HOME
cell	factory			
	server	S2	Policy	PH2
workgroup -W1	factory		Policy	PH1
			Policy	PH2
	server	S2	Policy	PH2
host-H1	factory		Policy	PH1
	server	S1	Policy	PH1
host-H2	factory		Policy	PH2
	server	S2	Policy	PH2

A find factories request is made with object interface = Policy and no other parameters. The request is made to a factory finder which resides on H1 (resulting in *LOCAL being the same as H1). The following table shows which Policy homes will be returned for each specified scope of location. All possible legal scopes are shown (although some do not make sense, such as S2 with H1). Notation such as *LOCAL/H1 means either has the same meaning and result.

Cell	Workgroup	Host	Server	Homes Found	Example#
*LOCAL	*LOCAL/W1	*LOCAL/H1	*ANY	PH1	1
*LOCAL	*LOCAL/W1	*LOCAL/H1	S1	PH1	2
*LOCAL	*LOCAL/W1	*LOCAL/H1	S2	none	3
*LOCAL	*LOCAL/W1	H2	*ANY	PH2	4
*LOCAL	*LOCAL/W1	H2	S1	none	5
*LOCAL	*LOCAL/W1	H2	S2	PH2	6
*LOCAL	*LOCAL/W1	*IGNORE	*ANY	PH1 PH2	7
*LOCAL	*LOCAL/W1	*IGNORE	S1	none	8
*LOCAL	*LOCAL/W1	*IGNORE	S2	PH2	9
*LOCAL	*IGNORE	H1	*ANY	PH1	10
*LOCAL	*IGNORE	H1	S1	PH1	11
*LOCAL	*IGNORE	H1	S2	none	12
*LOCAL	*IGNORE	H2	*ANY	PH2	13
*LOCAL	*IGNORE	H2	S1	none	14
*LOCAL	*IGNORE	H2	S2	PH2	15

Cell	Workgroup	Host	Server	Homes Found	Example#
*LOCAL	*IGNORE	*IGNORE	*ANY	none	16
*LOCAL	*IGNORE	*IGNORE	S1	none	17
*LOCAL	*IGNORE	*IGNORE	S2	PH2	18

There are some points illustrated by this example that are worth stating. Although these points may have been stated or implied in previous sections, the example should help bring them into clear understanding.

- Cell does not equal everything - Using a cell scope of location (example 18) does not mean find all homes in the cell (every home on every server in every host in the cell). It simply means that you are looking for individual homes which have advertised themselves as being willing to be found in a cell scoped search.
- Server name versus *ANY - Using a server name at workgroup and cell levels is semantically different than using *ANY at these levels. Examples 7 through 9 and 16 through 18 illustrate this. *ANY only looks for homes which have specifically advertised themselves to be available at that level. Using a server name only looks for home which are in servers where the server has specifically advertised itself to have it's homes available at that level.
- Homes configured in Object Builder - Whether a home advertises itself at the workgroup and cell level is defined for the home with Object Builder. This means that whatever is specified holds true for all servers in which that application is deployed. Note example 7 finds both Policy homes because the Policy application said the Policy home would be visible in the workgroup
- Servers configured with the System Manager - Whether a server advertises itself at the workgroup and cell level is defined for the server in the System Manager. This means two servers with the same application can handle advertising themselves differently. Examples 8 and 9 and examples 17 and 18 illustrate this point, with the Policy home in S2 being found, but not the Policy home in S1.

Understanding these concepts can open the door to a variety of configuration possibilities. This enables powerful use of factory finding by your applications to deal with various goals in your distributed object deployment. However, it takes careful planning of how you want to configure your homes and servers along with defining the scopes of location that will be used with your factory finders in order to come up with a powerful solution.

Example 2 - one interface with three implementations

In this example, we will expand on the MyCollection example discussed in the "Factory keys" on page 133 section. We have the following situation:

- A single interface definition called MyCollection
- Three applications, each of which provides a different managed object implementation of the MyCollection interface.
 - A transient implementation
 - A persistent implementation which is backed by DB2
 - A persistent implementation which is backed by Oracle
 - None of the applications are workload managed.

The scenario is this. Some applications do not care which kind of collection they get, some care if it is transient or persistent and others care if it is persistent DB2 or persistent Oracle. All hosts will be configured with the transient implementation and one of the persistent implementations. For those who care about DB2 vs. Oracle, they will have to find the appropriate factory at the workgroup level. So following is the configuration information:

- Two hosts, H1 and H2, and each host is part of the same cell and workgroup (W1)
- Two free standing servers. S1 on H1 configured with the transient and DB2-backed applications. S2 on H2 configured with the transient and Oracle backed applications.
- The transient home is named TransientCollectionFactory (TCFact for short) and it's implementation is named TransientCollectionMO (TCMO).
- The DB2 persistent home is named PersistentCollectionFactory (PCFact) and it's implementation is named DB2CollectionMO (DCMO)
- The Oracle persistent home is also named PersistentCollectionFactory (PCFact) and it's implementation is named OracleCollectionMO (OCMO).
- The persistent homes are defined to be visible in workgroup, but the transient home is not.

In the example, the actual instances of the MyCollection homes in S1 will be TCH1 and DCH1 and in S2 will be TCH2 and OCH2. The resulting repository is as follows:

BRANCH	SUB BRANCH	ENTRY	ACTUAL HOME	HOME NAME	IMPL
cell	factory				
	server				
workgroup-W1	factory	MyCollection	DCH1	PCFact	DCMO
		MyCollection	OCH2	PCFact	OCMO
	server				
host-H1	factory	MyCollection	TCH1	TCFact	TCMO
		MyCollection	DCH1	PCFact	DCMO
	server	S1 MyCollection	TCH1	TCFact	TCMO
		MyCollection	DCH1	PCFact	DCMO

BRANCH	SUB BRANCH	ENTRY	ACTUAL HOME	HOME NAME	IMPL
host-H2	factory	MyCollection	TCH2	TCFact	TCMO
		MyCollection	OCH2	PCFact	OCMO
	server	S2 MyCollection	TCH2	TCFact	TCMO
		MyCollection	OCH2	PCFact	OCMO

To describe these scenarios we will use two different flavors of factory finders, which are:

- A local host scoped factory finder - cell=*LOCAL, workgroup=*LOCAL, host=*LOCAL, server=*ANY
- A local workgroup scoped factory finder - cell=*LOCAL, workgroup=*LOCAL, host=*IGNORE, server=*ANY

Clients may know some or all of the following about the configuration:

- All hosts have a collection home.
- All hosts have a transient collection home.
- All hosts have a persistent collection home.
- Persistent collections may be DB2 or Oracle.
- Persistent collection homes are visible at the workgroup level.

The following are the scenarios we indicated earlier along with all the details about the scenario:

1. Client wants any MyCollection

Client knows: All hosts have a collection home
Factory key: Object interface=MyCollection
Object home=TCFact
Local host: H1 H2
FactoryFinder used: Local host on H1 Local host on H2
Factories returned: TCH1 TCH2

2. Client wants transient MyCollection

Client knows: All hosts have a transient collection home
Factory key: Object interface=MyCollection
Object home=TCFact
Local host: H1 H2
FactoryFinder used: Local host on H1 Local host on H2
Factories returned: TCH1 TCH2

3. Client wants persistent MyCollection

Client knows:	All hosts have a persistent collection home	
Factory key:	Object interface=MyCollection	
	Object home=PCFact	
Local host:	H1	H2
FactoryFinder used:	Local host on H1	Local host on H2
Factories returned:	DCH1	OCH2

4. Client wants persistent DB2 backed MyCollection

Client knows:	Persistent collections may be DB2 or Oracle and persistent collection homes are visible at the workgroup level	
Factory key:	Object interface=MyCollection	
	Object implementation=DCMO	
Local host:	H1	H2
FactoryFinder used:	Local workgroup	Local workgroup
Factories returned:	DCH1	DCH1

It is worth noting that the factory finders used in the above examples all have equivalent instances provided as part of the set of Default lifecycle objects described in the next section.

As in the previous example, some points illustrated by this example are worth stating even if they have been stated or implied in previous sections.

- Clients do not have to know implementation specifics about objects to get their homes, but can make use of implementation information if they have a need to.
- The enterprise in this example had a strategy for how they would configure their environment for these differing implementations of the same interface. This strategy was known by client programs so that they could select factory finders which had appropriate scopes of location. Scenarios 1-3 knew to use local host factory finders while scenario 4 knew to use a workgroup scoped factory finder.
- The strategy selected still provides a lot of flexibility for changing specifics of the implementations and their deployment without affecting client code.

The key is that in a distributed environment you need to have a strategy for deployment of applications and associated configuration of factory finders. This strategy needs to be known by the client programs which will be accessing the homes.

Example 3 - workload managed server group and homes

This example takes a look at a workload managed server group. This is the situation:

- A Policy application which is not workload managed. The Policy home is specified to be visible at the workgroup level.
- An Agent application which is workload managed.
- Two hosts, H1 and H2, and each host is part of the same cell and workgroup (W1)
- A server group (SG) which is configured with both the Policy and the Agent applications.
- Server group SG has two servers, S1 on H1 and S2 on H2. The server group is specified to be visible at the cell level and also specifies that it's servers are visible at the cell level.

In the example, the actual Policy homes in S1 and S2 will be called PH1 and PH2, respectively. Now, there are real Agent homes on both S1 and S2, but references to them can actually get routed to either because they are workload management enabled. Therefore, we will call the Agent home AHsg. The resulting repository is as follows:

BRANCH	SUB BRANCH	ENTRY		ACTUAL HOME
cell	factory			
	server	S1	Policy	PH1
		S2	Policy	PH2
		SG	Agent	AHsg
workgroup-W1	factory			PH1
				PH2
				AHsg
	server	SG	Agent	AHsg
host-H1	factory			PH1
	server	S1	Policy	PH1
host-H2	factory			PH2
	server	S2	Policy	PH2

The purpose of this example is to point out the specifics of factory registration for workload managed server groups and homes. These are the points that need to be understood:

- Although Policy is configured as part of the workload managed server group SG, the Policy home is not enabled for workload management. It is

therefore registered in the repository in the same way as if both S1 and S2 were free standing servers and not part of a workload managed server group.

- Although both S1 and S2 actually have instances of the Agent home, these Agent homes are not registered in the S1 or S2 server branches nor are they registered in the factory branch of the H1 and H2 host branches. The rationale for this has to do with honoring scope of location specifications. Suppose there was a factory finder with host=H1 and it returned a reference to the Agent home on H1. When the client then invoked a method on the Agent home, it could be processed by the H2 Agent home or the H1 agent home because the Agent home reference is a workload managed reference. This would violate the scope of location in the factory finder used to find the home.
- Workload managed server groups are supposed to have all of the servers they contain be on hosts within the same workgroup. Therefore, it is valid to return workload managed object references from a factory finder which is scoped with workgroup=W1, host=*IGNORE. Specifically, we know that a create request sent to an Agent home will result in an object being created within the workgroup (handled by some Agent home instance on a server on a host in the workgroup).
- At the workgroup level, in the factory branch, there is no distinction between the Agent and the Policy homes. A factory finder with workgroup=W1, host=*IGNORE will return AHsg for a factory key of object interface=Agent and will return PH1 and PH2 for a factory key of object interface=Policy.
- At the workgroup or at the cell level, in the server branch, there is no distinction made between a server and a server group, provided the server has been made visible in the workgroup or cell and the server group has been made visible in the cell. Remember that a server group never shows up in a host server branch and always shows up in a workgroup server branch.
- When using a scope of location that contains a server name or server group name, only non workload managed homes will be found if it is a server name and only workload managed homes will be found if it is a server group name.

These points bring out the question of strategy for home registration and factory finding when you have a mix of workload managed and non workload managed homes, and you want to isolate your client code from the distinction between the two. Assuming you do not have multiple workgroups with differing configurations, the simplest approach is to have all homes make themselves visible at the workgroup level. Then use a workgroup scoped factory finder for all of your factory finding. This will give you access to all of your homes and you will not have to deal with server names and server group names. Of course, there may be other factors than workload

management that enter into your consideration. In that case, the workload management factors presented here need to be understood as you analyze your situation.

Example 4 - using a widening of scope

Up until now, the examples have all dealt with factory finding operations where the factory finder's scope of location was specified with one Scope structure. Remember, however, that location objects return a sequence of Scope structures, each of which is considered in the factory finding operation. We will look at a technique referred to as widening of scope, where the factory finder contains a location object whose sequence of scopes are each at progressively wider scope. This can be used to locate factories that are as "near as possible" to you.

This is the configuration information for this example:

- There is one server group (SGroup) which has only 1 server (S1sg)
- There are three hosts:
 - H1 - contains server group server S1sg and a free standing server S1fs
 - H2 - contains free standing server S2fs
 - H3 - contains free standing server S3fs
- There are two workgroups:
 - Wx - contains host H1 and H2
 - Wz - contains H3
- There are five applications, configured as follows:
 - Agent - workload managed home in SGroup
 - Customer - non workload managed home in the server groups server S1sg
 - Policy - in free standing server S1fs
 - Claim - in free standing server S2fs
 - Account - in free standing server S3fs

All of the application homes are defined to be visible in cell and visible in workgroup (visible in workgroup is irrelevant for the workload managed home, but it is not inappropriate, simply redundant).

- None of the servers nor the server group are defined to be visible at any level other than their default.

Given all of the above, we end up with a factory repository that looks like the following:

BRANCH	SUB BRANCH	ENTRY		ACTUAL HOME
cell	factory	Agent		AgentSGroup
		Customer		CustomerS1sg
		Policy		PolicyS1fs
		Claim		ClaimS2fs
	Account		AccountS3fs	
	server			
workgroup-WxD	factory	Agent		AgentSGroup
		Customer		CustomerS1sg
		Policy		PolicyS1fs
	Claim		ClaimS2fs	
	server	SGroup	Agent	AgentSGroup
workgroup-Wz	factory	Account		AccountS3fs
	server			
host-H1	factory	Customer		CustomerS1sg
		Policy		PolicyS1fs
	server	S1sg	Customer	CustomerS1sg
		S1fs	Policy	PolicyS1fs
host-H2	factory	Claim		ClaimS2fs
	server	S2fs	Claim	ClaimS2fs
host-H3	factory	Account		AccountS3fs
		S3fs	Account	AccountS3fs

We will now introduce one of the default lifecycle objects that will be described in the next section, Default lifecycle objects. This factory finder is referred to as the server-scope-widened factory finder, and there is one of these for every server in your configuration. It is defined to have a scope of location defined by the following five scope structures:

Order	Cell	Workgroup	Host	Server	Comments
1	*LOCAL	*IGNORE	host name	server name	
2	*LOCAL	workgroup name	*IGNORE	servergroup name	only if workload managed server
3	*LOCAL	*IGNORE	host name	*ANY	
4	*LOCAL	workgroup name	*IGNORE	*ANY	
5	*LOCAL	*IGNORE	*IGNORE	*ANY	

So now lets assume I have some code running in server S1sg. Possibly it is an application factory that has been developed which encapsulates business logic about the creation of Agent, Customer, Policy, Claim and Account objects. This application factory wants to be independent of the configuration, but would like to create objects as close to itself as possible. Therefore, it uses the S1sg-server-scope-widened factory finder. It will make find_factory calls on the factory finder rather than find_factories because it wants the factory finder to only return the first factory that it finds that satisfies the request rather than all possible factory finders that satisfy the request. Lets look at how the homes for each of these object types would be found.

Customer

Home is in the same server. Using the first scope (host H1, server S1sg) the Customer home is found in the H1 server branch S1sg.

Agent Workload managed home is in the same server group that this server is in. The first scope fails to locate the Agent home. The second scope (workgroup Wx, server SGroup) finds the Agent home in the Wx server branch SGroup.

Policy Home is in a different server on the same host. The first and second scopes fail to locate the Policy home. The third scope (host H1, server *ANY) finds the Policy home in the H1 factory branch.

Claim Home is in a different host but in the same workgroup. The first through third scopes all fail to locate the Claim home. The fourth scope (workgroup Wx, server *ANY) finds the Claim home in the Wx factory branch.

Account

Home is in a different host and workgroup. The first through fourth scopes all fail to location the Account home. The fifth scope (cell *LOCAL, server *ANY) finds the account home in the cell factory branch.

As you can see, by using the one factory finder the application factory was always able to find the home that was closest to himself.

This example may seem contrived because there is only one instance of each home, and using a cell scoped factory finder would have located the exact same homes just fine. However, this was only to simplify the table showing the lifecycle repository for the example. Imagine there are several server groups, each containing several servers and there are many more free standing servers, all across ten or twenty hosts organized into several workgroups. Or imagine a configuration an order of magnitude larger than that. Let's also assume there is no consistency to how the applications are configured across these servers and server groups. The technique described in

this example will work just as well in that environment, provided all homes are made visible in workgroup and visible in cell.

There is another point about the application factory knowing to use the `S1sg-server-scope-widened` factory finder. Would this pose a problem if this same application factory were deployed in tens or hundreds of servers in the environment? The implementation of the application factory handles this by accessing the name of the server it is running in, getting the `<servername>-server-scope-widened` factory finder from the name space and using that. This way, the application factory is always looking for homes that are closest to itself, no matter where it exists in the environment.

Default lifecycle objects

The default lifecycle objects are your starting point for factory finding. Component Broker creates several different default factory finder objects (configured with the appropriate location objects) which are generally useful. While doing development, these default factory finders may supply all of the factory finding capability that you need. When it comes to deployment of your applications in a distributed environment, it is more likely that you may have to create or configure some factory finder and location objects that reflect your particular strategy for factory finding. However, even in this case, the default location objects may well be used as building blocks to the definition of the location objects you need to define.

Whenever a new server is started for the first time, there are default lifecycle objects created that represent that server. All application and name servers get default lifecycle objects whose scope of location is specific to that server. If the server is a name server, additional default lifecycle objects are created, providing scope of location to the major branches in the System name space (that is, host, workgroup and cell). In addition, if you have a workload managed server group, there will be default lifecycle objects whose scope of location is specific to the server group.

All of these default lifecycle objects are bound into the system name space so that they can be easily accessed by your client applications and business object applications. The scheme for how they are named and where they are bound provides an easy and intuitive way for you to know how to construct the name for the factory finder that you want to use.

The default lifecycle objects occur in pairs, a `FactoryFinder` object with its contained `SingleLocation` or `OrderedLocation` object. Although we have seen that it is almost always the factory finder that you want to access from the name space, both the factory finder and location objects are bound into the name space. Each pair has the equivalent name, with the factory finder being bound into a context for factory finders and the location object being bound

into a context for location objects. You might want to access a location object from the system name space if it was being used as one of the locations for initializing an OrderedLocation.

There are two basic flavors of factory finder with location object pairs. The first uses a SingleLocation object and is used to define a scope of location with only one Scope structure. These are used to scope to a specific server, server group or branch of the system name tree. The other flavor are the widened locations (see Using a Widening of Scope) which start with a Scope structure for some specific point and then have progressively broader scopes defined. These widened locations are OrderedLocation objects which are configured with an appropriate sequence of the default SingleLocation objects.

The following documentation of the default lifecycle objects presents them in their factory finder and location object pairs. The pairs are:

- host-scope
- workgroup-scope
- cell-scope
- <servername>-server-scope
- <servergroupname>-server-scope
- host-scope-widened
- workgroup-scope-widened
- <servername>-server-scope-widened
- <servergroupname>-server-scope-widened

A description of each is given, followed by the exact specification of the Scope structure(s) with which they operate. Then a list of the names where the factory finder and location object are bound into the name space is given.

host-scope

The host-scope lifecycle objects are used to define a scope of location which includes an entire host. They can be used to find a home from the factory branch of the host lifecycle repository. This includes all non workload managed homes in all the servers configured on that host. The Scope structure for the location object contains:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	*IGNORE	hostname	*ANY	*ANY	*ANY

They are bound into the name space at:

```

/host/resources/factory-finders/host-scope
/host/resources/locations/host-scope

```

workgroup-scope

The workgroup-scope lifecycle objects are used to define a scope of location which includes an entire workgroup. They can be used to find a home from the factory branch of the workgroup lifecycle repository. This includes, from all hosts in this workgroup, all workload managed homes in server groups and all non workload managed homes which are defined to be visible in workgroup. The Scope structure for the location object contains:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	workgroup name	*IGNORE	*ANY	*ANY	*ANY

They are bound into the name space at:

```
/workgroup/resources/factory-finders/workgroup-scope  
/workgroup/resources/locations/workgroup-scope
```

cell-scope

The cell-scope lifecycle objects are used to define a scope of location which includes an entire cell. They can be used to find a home from the factory branch of the cell lifecycle repository. This includes, from all hosts in this cell, all homes (workload managed and non workload managed) which are defined to be visible in cell. The Scope structure for the location object contains:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	*IGNORE	*IGNORE	*ANY	*ANY	*ANY

They are bound into the name space at:

```
/cell/resources/factory-finders/cell-scope  
/cell/resources/locations/cell-scope
```

<servername>-server-scope

The <servername>-server-scope lifecycle objects are used to define a scope of location which includes an entire server. They can be used to find a home from the server branch of the host lifecycle repository. This includes all homes in the server which are non workload managed. These objects are registered in the cell, workgroup and host branches of the system name space. This allows you to find them with only knowledge of the server name without having to know which host they are on. However, it makes no difference if you access them from the cell, workgroup or host branch. The resulting scope of location is always to the host server branch of the lifecycle repository. The Scope structure for the location object contains:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	*IGNORE	hostname	servername	*ANY	*ANY

They are bound into the name space at:

```

/host/resources/factory-finders/<servername>-server-scope
/workgroup/resources/factory-finders/<servername>-server-scope
/cell/resources/factory-finders/<servername>-server-scope

/host/resources/locations/<servername>-server-scope
/workgroup/resources/locations/<servername>-server-scope
/cell/resources/locations/<servername>-server-scope

```

<servergroupname>-server-scope

The <servergroupname>-server-scope lifecycle objects are used to define a scope of location which includes an entire servergroup. They can be used to find a home from the server branch of the workgroup lifecycle repository. This includes all homes in the server group which are workload managed. These objects are registered in the cell and workgroup branches of the system name space. This allows you to find them with only knowledge of the server group name without having to know which workgroup they are in. However, it makes no difference if you access them from the cell or workgroup branch. The resulting scope of location is always to the workgroup server branch of the lifecycle repository. The Scope structure for the location object contains:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	workgroupname	*IGNORE	servergroupname	*ANY	*ANY

They are bound into the name space at:

```

/workgroup/resources/factory-finders/<servergroupname>-server-scope
/cell/resources/factory-finders/<servergroupname>-server-scope

/workgroup/resources/locations/<servergroupname>-server-scope
/cell/resources/locations/<servergroupname>-server-scope

```

host-scope-widened

The host-scope-widened lifecycle objects are used to define a scope of location which starts at the host, then goes to the workgroup and finally the cell. An explanation of each scope in this sequence can be found above at host-scope, workgroup-scope and cell-scope. The OrderedLocation is simply a sequence of these other locations. The following chart shows these locations along with the Scope structure defined by each:

Location	Cell	Workgroup	Host	Server	Container	Home
host-scope	*LOCAL	*IGNORE	hostname	*ANY	*ANY	*ANY

Location	Cell	Workgroup	Host	Server	Container	Home
workgroup-scope	*LOCAL	workgroupname	*IGNORE	*ANY	*ANY	*ANY
cell-scope	*LOCAL	*IGNORE	*IGNORE	*ANY	*ANY	*ANY

They are bound into the name space at:

```
/host/resources/factory-finders/host-scope-widened
/host/resources/locations/host-scope-widened
```

workgroup-scope-widened

The workgroup-scope-widened lifecycle objects are used to define a scope of location which starts at the workgroup and then goes to the cell. An explanation of each scope in this sequence can be found above at “workgroup-scope” on page 148 and cell-scope. The OrderedLocation is simply a sequence of these other locations. The following chart shows these locations along with the Scope structure defined by each:

Location	Cell	Workgroup	Host	Server	Container	Home
workgroup-scope	*LOCAL	workgroupname	*IGNORE	*ANY	*ANY	*ANY
cell-scope	*LOCAL	*IGNORE	*IGNORE	*ANY	*ANY	*ANY

They are bound into the name space at:

```
/workgroup/resources/factory-finders/workgroup-scope-widened
/workgroup/resources/locations/workgroup-scope-widened
```

<servername>-server-scope-widened

The <servername>-server-scope-widened lifecycle objects are used to define a scope of location which starts at the server, then goes to the server group (if this is a workload managed server), then the host, the workgroup and finally the cell. An explanation of each scope in this sequence can be found above at <servername>-server-scope, <servergroupname>-server-scope, host-scope, workgroup-scope and cell-scope. These objects are registered in the cell, workgroup and host branches of the system name space. This allows you to find them with only knowledge of the server name without having to know which host they are on. The OrderedLocation is simply a sequence of these other locations. The following chart shows these locations along with the Scope structure defined by each:

Location	Cell	Workgroup	Host	Server	Container
<servername>-server-scope	*LOCAL	*IGNORE	hostname	servername	*ANY

Location	Cell	Workgroup	Host	Server	Container
<servergroup>-server-scope (only if workload managed server)	*LOCAL	workgroupname	*IGNORE	servergroupname	*ANY
host-scope	*LOCAL	*IGNORE	hostname	*ANY	*ANY
workgroup-scope	*LOCAL	workgroupname	*IGNORE	*ANY	*ANY
cell-scope	*LOCAL	*IGNORE	*IGNORE	*ANY	*ANY

They are bound into the name space at:

```
/host/resources/factory-finders/<servername>-server-scope-widened
/workgroup/resources/factory-finders/<servername>-server-scope-widened
/cell/resources/factory-finders/<servername>-server-scope-widened
```

```
/host/resources/locations/<servername>-server-scope-widened
/workgroup/resources/locations/<servername>-server-scope-widened
/cell/resources/locations/<servername>-server-scope-widened
```

<servergroupname>-server-scope-widened

The <servergroupname>-server-scope-widened lifecycle objects are used to define a scope of location which starts at the server group, then goes to the workgroup and finally the cell. An explanation of each scope in this sequence can be found above at <servergroupname>-server-scope, workgroup-scope and cell-scope. These objects are registered in the cell and workgroup branches of the system name space. This allows you to find them with only knowledge of the server group name without having to know which workgroup they are in. The OrderedLocation is simply a sequence of these other locations. The following chart shows these locations along with the Scope structure defined by each:

Location	Cell	Workgroup	Host	Server	Container
<servergroupname>-server-scope	*LOCAL	workgroupname	*IGNORE	servergroupname	*ANY
workgroup-scope	*LOCAL	workgroupname	*IGNORE	*ANY	*ANY
cell-scope	*LOCAL	*IGNORE	*IGNORE	*ANY	*ANY

They are bound into the name space at:

```
/workgroup/resources/factory-finders/<servergroupname>-server-scope-widened
/cell/resources/factory-finders/<servergroupname>-server-scope-widened
```

```
/workgroup/resources/locations/<servergroupname>-server-scope-widened
/cell/resources/locations/<servergroupname>-server-scope-widened
```

Tips for using the default lifecycle objects

This section points out a few things that are good to know if you are to take full advantage of the capabilities provided by the default lifecycle objects.

1. **Accessing Default LifeCycle Objects on Other Hosts and Workgroups** In Component Broker, the code you run will be associated with a bootstrap host. When you use something like `/host` or `/workgroup` to start a name path, you are looking in the host or workgroup branch of the system name space that is associated with your bootstrap host. There are times that you would like to obtain a resource from the host or workgroup tree of a different host or workgroup (assuming that you know the name of the host or workgroup). For example, you might want to obtain the host-scope factory finder for host `abc.austin.ibm.com`. To do this, you would substitute `/cell/hosts/abc.austin.ibm.com` for `/host` that appears in the above documentation. Specifically, rather than specifying:

```
/host/resources/factory-finders/host-scope
```

you would specify:

```
/cell/hosts/abc.austin.ibm.com/resources/factory-finders/host-scope
```

Similarly, if you wanted the workgroup-scope factory finder for WorkgroupXYZ, you would specify:

```
/cell/workgroups/WorkgroupXYZ/resources/factory-finders/workgroup-scope
```

See the documentation for the System name space to see how this name path gets you to the desired factory finder.

2. **Best Way to Access a Server Scope Factory Finders** There are times when you will want to access the `<servername>-server-scope` factory finder, or it's widened counterpart, for a specific server. If you do this using the local host tree (that is, using `/host`) you have made the assumption that the server is on the same host that is your bootstrap host. In order to avoid this dependency, and since all of the server scope factory finders are registered at the cell level as well, you should access it using the cell name tree. For example, to access the server called `knownServerName`, you should specify: `/cell/resources/factory-finders/knownServerName-server-scope`. This practice also holds true for server group names, isolating you from having to be in the same workgroup.
3. **Determining the Name of the Current Server** In the previous tip, you need to know the name of the server whose factory finder you are accessing. However, very often the scenario is that you develop a business object that will run in many servers, and you want to obtain the `<servername>-server-scope` factory finder for the server your business object is running in. You can dynamically determine the name of the server your code is running in by using the `CBSeriesGlobal::serverName()` method. Once you have obtained the server name you can build the

appropriate name path, for example: /host/resources/factory-finders/myCurrentServer-server-scope. Unlike our previous tip which utilized the cell name tree, we know that in this case, using the local host name tree will work fine because our code is running in that server on that host, and therefore it is our local host.

Lifecycle interfaces and implementations

The Detailed view of location-based factory finding section has provided information for an in depth understanding of the underlying behavior and semantics of factory finding operations. This section continues on, providing the details about the interfaces used in factory finding, specifically the `FactoryFinder`, `SingleLocation`, and `OrderedLocation` interfaces. In addition to the interfaces, we will look at the different implementations, specifically the managed object versions and the local only versions. A major part of looking at the implementations will be how you actually create instances, addressing the use of specialized homes for the managed object versions. For local only implementations, we will look at the C++ creation methods. There is currently not a local only implementation of these objects in Java, so Java clients should use the managed object versions.

The section is written to provide you with an understanding of the interfaces and how to use them. However, full details of the syntax for all of the operations should be obtained from the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*, specifically the chapters on the `CosLifeCycle`, `IExtendedLifeCycle`, `ILifeCycleLocalObjectImpl` and `ILifeCycleManagedClient` IDL modules. This section is intended to provide the bridge needed to tie the semantics of factory finding presented in the previous sections of this Advanced Programming Guide with the detailed syntax specification in the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*. In addition, a detailed programming example will be provided in the section following this one.

The following topics are discussed in this section:

- Lifecycle object interfaces
- Scope structures and strings
- Scope manipulator interface
- Location interface
- `SingleLocation` interface
- `OrderLocation` interface
- Factory key structures and strings
- `FactoryFinder` interface
- Managed vs. local only implementations
- Local only creation mechanisms
- Accessing managed lifecycle objects from the name space

- Creating managed lifecycle objects
- Using the SingleLocationHome interface
- Using the OrderedLocationHome interface
- Using the FactoryFinderHome interface
- Configuring managed lifecycle objects

These sections will show code snippets. A complete coding example follows in the Lifecycle example section.

Lifecycle object interfaces

The interfaces in this section are presented in an order determined by usage, building such that each subsequent sections has all prerequisite knowledge previously defined.

Scope structures and strings

The Scope structure contains six strings representing the six topological and infrastructural boundary elements which define a scope of location. We have already looked extensively at the contents and meaning of a scope of location, particularly in the Defining scope of location section. Therefore, we will not reiterate the appropriate values contained by these fields. We also know that location objects return a sequence of these scope structures. The Scope structure and OrderedScopes sequence are defined in the IExtendedLifeCycle module as follows:

```
struct Scope
{
    CosNaming::Istring cell;
    CosNaming::Istring workgroup;
    CosNaming::Istring host;
    CosNaming::Istring server;
    CosNaming::Istring container;
    CosNaming::Istring home;
};
typedef sequence OrderedScopes;
```

In addition to the structure form of Scope, there is also a string form that is constructed using the name string syntax which is supported by the Naming Service. The string form, ScopeString, represents the contents of a single scope structure. There is also an OrderedScopeStrings to represent of sequence of scopes. They are defined in the IExtendedLifeCycle module as follows:

```
typedef CosNaming::Istring ScopeString;
typedef sequence OrderedScopeStrings;
```

The syntax for expressing scope boundary values scope boundary values in the string form is illustrated in the following example:

```
<cell-value>.cell/<workgroup-value>.workgroup/
<host-value>.host/<server-value>.server/
<container-value>.container/<home-value>.home
```

Since this form is the name string syntax, we can think of the this in terms of a `CosNaming::Name`. In each name component, the kind field is used to identify the boundary value, and the ID field is used to specify the value (denoted in angle-brackets in the above example). Assuming we needed a scope which identified a specific server in a specific workgroup, the scope boundary could be expressed as in the following example:

```
*LOCAL.cell/consumer insurance.workgroup/*IGNORE.host/  
whole life.server/*ANY.container/*ANY.home
```

The string forms of the scope structure and sequence are introduced to provide a convenient way to pass this information when creating a lifecycle object. When doing so, not all six fields of the structure need to be specified. Unspecified fields will be given the following default values:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	*LOCAL	*LOCAL	*ANY	*ANY	*ANY

Scope manipulator interface

The `ScopeManipulator` interface provides mechanisms to convert between the structure and string forms of a scope. This interface, which is defined in the `IExtendedLifecycle` module, provides the operations:

- `scope_to_string`
- `string_to_scope`

These operations are typically not needed by your client code using the `LifeCycle Service`. See the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* if you need additional information on these operations.

Location interface

The `Location` interface provides everything that is needed for a factory finder object to obtain scope information from any location object. As we have already discussed in `Interaction between FactoryFinder and location objects` and in `Location object implementations`, the `Location` interface is an abstract interface. There is never an implementation which is just a `Location`, but it is always sub typed and an implementation for the sub type is provided. `Component Broker` provides implementations of two sub types of this interface, `SingleLocation` and `OrderedLocation`. In addition, you can sub type and implement your own form of `Location` objects if you have unique requirements for determination of scope of location that is not satisfied by the `Component Broker` provided implementations.

The `Location` interface is defined in the `IExtendedLifecycle` module, and provides the single operation:

- `get_scopes`

This operation returns an `OrderedScopes` sequence, which is used by the factory finder object to define the scope of location where it will perform factory searches. There should be no preconceived notion of how the location object implementation determines the values in the `Scope` structures of the returned `OrderedScopes`. An implementation might have fixed values or values which are dynamically determined at run time based on some condition or constraint.

This operation is typically not needed by your client code using the `LifeCycle Service`. See the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* if you need additional information on this operation.

SingleLocation interface

The `SingleLocation` interface is defined in `IExtendedLifeCycle` and it inherits the `Location` interface. `SingleLocation` is simply an interface which specifies that an implementation of this interface will limit itself to returning a single `Scope` structure within the `OrderedScopes` sequence returned by `get_scopes`. The `Component Broker` implementations of this interface are initialized when created to contain a static set of the six boundary values contained in a scope. As with the `Location` interface, someone could conceivably provide an implementation of the `SingleLocation` interface which dynamically determined the scope values.

The `SingleLocation` interface introduces one new operation:

- `get_scope()`

This operation is intended to make it easier to query the `SingleLocation`, asking it to return a `Scope` structure rather than an `OrderedScopes`. This operation is typically not needed by your client code using the `LifeCycle Service`. See the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* if you need additional information on this operation.

OrderLocation interface

The `OrderLocation` interface is defined in `IExtendedLifeCycle` and it inherits the `Location` interface.

`OrderLocation` is an interface which specifies that an implementation should maintain a sequence of other `Location` objects. In response to a `get_scopes`, an ordered location will return an `OrderedScopes` which is the concatenation of all the `OrderedScopes`, obtained in sequence, returned by the `Location` objects it contains. The `Component Broker` implementations of this interface are initialized when created to contain a static set of other `Location` objects.

The SingleLocation interface introduces one new operation:

- `get_locations()`

This operation returns the sequence of location objects it contains. This operation is typically not needed by your client code using the LifeCycle Service. See the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* if you need additional information on this operation.

Factory key structures and strings

Factory finding operations take a factory key as input. We have already extensively covered the semantic meaning of a factory key in the Factory keys section, and we will now take a look at the syntax. The type `Key` is introduced in the `CosLifeCycle` module and is defined to be structured the same as a `CosNaming::Name`. They are defined as follows:

From `CosNaming`:

```
struct NameComponent
{
    Istring id;
    Istring kind;
};
typedef sequence Name;
```

From `CosLifeCycle`:

```
typedef CosNaming::Name Key;
```

As you can see, the `CosLifeCycle::Key` is actually a sequence of structures where each element contains two string fields: an *ID* and a *kind* field. These are treated as name value pairs, where the ID field contains the value and the kind field contains the name. The valid names that can be used in the kind field are:

- Object interface
- Object home
- Object implementation
- Factory interface

When the kind-field of the element is set to “object interface”, then the ID-field should be set with the fully qualified interface name of the object type the factory creates. This key type is required. When the kind-field of the element is set to “object home”, then the ID-field should be set with the name of the home. This key type is optional. When the kind-field of the element is set to “object implementation”, then the ID-field should be set with the name of the implementation class of the objects created by the factory. When the kind-field of the element is set to “factory interface”, then the ID-field should be set with the interface ID of the factory itself. The interface ID is a string which is generated by the IDL compiler for each interface in the IDL. This

string can be accessed using the <interface name>_RID variable. This key type is optional. See the Factory keys section for more complete explanation of these name value pairs.

Referring back to Example 2 - one interface with three implementations, suppose we wanted to get a factory for a persistent MyCollection which was backed by DB2 and we also wanted to make sure the factory returned supported the MyModule::MyCollectionHome interface. To do this, you could create a key as follows:

```
CosLifeCycle::Key      key(4);
key.length(4);
key[0].kind = CORBA::string_dup("object interface");
key[0].id   = CORBA::string_dup("MyModule::MyCollection");
key[1].kind = CORBA::string_dup("object home");
key[1].id   = CORBA::string_dup("PersistentCollectionFactory");
key[2].kind = CORBA::string_dup("factory interface");
key[2].id   = CORBA::string_dup
  (::MyModule::MyCollectionHome:MyCollectionHome_RID);
key[3].kind = CORBA::string_dup("object implementation");
key[3].id   = CORBA::string_dup("MyModuleDB2MO::MyCollectionDB2MO");
```

The elements of the key structure may appear in any order. If the kind-field of a key element is not one of the four recognized strings, that element is ignored. No error is generated in this case. Also, typically you would not create a key with all four of these items specified. Most usage of the key only has "object interface" specified, and in some cases one of the other values is added.

As you can see, the setup of a factory key structure does not look like the simplest thing to do when all you want is to provide an input argument that you will be passing to a factory finding operation. Because of this, Component Broker provides a string form of the CosLifeCycle::Key definition. The string form is called a FactoryKeyString and is defined in the IExtendedLifeCycle module, as follows: CosNaming::Istring FactoryKeyString;. Similar to the ScopeString described earlier, the FactoryKeyString follows the name string syntax. Therefore, the factory key string used to find a factory for any kind of a MyModule::MyCollection would be: "MyModule::MyCollection.object interface". To find a factory for the persistent version of a MyModule::MyCollection, the factory key string would look like this: "MyModule::MyCollection.object interface/PersistentCollectionFactory.object home"

FactoryFinder interface

The FactoryFinder interface is introduced by the CosLifeCycle module and extended by the IExtendedLifeCycle module. This is the interface you will be using for finding factories (homes) which you will use for the creation of managed objects.

In the `CosLifeCycle` module, `FactoryFinder` introduces one operation, `find_factories`. This operation uses a `CosLifeCycle::Key` for input and returns all factories which satisfy the request. In `Component Broker`, the `FactoryFinder` interface has been extended in the `IExtendedLifeCycle` module to introduce operations that can return a single factory and also to make use of the simpler `FactoryKeyString` as input. This section does not differentiate between the modules which introduce the operations, but when you refer to the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* you will have to consider both the `CosLifeCycle` and `IExtendedLifeCycle` documentation for `FactoryFinder`. The operations supported by the `FactoryFinder` interface are:

- `find_factories`
- `find_factory`
- `find_factories_from_string`
- `find_factory_from_string`
- `get_location`

The first two operations, `find_factories` and `find_factory`, each take a `CosLifeCycle::Key` as input. As the names imply, `find_factory` returns a single factory (the first found which satisfies the request) and `find_factories` returns all factories that satisfy the request. They throw the `CosLifeCycle::NoFactory` exception if they are not able to find an appropriate factory.

The next two operations, `find_factories_from_string` and `find_factory_from_string`, each take an `IExtendedLifeCycle::FactoryKeyString` as input. The operations work in exactly the same way as their counterparts which take a `CosLifeCycle::Key`.

The last operation, `get_location`, can be used when you want to obtain a reference to the location object that the factory finder contains.

The scope of location within which factory finders search for a factory is defined by the contained location object. When a `find_factories` or `find_factories_from_string` operation returns all factories, it specifically means all factories within the scope of location configured for the factory finder, not all factories in the environment. It is possible for a particular factory to occur more than once in the sequence of factories returned from these operations. This can occur when the configured location object returns multiple scopes in the `OrderedScopes` sequence. As we learned in the `Lifecycle repository structure` section, a factory may be registered into many branches of the repository. Therefore, it may be found for more than one scope of location during the `find_factories` operation. The implementation of this interface in `Component Broker` makes no attempt to remove duplicates from the returned sequence of factories.

The following examples show usage of these factory finding operations. The `find_factory` and `find_factories_from_string` operations are shown. The `find_factory_from_string` and `find_factories` handling can be extrapolated from the two examples given. In general, the `find_factory_from_string` operation is the one that is most often used. It returns one factory (generally all you want) and has the simplest input argument.

Since we have not yet discussed how to obtain or create an instance of a factory finder, the examples assume that we already have a reference to one.

Example 1 - `find_factory`: This example does a `find_factory` operation to get a reference to a home that makes `PolicyModule::Policy` objects. This operation takes a `CosLifeCycle::Key` as input and returns a `CosLifeCycle::Factory`. The return value must be narrowed to the `IManagedClient::IHome` interface before it can be used as a home in subsequent code. The appropriate error handling code has been included. The `try catch` block handles the `CosLifeCycle::NoFactory` exception in the case where the factory finder didn't find a factory to return. The `if else` clause tests to see if the `_narrow` operation failed or if the catch block was entered.

```
// Declare the variables (assume myFF initialized)
IExtendedLifeCycle::FactoryFinder_var myFF;
CosLifeCycle::Factory_var homeAsFactory;
IManagedClient::IHome_var policyHome;
CosLifeCycle::Key key(1);

// Initialize factory key
key.length(1);
key[0].kind = CORBA::string_dup("object interface");
key[0].id = CORBA::string_dup("PolicyModule::Policy");

// Find the home and narrow to appropriate reference
try
{
    homeAsFactory = myFF->find_factory(key);
    policyHome = IManagedClient::IHome::_narrow(homeAsFactory);
}
catch(CosLifeCycle::NoFactory &e)
{
    policyHome = IManagedClient::IHome::_nil();
}
// Test if we were successful
if (CORBA::is_nil(policyHome))
    cout << "failed";
else
    cout << "success";
```

Example 2 - `find_factories_from_string`: This example does a `find_factories_from_string` operation to get a sequence of references to homes that make `PolicyModule::Policy` objects. This operation takes an `IExtendedLifeCycle::FactoryKeyString` as input and returns

CosLifeCycle::Factories sequence. Each element of the return value must be narrowed to the IManagedClient::IHome interface before they can be used as a home in subsequent code. The appropriate error handling code has been included. The *try catch* block handles the CosLifeCycle::NoFactory exception in the case where the factory finder didn't find a factory to return. The *if else* clause inside the loop tests to see if the *_narrow* operation failed.

```
// Declare the variables (assume myFF initialized)
IExtendedLifeCycle::FactoryFinder_var myFF;
CosLifeCycle::Factories_var factories;
IManagedClient::IHome_var policyHome;
IExtendedLifeCycle::FactoryKeyString key;

// Initialize factory key
key = CORBA::string_dup("PolicyModule::Policy.object interface");

// Find the homes
try
{
    factories = myFF->find_factories_from_string(key);
}
catch(CosLifeCycle::NoFactory &e)
{
    cout << "failed";
    exit;
}

// Narrow each home to the appropriate interface
for (int i=0; i < factories->length(); i++)
{
    policyHome = IManagedClient::IHome::_narrow(factories[i]);
    if (CORBA::is_nil(policyHome))
        cout << "failed";
    else
        cout << "success";
}
}
```

CosLifeCycle::NoFactory exceptions: When you get a CosLifeCycle::NoFactory exception, there are a couple of things you may want to check to determine why. First the factory finder looks only within its scope for a factory that supports the type of object specified in the factory key. If this happens you should expand the scope of location boundary to obtain a factory finder with a broader scope of location and try the request again.

Also, to find the factory, the principal interface name (the *object interface*) specified to the factory finder must match exactly the name of the interface registered with the factory. If you cannot find the factory that you expect, check the spelling and case of the principal interface you specified in the factory key. You can use the System Management facilities to examine the interface name registered with the corresponding Home if you are uncertain of the exact spelling.

Managed vs. local only implementations

In the previous sections on lifecycle object interfaces, we have been examining the interfaces without much consideration for the implementation of these interfaces. We are now ready to delve into the implementation aspects.

The LifeCycle Service provides both managed and local only implementations of the FactoryFinder, SingleLocation and OrderedLocation interfaces. In the Managed objects and local only objects and Creating and obtaining lifecycle objects sections, we discussed the nature of these two types of objects, both in general and also how they apply to lifecycle. A summary of the important points regarding these two types of objects is appropriate at this point:

Managed Objects	Local Only Objects
Accessible by remote processes	Only accessible within a process
Can be bound into the system name space	Cannot be bound into the system name space
Usually persistent	Always transient
Can only exist in server processes	Can exist in server or client processes
Need reference to a home to create, which implies you also had a reference to a factory finder in order to find the home.	Can be created independent of other objects, using native language capabilities
Suited for long term and/or shared use	Suited for one time only use

In the subsequent sections, we will be looking at how to obtain references to lifecycle objects. From a client program perspective, there are two ways to do this:

- Resolve to an existing managed object bound into the name space
- Create a local only object directly in C++

There are three ways that managed lifecycle objects get bound into the name space for use by client programs.

- Created by Component Broker as one of the Default LifeCycle Objects
- Created by one of your programs
- Configured using the Component Broker System Manager

Of course, you could have a program create a managed lifecycle object and then use it immediately rather than bind it into the name space for later use. However, you would be more likely to use a local only lifecycle object if you were going to simply create it for immediate use.

In the following sections we will look at how you access lifecycle objects by examining Local only creation mechanisms and Accessing managed lifecycle objects from the name space. Then we will look at the mechanisms to create

and configure managed lifecycle objects and bind them into the name space in the sections Creating managed lifecycle objects and Configuring managed lifecycle objects.

Local only creation mechanisms

The local only creation mechanisms can be used from your VisualAge C++ code, be it client application or business object implementation. There is currently no implementation of local only lifecycle objects in Java, which means you cannot use them from your Java client code. However, they can be created and used from Java business objects using the normal Java local only creation mechanisms which support cross language bindings.

When creating a local only factory finder for use by your program, you typically just need to create the factory finder. You can initialize it with scope of location information during creation, and the creation of the factory finder will also create the appropriate location object for you. Alternatively, you can explicitly create the location object(s) needed and then use them when creating the factory finder. In the following sections we discuss the specifics of creating local only SingleLocation, OrderedLocation and FactoryFinder objects.

Local only SingleLocation creation: There are three static functions for creating local only SingleLocation objects. All of these are overloaded C++ static class functions and can only be invoked locally from a VisualAge C++ program or from a Java Business Object implementation.

The following are the static functions which enable three different ways to create a local only SingleLocation object:

_create()

The function `ILifeCycleLocalObjectImpl::SingleLocation::_create` creates a `SingleLocation` with the default scope of boundary values as shown in the following table:

Cell	Workgroup	Host	Server	Container	Home
*LOCAL	*LOCAL	*LOCAL	*ANY	*ANY	*ANY

_create(Scope)

The function `ILifeCycleLocalObjectImpl::SingleLocation::_create(Scope)` creates a `SingleLocation` with a scope of location set to the boundary values supplied in the `Scope` structure.

_create(ScopeString)

The function `ILifeCycleLocalObjectImpl::SingleLocation::_create(ScopeString)` creates a `SingleLocation` with a scope of location values supplied in the `ScopeString`.

The following example creates a new local only `SingleLocation` object using the `_create(ScopeString)` function and sets it with a server scope: of `myServer` on host `abc.austin.ibm.com`. Take special note that the name of the host is placed into single quotes because it contains the dot character, which is also a separator in the string syntax. Also note that we do not specify all six of the scope of location boundary values, letting the non specified values take on their respective defaults.

```
IExtendedLifecycle::SingleLocation_var myNewSL;  
myNewSL = ILifeCycleLocalObjectImpl::SingleLocation::_create(  
    "*IGNORE.workgroup/'abc.austin.ibm.com'.host/myServer.server"
```

Local only OrderedLocation creation: There are three static functions for creating local only `OrderedLocation` objects. All of these are overloaded C++ static class functions and can only be invoked locally from a VisualAge C++ program or from a Java Business Object implementation.

The following are the static functions which enable three different ways to create a local only `OrderedLocation` object:

`_create(Scopes)`

The function

`ILifeCycleLocalObjectImpl::OrderedLocation::_create(Scopes)` creates an `OrderedLocation` object with a scope of location boundary set to the first scope structure supplied in the sequence of scopes, followed by the second scope structure, and so on.

`_create(ScopesStrings)`

The function

`ILifeCycleLocalObjectImpl::OrderedLocation::_create(ScopeStrings)` creates an `OrderedLocation` object with a scope of location boundary set to the first scope supplied in the `ScopeStrings` sequence, followed by the second scope, and so on.

`_create(SequenceOfLocations)`

The function

`LifeCycleLocalObjectImpl::OrderedLocation::_create(SequenceOfLocations)` creates an `OrderedLocation` whose scope of location is defined by the sequence of locations passed on input. The locations in the sequence can be either local only or managed. They can be `SingleLocations`, other `OrderedLocations`, or any implementation of the `IExtendedLifecycle::Location` abstract interface. This produces an `OrderedLocation` object with a scope of location boundary set to the scope of location of the first `Location` in the `SequenceOfLocations`, followed by the scope of location of the second, and so on.

The example below creates a new local only `OrderedLocation` object using the `_create(ScopeStrings)` function and sets it with the host scope for host `xyz.rchland.ibm.com`, followed by the host scope for `abc.rchland.ibm.com`. As

in the previous example, the host names must be enclosed in single quotes because they contain the dot character, and non specified scope of location boundary values will be allowed to default.

```
IExtendedLifeCycle::OrderedLocation_var myNewOL;  
IExtendedLifeCycle::OrderedScopeStrings oss(2); // 2 scope strings  
  
oss[0] = CORBA::string_dup(  
    "*IGNORE.workgroup/'zyx.rchland.ibm.com'.host");  
oss[1] = CORBA::string_dup(  
    "*IGNORE.workgroup/'abc.rchland.ibm.com'.host");  
oss.length(2);  
myNewOL = ILifeCycleLocalObjectImpl::OrderedLocation::_create(oss);
```

Local only FactoryFinder creation: There are seven static functions for creating local only FactoryFinder objects. All of these are overloaded C++ static class functions and can only be invoked locally from a VisualAge C++ program or from a Java Business Object implementation. As we have previously mentioned, the creation mechanisms for local only factory finders provide ways for you to create the factory finder and have it implicitly create the contained location object for you. You will notice that of the seven creation methods, three are similar to the SingleLocation creation methods, three are similar to the OrderedLocation methods and the seventh is unique, taking a location object an input argument.

You can create a local only factory finder in any one of the following ways:

_create()

The function `ILifeCycleLocalObjectImpl::FactoryFinder::_create` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::SingleLocation::_create()` function.

_create(Location)

The function

`ILifeCycleLocalObjectImpl::FactoryFinder::_create(Location)` creates a factory finder which contains the Location object passed as input to the function.

_create(Scope)

The function `ILifeCycleLocalObjectImpl::FactoryFinder::_create(Scope)` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::SingleLocation::_create(Scope)` function.

_create(ScopeString)

The function

`ILifeCycleLocalObjectImpl::FactoryFinder::_create(ScopeString)` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::SingleLocation::_create(ScopeString)` function.

`_create(SequenceOfLocation)`

The function

`ILifeCycleLocalObjectImpl::FactoryFinder::_create(SequenceOfLocation)` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::OrderedLocation::_create(SequenceOfLocation)` function.

`_create(OrderedScopes)`

The function

`ILifeCycleLocalObjectImpl::FactoryFinder::_create(OrderedScopes)` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::OrderedLocation::_create(OrderedScopes)` function.

`_create(OrderedScopeStrings)`

The function

`ILifeCycleLocalObjectImpl::FactoryFinder::_create(OrderedScopeStrings)` creates a factory finder whose location object is created using the `ILifeCycleLocalObjectImpl::OrderedLocation::_create(OrderedScopeStrings)` function.

The example below shows the creation of a factory finder that looks first in the server Preferred and then in the server Backup, both of which are on host myHost. This is the `_create(OrderedScopeStrings)` creation method.

```
ExtendedLifeCycle::FactoryFinder_var myNewFF;
ExtendedLifeCycle::OrderedScopeStrings oss(2); // 2 scope strings

oss[0] = CORBA::string_dup(
    "*IGNORE.workgroup/myHost.host/Preferred.server");
oss[1] = CORBA::string_dup(
    "*IGNORE.workgroup/myHost.host/Backup.server");
oss.length(2);
myNewFF = ILifeCycleLocalObjectImpl::FactoryFinder::_create(oss);
```

Accessing managed lifecycle objects from the name space

The typical programming model for obtaining a factory finder is to obtain one from the system name space by doing a resolve operation. In Default lifecycle objects we learned about the many default lifecycle objects created by Component Broker and where they are bound into the name space. In the subsequent sections we will see how you can create and configure managed lifecycle objects and have them bound into the name space.

There are fixed locations in the name space where lifecycle objects are bound. They are:

```
/host/resources/factory-finders
/workgroup/resources/factory-finders
/cell/resources/factory-finders
```

```
/host/resources/locations
/workgroup/resources/locations
/cell/resources/locations
```

When wanting to obtain a factory finder or location object from the name space:

- It must already exist and be bound into the name space.
- You need to know the name by which it was bound into the name space.
- You need to know which branch of the system name space it was bound into.

It is highly likely that your programs will want to obtain a factory finder from the name space. The following example takes a look at obtaining the default server scope factory finder for myServer from the host branch of the system name space. We use the `CBSeriesGlobal::nameService()` operation to get a reference to the root naming context for the host we have bootstrapped to. The appropriate error detection for a `NotFound` exception and for the returned object not being a factory finder has been included.

```
// Declare the variables
IExtendedLifeCycle::FactoryFinder_var myFF;
CORBA::Object_var ffAsObject;

// Get object from name space and narrow to factory finder
try
{
    ffAsObject = CBSeriesGlobal::nameService()->resolve_with_string(
        "host/resources/factory-finders/myServer-server-scope");
    myFF = IExtendedLifeCycle::FactoryFinder::_narrow(ffAsObject);
}
catch(CosNaming::NamingContext::NotFound &e)
{
    myFF = IExtendedLifeCycle::FactoryFinder::_nil();
}

// Test if we were successful
if (CORBA::is_nil(myFF))
    cout << "failed";
else
    cout << "success";
```

Creating managed lifecycle objects

When the default lifecycle objects do not provide the scopes of location needed by some or all of your applications, you will have to either create (with a program) or configure (using the System Manager) them into your system. This section looks at how you would write code in a program to create managed lifecycle objects. The creation of managed lifecycle objects would normally be considered to be part of setting up your environment, so

typically this type of code would appear in code that you would run once as part of setting up your environment or as a step in installing an application.

There are some common elements to creating managed lifecycle objects. To begin with, they are created in a similar manner as all managed objects are created. Your code will have to obtain a reference to a factory finder, use that factory finder to find an instance of the home for the type of lifecycle object you want to create, and then invoke some method on that home to create the object.

All of the lifecycle objects have implementations of specialized homes which provide creation methods other than the standard `createFromPrimaryKeyString` supported by the `IManagedClient::IHome` interface. These creation methods allow you to create lifecycle objects without having to create and initialize a primary key object as you often do when creating business objects. In addition, they allow you to pass parameters used in the initialization of the newly created object, similar to how the `_create` functions take parameters when creating local only lifecycle objects. However, unlike the local only `_create` methods, creation of a managed factory finder will not implicitly create the corresponding location. You always must obtain a reference to or create the location object used to initialize the factory finder. This location object must be a managed object itself; (that is, you cannot initialize a managed factory finder with a local only location object).

All managed lifecycle objects have the ability to bind themselves into the system name space. The creation methods on the specialized homes allow you to pass in a "relative name" and three boolean indicators of where the object should bind itself into the system name space. This means that the binding of the lifecycle object you create can occur as a side effect to creation rather than your having to explicitly do it after creation. For example, if you were to create a factory finder with a relative name of `myFF` and indicated that you wanted it bound into the workgroup tree (but not the host and cell trees), the new factory finder would bind itself into: `/workgroup/resources/factory-finders/myFF`. The following sections look at the specifics of creating each of types of lifecycle objects. To be complete examples, they would need to show the following steps:

1. Obtaining a reference to a factory finder from the name space
2. Using the factory finder to obtain a reference to the specialized home
3. Obtain a reference to other objects used to initialize the new object (if needed)
4. Make the call on the home to create the object

The following examples will only show complete code for step four. For each of the first three steps, a reference to the appropriate example will be given along with any additional type specific information that would be needed.

Using the SingleLocationHome interface

This example shows you how to create a SingleLocation object using a home supporting the SingleLocationHome interface. The steps which are not shown in detail are:

Obtain a factory finder

Follow the example in Accessing managed lifecycle objects from the name space, using a name path in the resolve_with_string that will get you the host scoped factory finder for the host on which you want to create the SingleLocation (for example, /cell/hosts/desiredHost/resources/factory-finders/host-scope). If you have special requirements that it be created in a particular server, get the appropriate server scoped factory finder instead.

Find the home

Follow the examples in FactoryFinder interface. The value used for “object interface” should be ILifeCycleManagedClient::SingleLocation. The other qualifiers should not be needed. The object returned needs to be narrowed to IExtendedLifeCycle::SingleLocationHome.

Obtain references to other objects

Not needed for SingleLocation creation.

Now we are ready to create the SingleLocation. There are two methods provided by the SingleLocationHome, which are createWithScope() and createWithScopeString(). This example uses the createWithScopeString() method. See “Defining scope of location” on page 128 for valid values and Scope Structures and Strings for valid syntax of the string. Similar to the local only example, we will initialize the single location to have a server scope of myServer on host abc.austin.ibm.com. We will not ask the new SingleLocation object to register itself in the name space.

```
// Declare the variables (SLHome assumed initialized)
IExtendedLifeCycle::SingleLocationHome_var SLHome;
IExtendedLifeCycle::SingleLocation_var myNewSL;

// Create the Single Location
myNewSL = SLHome->createWithScopeString(
    "*IGNORE.workgroup/'abc.austin.ibm.com'.host/myServer.server",
    "", // no relative name for binding to name space
    0, // not bound into the cell tree
    0, // not bound into the host tree
    0); // not bound into the workgroup tree
// Test if we were successful
if (CORBA::is_nil(myNewSL))
    cout << "failed";
else
    cout << "success";
```

Using the OrderedLocationHome interface

This example shows you how to create an OrderedLocation object using a home supporting the OrderedLocationHome interface. The steps which are not shown in detail are:

Obtain a factory finder

Follow the example in Accessing managed lifecycle objects from the name space, using a name path in the resolve_with_string that will get you the host scoped factory finder for the host on which you want to create the OrderedLocation. (for example, /cell/hosts/desiredHost/resources/factory-finders/host-scope). If you have special requirements that it be created in a particular server, get the appropriate server scoped factory finder instead.

Find the home

Follow the examples in FactoryFinder interface. The value used for "object interface" should be ILifeCycleManagedClient::OrderedLocation. The other qualifiers should not be needed. The object returned needs to be narrowed to IExtendedLifeCycle::OrderedLocationHome.

Obtain references to other objects

Some of the operations on the OrderedLocationHome require you to pass in a sequence of locations. Follow the directions in Accessing managed lifecycle objects from the name space or Using the SingleLocationHome interface to obtain or create references to the appropriate managed location objects. In addition, if you have your own managed object implementation of the Location interface, references to instances of that type could be used as well.

Now we are ready to create the SingleLocation. There are three methods provided by the OrderedLocationHome. The first two, createWithScopes and createWithScopeStrings, allow you to pass in a sequence of Scope structures or sequence of ScopeStrings which will implicitly result in the creation of managed SingleLocation objects to be contained in the OrderedLocation object. The third method is createWithLocations which takes a sequence of Location objects. The location objects in the sequence must be managed objects.

The following example uses the createWithLocations method. The OrderedLocation will be initialized with two server scope single locations, thus creating an object whose scope of location is ServerA followed by ServerB. The example assumes the variables holding references to these have already been initialize. We will not ask the new OrderedLocation object to register itself in the name space.

```
// Declare the variables
IExtendedLifeCycle::SequenceOfLocations seqOfLoc(2); // two locations
IExtendedLifeCycle::SingleLocation_var SLServerA; // already initialized
```

```

IExtendedLifeCycle::SingleLocation_var SLServerB; // already initialized
IExtendedLifeCycle::OrderedLocationHome_var OLHome; // already initialized
IExtendedLifeCycle::OrderedLocation_var myNewOL;

// Initialize the input parameter
seqOfLoc[0] = IExtendedLifeCycle::SingleLocation::_duplicate(SLServerA);
seqOfLoc[1] = IExtendedLifeCycle::SingleLocation::_duplicate(SLServerB);
seqOfLoc.length(2);

// Create the Ordered Location
myNewOL = OLHome->createWithLocations(
seqOfLoc,
"", // no relative name for binding to name space
0, // not bound into the cell tree
0, // not bound into the host tree
0); // not bound into the workgroup tree

// Test if we were successful
if (CORBA::is_nil(myNewOL))
cout << "failed";
else
cout << "success";

```

Using the FactoryFinderHome interface

This example shows you how to create a FactoryFinder object using a home supporting the FactoryFinderHome interface. The steps which are not shown in detail are:

Obtain a factory finder

Follow the example in Accessing managed lifecycle objects from the name space, using a name path in the resolve_with_string that will get you the host scoped factory finder for the host on which you want to create the FactoryFinder. (for example, /cell/hosts/desiredHost/resources/factory-finders/host-scope). If you have special requirements that it be created in a particular server, get the appropriate server scoped factory finder instead.

Find the home

Follow the examples in FactoryFinder interface. The value used for “object interface” should be ILifeCycleManagedClient::FactoryFinder. The other qualifiers should not be needed. The object returned needs to be narrowed to IExtendedLifeCycle::FactoryFinderHome.

Obtain references to other objects

You must have a reference to a location object. Follow the directions in Accessing managed lifecycle objects from the name space or Using the SingleLocationHome interface or Using the OrderedLocationHome interface to obtain or create a reference to an appropriate managed

location object. Alternatively, if you have your own managed object implementation of the Location interface, a reference to an instance of that type could be used as well.

Now we are ready to create the FactoryFinder. There is just one method provided by the FactoryFinderHome, createWithLocation. In the example, the FactoryFinder will be created with a reference to an existing OrderedLocation object to which we already have a reference. We will ask the new FactoryFinder object to register itself in the name space at /cell/resources/factory-finders/myFF1 and also at /host/resources/factory-finders/myFF1.

```
// Declare the variables
IExtendedLifeCycle::OrderedLocation_var existingOL;// already initialized
IExtendedLifeCycle::FactoryFinderHome_var FFHome; // already initialized
IExtendedLifeCycle::FactoryFinder_var myNewFF;

// Create the Factory Finder
myNewFF = FFHome->createWithLocation(
existingOL,
"myFF1", // relative name for binding to name space
1, // bound into the cell tree
1, // bound into the host tree
0); // not bound into the workgroup tree

// Test if we were successful
if (CORBA::is_nil(myNewFF))
cout << "failed";
else
cout << "success";
```

Configuring managed lifecycle objects

Another approach to establishing managed lifecycle objects in your environment is to configure them using the Component Broker System Manager. When you use the System Manager to configure a managed FactoryFinder object, the location object which it contains must also be configured by the System Manager. Also, any location objects contained in a System Manager configured OrderedLocation object must also be configured using the System Manager. The implication to this is that you cannot use any of the default location objects or user developed implementations of the Location interface for a factory finder which is being configured with the System Manager.

For complete information about how to use the System Manager to configure lifecycle objects, See the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*.

Lifecycle example

This section takes a look at an example program which illustrates principles of the LifeCycle Service as well as syntax of the operations. The example is a fully functional client application that creates a managed factory finder which is bound into the name space. We will start by explaining the scenario of what the enterprise is trying to accomplish with this factory finder. Although the example may be somewhat contrived, it illustrates many of the points important to the LifeCycle Service. Hopefully this illustration will lead you to practical uses of lifecycle within your own enterprise.

This factory finder is to be used by client programs of a Customer Support Administration application, which is composed of a specific set of interrelated business objects (BOs) in individual applications generated by Object Builder. The applications defining these BOs are deployed in various servers within the Component Broker environment. They want the factory finder to be available to all clients without regard to where these clients are running and without requiring configuration knowledge on the part of the clients. Therefore, it will be bound into the name space in the cell branch of the System name space using the name of the application, specifically at `/cell/resources/factory-finders/customerSupportAdministration`.

The strategy for deployment of these business object applications is as follows:

- There will be one server within the configuration that will house the most critical BOs that have a high degree of interaction (that is, the BOs need to be “near” each other) This server will be called the “preferredServer”. However, which BOs will be in here might hang over time as the application performance is evaluated. This server could be deployed on any of a number of different hosts in the environment.
- There is a host whose servers house many of the transient BOs (sometimes called Application Objects, or AOs). However, there will be times when this host needs to be removed from the configuration without interrupting the Consumer Support Administration application, and therefore it cannot be absolutely depended upon. We will call this the “optionalHost”.
- There is another host which will always be in the configuration. It has servers which house some of the more important BOs which are not required to be in the preferredServer. It may also house some of the same AOs from the optionalHost. We will call this the “requiredHost”. There are some number of other hosts whose servers house additional BOs and AOs for the application.

So now we will look at how to configure the customerSupportAdministration factory finder to take into account these objectives. Let’s look at the pieces needed to make up this factory finder:

- These requirements are outside the scope of any of the default lifecycle objects, so we know we need a created factory finder configured with a created location object.
- The location involves several different scopes of location, therefore the created location object needs to be an OrderedLocation object.
- The first scope of location, the “preferredServer”, can be defined using a default server scoped SingleLocation, that is the preferredServer-server-scope location object.
- The second scope of location, the “optionalHost” might be satisfied by the default host scoped SingleLocation. However, we said that the host might not be in the configuration at times, and the default host-scope location object for it would not be available at those times. Therefore, we will need to create a host scoped SingleLocation for the optionalHost. This location object will enable things to be found on this host when it is in the configuration, but it’s presence will not cause a problem with factory finding when the host is not configured.
- The third scope of location, the “requiredHost” can be satisfied by the default host scoped SingleLocation, that is the host-scope location object for requiredHost. However, we see that the fourth scope of location is “some number of other hosts”. Therefore, we probably want to make this the default host-scope-widened location object for requiredHost, thus taking in the rest of the workgroup and cell (we will assume that all the homes for these BOs and AOs will be visible in the workgroup and cell branches of the lifecycle repository).

To summarize, we need a factory finder which is configured as follows (indentation indicates containment):

```

created FactoryFinder (Consumer Support Administration)
  created OrderedLocation (Consumer Support Administration)
    default SingleLocation (preferredServer-server-scope)
    created SingleLocation (host scoped for optionalHost)
      default OrderedLocation (host-scope-widened for requiredHost)
        default SingleLocation (host-scope for requiredHost)
        default SingleLocation (workgroup-scope)
        default SingleLocation (cell-scope)

```

Our example program builds this factory finder, configured as above and binds it into /cell/resources/factory-finders/customerSupportAdministration. Each section of the example has a Step number for reference purposes. We will explain each of the steps of the program here:

Step 1 - Demonstrates creation of a local only factory finder

We will need to create lifecycle objects in our example, and therefore need to find the homes for lifecycle objects. We will make an

assumption that this example program will be run with its bootstrap host specifying the machine where we want this factory finder to be created. Therefore, we will just need the host-scope factory finder for our bootstrap (local) host. We choose to do this with a local only factory finder created with all the default scope of location boundary values (that is local host scope). You will notice that we do this using the `_create()` method with no parameters. Alternatively, we could have done a resolve to the name space to get a reference to the default host-scope factory finder.

Step 2 - Demonstrates factory finding operations

In our example, we need to create a `FactoryFinder`, a `SingleLocation` and an `OrderedLocation`. In this step, we use the local only factory finder to find the `FactoryFinderHome`, `SingleLocationHome` and `OrderedLocationHome`. Note that the code would be no different in this step if the factory finder was a reference to a managed factory finder.

Step 3 - Demonstrates obtaining lifecycle objects from the name space

We will need references to a couple of the default location objects, specifically the preferredServer-server-scope location and the host-scope location for the `requiredHost`. We do this by doing a `resolve_with_string` to the name space. Although this example shows getting location objects, the same mechanism would be used to get factory finders from the name space, with the only difference being the name path specified to `resolve_with_string`.

Step 4 - Demonstrates the creation of a SingleLocation object

In this step we create the single location for the `optionalHost`. This is done using the `createWithScopeString` call on the `SingleLocationHome`. We do not bind the new object into the name space.

Step 5 - Demonstrates the creation of an OrderedLocation object

In this step we create the ordered location for the Consumer Support Administration application. This is done using the `createWithLocations` call on the `OrderedLocationHome`. The sequence of locations we initialize it with contains a default `SingleLocation`, a created `SingleLocation` and a default `OrderedLocation`. We do not bind the new object into the name space.

Step 6 - Demonstrates the creation of a FactoryFinder object

In this step we create the factory finder for the Consumer Support Administration application. This is done using the `createWithLocation` call on the `FactoryFinderHome`. We request that the new factory finder bind itself into the cell branch of the name space using the name `consumerSupportAdministration` (that is at `/cell/resources/factory-finders/consumerSupportAdministration`).

Before looking at the example, note a couple of other items. We provide a minimal level of error checking. There is a *try catch* block around the entire program which will catch and report any unexpected exception returned from anywhere within the try block. Then, when we might expect specific exceptions (for example, `CosLifecycle::NoFactory` from a `find_factory_from_string` call) we also handle these with a *try catch* block. Lastly, the `_narrow` operation is defined to return a nil object reference if it fails, so we check for that also. This could be considered a minimally acceptable level of error handling. As this is intended to be a run once program, it is probably sufficient.

The example follows:

```
#include <CBSeriesGlobal.hh>
#include <ILifeCycleManagedClient.hh>
#include <ILifeCycleLocalObjectImpl.hh>

int main(int argc, char * argv[])
{
    //-----
    // Declare variables used in more than one step
    //-----
    IExtendedLifecycle::FactoryFinder_var localHostFF;

    IExtendedLifecycle::SingleLocationHome_var SLHome;
    IExtendedLifecycle::OrderedLocationHome_var OLHome;
    IExtendedLifecycle::FactoryFinderHome_var FFHome;

    IExtendedLifecycle::Location_var          defaultSL;
    IExtendedLifecycle::Location_var          defaultOL;

    IExtendedLifecycle::SingleLocation_var    createdSL;
    IExtendedLifecycle::OrderedLocation_var    createdOL;

    IExtendedLifecycle::FactoryFinder_var     createdFF;

    //-----
    // Use try/catch to handle unexpected errors
    //-----
    try
    {
        //-----
        // STEP 1 - create a local only FactoryFinder
        //-----
        cout << "Step 1 - create a local only FactoryFinder" << endl;
        localHostFF = ILifeCycleLocalObjectImpl::FactoryFinder::_create();

        //-----
        // STEP 2 - get homes of lifecycle objects by factory finding
        //-----
        cout << "Step 2 - get homes of lifecycle objects by factory finding"
            << endl;
    }
}
```

```

CosLifeCycle::Factory_var homeAsFactory;

// Find all three homes and narrow to correct interfaces
try
{
    homeAsFactory = localHostFF->find_factory_from_string(
        "ILifeCycleManagedClient::SingleLocation.object interface");
    SLHome =
        IExtendedLifeCycle::SingleLocationHome::_narrow(homeAsFactory);

    homeAsFactory = localHostFF->find_factory_from_string(
        "ILifeCycleManagedClient::OrderedLocation.object interface");
    OLHome = IExtendedLifeCycle::OrderedLocationHome::_narrow(
        homeAsFactory);

    homeAsFactory = localHostFF->find_factory_from_string(
        "ILifeCycleManagedClient::FactoryFinder.object interface");
    FFHome = IExtendedLifeCycle::FactoryFinderHome::_narrow(
        homeAsFactory);
}
catch(CosLifeCycle::NoFactory &e)
{
    cout << "could not find one of the lifecycle object homes" << endl;
    exit(-1);
}

// Now check to make sure all the narrows worked
if (CORBA::is_nil(SLHome) ||
    CORBA::is_nil(OLHome) ||
    CORBA::is_nil(FFHome))
{
    cout << "could not narrow to lifecycle object homes" << endl;
    exit(-1);
}

//-----
// STEP 3 - get default Location objects by naming resolve
//-----
cout << "Step 3 - get default Location objects by naming resolve"
    << endl;
CORBA::Object_var locationAsObj;

// Get objects from name space and narrow to Location interface
try
{
    locationAsObj = CBSeriesGlobal::nameService()->resolve_with_string(
        "cell/resources/locations/preferedServer-server-scope");
    defaultSL = IExtendedLifeCycle::Location::_narrow(locationAsObj);

    locationAsObj = CBSeriesGlobal::nameService()->resolve_with_string(
        "cell/hosts/requiredHost/resources/locations/host-scope-widened");
    defaultOL = IExtendedLifeCycle::Location::_narrow(locationAsObj);
}
catch(CosNaming::NamingContext::NotFound &e)

```

```

    {
    cout << "one of the location objects not found in name space" << endl;
    exit(-1);
    }

// Now check to make sure the narrows worked
if (CORBA::is_nil(defaultSL) ||
    CORBA::is_nil(defaultOL))
    {
    cout << "could not narrow location objects from name space" << endl;
    exit(-1);
    }

//-----
// STEP 4 - create a SingleLocation object
//-----
cout << "Step 4 - create a SingleLocation object" << endl;

createdSL = SLHome->createWithScopeString(
    "*IGNORE.workgroup/optionalHost.host",
    "", // no relative name for binding to name space
    0, // not bound into the cell tree
    0, // not bound into the host tree
    0); // not bound into the workgroup tree

// Make sure it was created OK
if (CORBA::is_nil(createdSL))
    {
    cout << "creation of single location object failed" << endl;
    exit(-1);
    }

//-----
// STEP 5 - create an OrderedLocation object
//-----
cout << "Step 5 - create an OrderedLocation object" << endl;

IExtendedLifeCycle::SequenceOfLocations seqOfLoc(3);

// Initialize the input parameter
seqOfLoc[0] = IExtendedLifeCycle::Location::_duplicate(defaultSL);
seqOfLoc[1] = IExtendedLifeCycle::Location::_duplicate(createdSL);
seqOfLoc[2] = IExtendedLifeCycle::Location::_duplicate(defaultOL);
seqOfLoc.length(3);

// Create the Ordered Location
createdOL = OLHome->createWithLocations(
    seqOfLoc,
    "", // no relative name for binding to name space
    0, // not bound into the cell tree
    0, // not bound into the host tree
    0); // not bound into the workgroup tree

// Make sure it was created OK
if (CORBA::is_nil(createdOL))

```

```

        {
            cout << "creation of ordered location object failed" << endl;
            exit(-1);
        }

//-----
// STEP 6 - create the FactoryFinder object
//-----
cout << "Step 6 - create the FactoryFinder object" << endl;

createdFF = FFHome->createWithLocation(
    createdOL,
    "consumerSupportAdministration",
    // relative name to bind in name space
    1,    // bound into the cell tree
    0,    // not bound into the host tree
    0);  // not bound into the workgroup tree

// Make sure it was created OK
if (CORBA::is_nil(createdFF))
    {
        cout << "creation of factory finder object failed" << endl;
        exit(-1);
    }

//-----
// We had a problem somewhere, don't know where
//-----
}
catch(...)
{
    cout << "failed - unknown exception occurred" << endl;
    exit(-1);
}

//-----
// Finished - Exit with Success message
//-----
cout << "Success - FactoryFinder created & bound in name space"
    << endl;
exit(0);
}

```

Chapter 7. Naming Service

A Naming Service is the main mechanism for objects on the ORB to locate other objects by name. A name is a recognizable value that identifies an object. The Naming Service maps names to object references.

A name-to-object association is called a *name binding*. A *naming context* is a namespace in which the object name is unique. Every object has a unique reference ID.

You can optionally associate one or more names with an object reference. You always define a name relative to its naming context.

The Naming Service lets you create naming hierarchies so you can easily locate objects. Clients can navigate through different naming context trees in search of objects they want.

How you decide to implement a Naming Service depends on how you plan to use the service in conjunction with other services to locate objects. A Naming Service can be used as the backbone of an enterprise-wide filing system to construct large naming graphs where Naming Contexts model “directories” or “folders” and other names identify “document” or “file” kinds of objects. A Naming Service can also be used in a more limited role and have less sophisticated implementation, where naming contexts represent the types and locations of services that are available in the system.

You can implement a Naming Service to be application specific, or to be based on a variety of naming systems currently available on system platforms.

Naming objects in the distributed object system

Business objects and other resources can be assigned a name in the system name space. This is called binding an object. After you bind an object with a name you can find that object or resolve it by its name.

Binding an object has several advantages. Most notably it provides end-users and programmers a way to talk about particular objects. Names essentially form identities for objects. These names can be passed between people and their programs either through user interfaces or in program code.

The Naming Service has these primary uses for locating:

- System resources, including collections of business objects.

- Business objects grouped in application naming contexts.

In both instances, the Naming Service is used to form an enterprise-wide (or at least workgroup-wide) name space from which named objects can be reliably found.

A fundamental concept in the Component Broker Naming Service is that the entire name space is composed of objects bound in one or more naming contexts. Each naming context is itself an object and can be bound to other naming contexts to form a tree. In this fashion, the name space can be structured into a hierarchy of names. Any object then has both a relative name, their name within a naming context, and a compound name, the name path representing a particular traversal through the name hierarchy from a higher-level naming context to that object.

To further exemplify the placement of objects and naming contexts in the name tree consider the name tree depicted in the diagram that follows. This name tree is composed of a hierarchy of naming contexts and objects (naming contexts contain other naming contexts and/or objects). Notice that only the object bindings have names and not the objects themselves. Thus an object name is by its very nature contextual. The name of an object only exists within a given context, and then only if the object is bound to that context. The context at the top of the tree does not have a name at all (by virtue of not being bound to any other context). This document refers to this as the root of the tree.

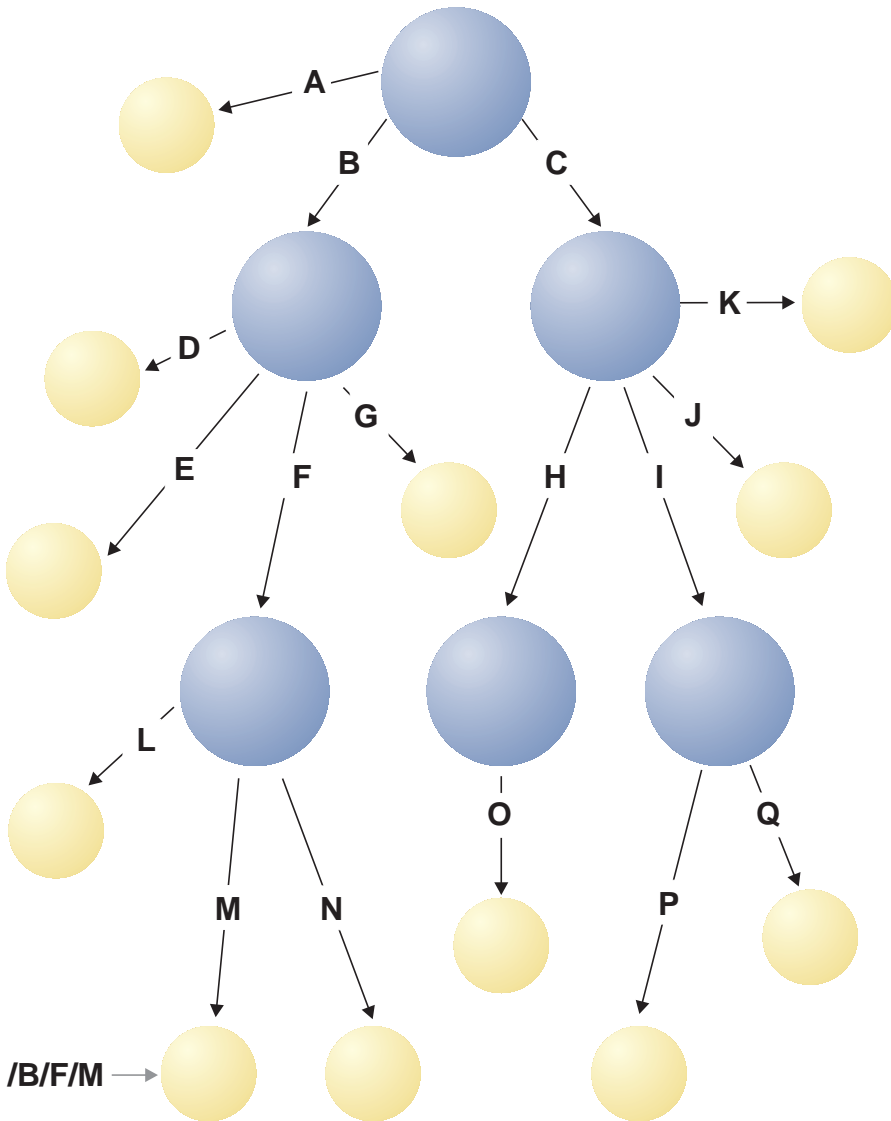


Figure 4. General name tree organization

Name trees are useful for managing large numbers of named-objects. You can partition the name space into categories of related objects that are meaningful to you or your application. Such categorization makes it easier to find an object because you only have to remember the category and the name of the object within that context.

A consequence of instituting name trees in this fashion is that the same object can be bound multiple times with different names or in different contexts.

There are potentially positive and negative aspects to this. An object can have different names for different applications. This might be useful for aliasing or transparent resource sharing. In particular, if an object falls naturally within more than one category it can be bound by the same or different name in all categories to which it pertains. On the other hand, being bound multiple times introduces graphs and potential cycles in the name space. This can make traversing the name space more complex.

System name space

The system name space is the predefined name space structure that is delivered and installed with Component Broker. This is useful for ensuring that objects can be bound and located by well known name paths. The system name space is depicted in the diagrams that follow.

graphic to be completed

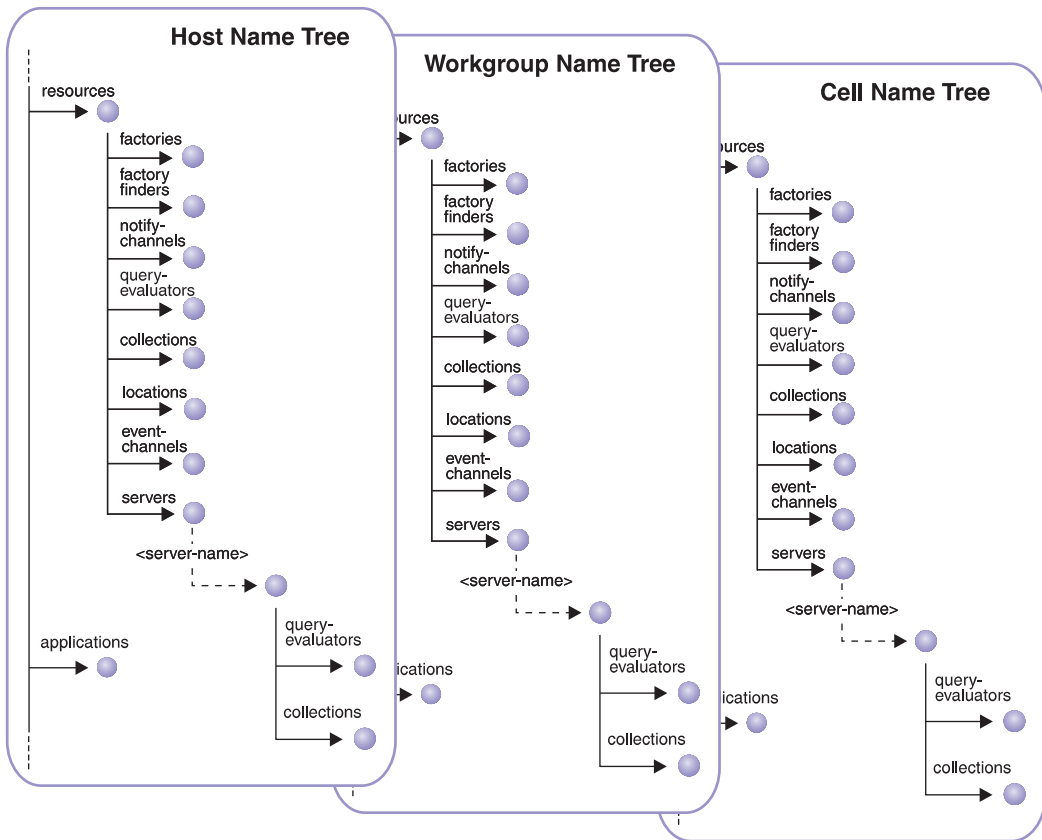


Figure 5. Name space (continued)

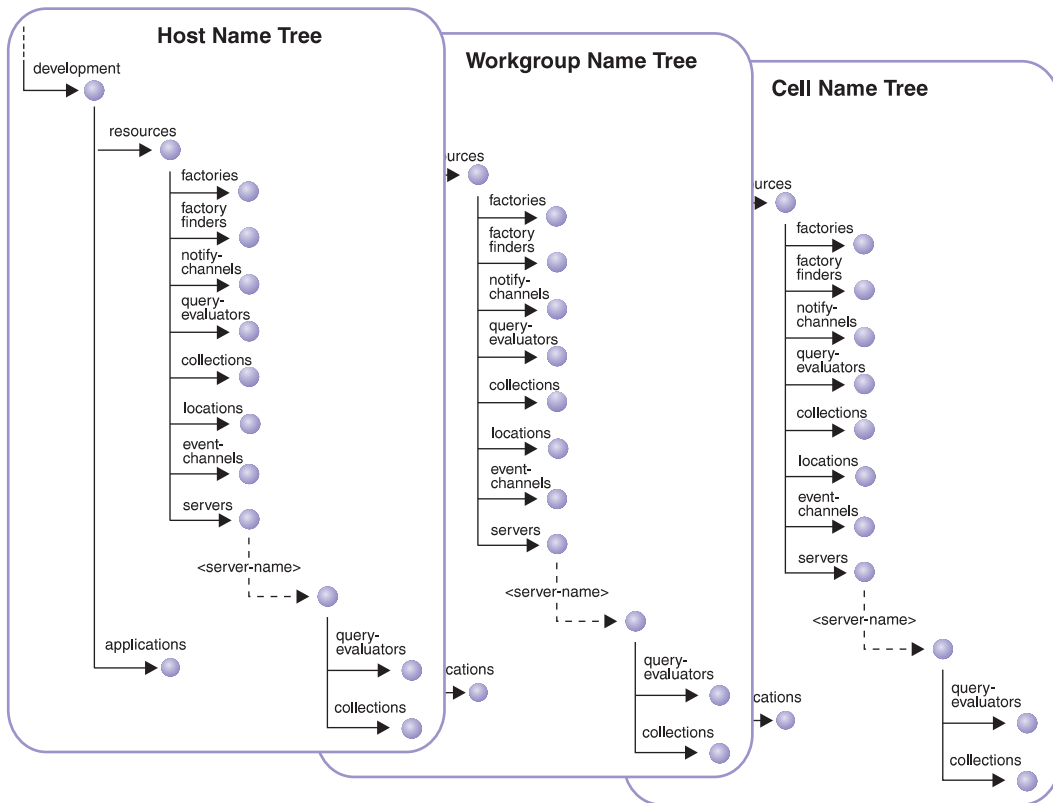


Figure 6. Name space (continued)

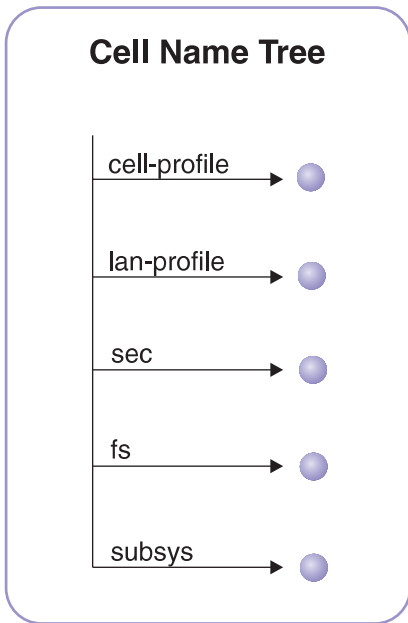


Figure 7. Name space (continued)

The name space is composed of three distinct parts, a *local name tree*, a *workgroup name tree*, and a *cell name tree*. In addition, the inter-domain root can be used to locate resources across cells. The system name space is modeled somewhat on the UNIX file system, that is every host is intended to contain a local name space in which a distributed name space can be mounted. The local name tree contains the local name space. The workgroup and cell name trees are distributed and shared name spaces. These other name trees are in effect *mounted* to the local name tree.

Choosing where to bind or find named objects in the system name space is a function of the type of the resource in question, and the visibility that should be attributed to the resource.

Visibility of named objects

Each portion of the system name space, the host name tree, workgroup name tree, and cell name tree, provides a different scope of visibility or sharing for named objects. Objects bound in the host name tree are specific to the host machine on which the host name tree resides. Objects bound in the workgroup name tree are specific to that workgroup. And likewise, objects bound in the cell name tree are specific to the cell. Conversely, these name trees also represent the scope over which resources bound in each respective tree can be shared.

Before binding an object in the system name space, you should first consider the extent to which you want that object shared. If you want to keep it relatively specific to your host machine, then bind it in the appropriate context under the host name tree. If you want it shared within your workgroup, then bind it in the workgroup name tree. And if you want it shared across your cell, then bind it in the cell name tree.

Likewise, when you go to resolve a named object, consider what scope you want to find it in. You may find different objects bound with the same name in different name trees. You may have to look at more than one name tree to find the named object. One strategy you may find useful is to always start by looking in the host name tree. If the object is not found there, then look in the workgroup name tree, and finally the cell name tree. In this way, you can find an object that is most specific to where your program is running.

Local and host name tree

An instance of the local name tree exists on every (server) host. In general this includes any node capable of being configured to contain server processes, including machines that are otherwise clients. Local name trees can be shared amongst a group of pure client machines, that is machines that are configured to not have any server processes. However, this latter approach may limit the ability for client machines to build and maintain private name spaces. Thus, if it is necessary for a client machine to have a private name space, this should be created under a pseudo-host root for that machine within the **workgroup/hosts** and/or **cell/hosts** contexts.

The sub-tree in the local name tree bound to **host**, while local to the host, is distinguished and can be made visible to other hosts. This is referred to as the host name tree.

Resources belonging to a host should be bound in the **host/resources** context of the local name tree. This context can be further organized by resource type or whatever is appropriate for an installation, according to the following rules:

- Factories should be bound in the **resources/factories** context.
- Factory finders should be bound in the **resources/factory-finders** context.
- Location objects should be bound in the **resources/locations** context.
- Event channels should be bound in the **resources/event-channels** context.
- Notification channels should be bound in the **resources/notify-channels** context.
- Query evaluators should be bound in the **resources/query-evaluators** context.
- Collections should be bound in the **resources/collections** context.

A naming context for each server in the host is bound in **resources/servers** by the server name. This can be used to bind resources that are specific to a particular server. Query evaluators are bound in **resources/servers/<server-name>/query-evaluators** for the server in which they exist. Collections are bound in **resources/server/<server-name>/collections** for the server in which they exist.

Prior to deployment, resources which are used in a development environment can be bound under **host/development/resources** context to keep them separate from production resources. The resources sub-tree is repeated under the **host/development/resources** context to assist to binding development resources there.

A naming context for each application installed on the host can be bound in **applications** by the application name. This can be used to bind resources that are specific to that particular application.

Every local name tree contains a binding to the inter-domain root at "...", a binding to the cell root at both ":@" and at **cell**, and to the workgroup root at **workgroup**. These can be used to navigate to other parts of the system name space.

Note that the local root context is not bound to any other naming context. This is also referred to as the absolute root for the host machine on which the local name tree resides. The *ORB::resolve_initial_references("NamingService")* operation returns the local root for the host machine. In Component Broker, *CBSeriesGlobal::nameService()* is the more commonly used mechanism to obtain the local root naming context.

Note in particular that the local root is not bound to any other name tree. Any object bound under the local root naming context and not under the host name tree (that is **host**) cannot be accessed from any other host through traversal of the name tree. For a remote machine to access the local root context, the bootstrapping mechanism must be used.

Workgroup name tree

A workgroup is a logical collection of hosts whose aggregation creates some administrative or operational synergy for the business. There is no concrete definition for a workgroup: typically it consists of a departmental or organizational unit of 5 to 200 hosts, although it is certainly not restricted to either of these boundaries.

One workgroup name tree exists per workgroup. Each host belongs to only one default workgroup bound to **workgroup** in the local name tree. Also, each host name tree in a workgroup is bound by host name in the **hosts** context of

the workgroup name tree. Thus any host can get to the host name tree of any other host in a workgroup. The **workgroup** binding of the local name trees and the **hosts** context of the workgroup name trees are not strictly synchronized, and so it is possible for the same host to appear in the **hosts** context of more than one workgroup. A host could only discover its complete containment relationship by examining each of the workgroup name trees within a cell.

In the case that a pure client host needs a host name tree that pertains to it, since by definition it does not have a host name tree of its own, can create one in the workgroup name tree. The host name tree should be bound with the host name in the hosts context of the workgroup name tree. This is referred to as a pseudo-host name tree as the host name tree really does not exist on the host machine, a pure client machine in this case.

Resources belonging to a workgroup should be bound in the **resources** context of the workgroup name tree. As with the **resources** context in the host name tree this context can be further organized by resource type or whatever is appropriate for an installation. Likewise, prior to deployment, resources that are used in a development environment can be bound in the **development/resources** context to keep them separate from production resources.

Cell name tree

A cell represents an administrative boundary for the name space. A cell contains one or more workgroups, thus it is a super-set of workgroups.

Note that there is no binding in the workgroup name tree to the workgroup cell. Even though only one workgroup name tree exists in a workgroup, it exists physically under one of the local name trees, and logically under the local name trees of all of the hosts in the workgroup. Thus, the cell for the workgroup can always be determined from the local name tree.

Nonetheless, for administrative and visibility purposes, workgroups are bound by workgroup name within the **workgroups** context of one or more cell name trees. Thus, a host can get to the workgroup name tree of any other workgroup in the cell to which they belong.

Similarly, the host name tree for each host in the cell is bound by host name in the **hosts** context of the cell name tree. Thus any host can get to the host name tree of any other host in the cell.

Resources belonging to a cell should be bound in the **resources** context of the cell name tree. As with the **resources** context in the host name tree this context can be further organized by resource type or whatever is appropriate for an installation.

Likewise, prior to deployment, resources which are used in a development environment can be bound in the **development/resources** context to keep them separate from production resources.

The remaining contexts of the cell name tree, that is, **cell-profile**, **lan-profile**, **sec**, **fs**, and **subsys**, match the contents of a standard DCE cell directory name space and can be left empty.

Navigation in the system name space

A host name tree exists on every server host machine. Typically one workgroup name tree exists for each business group. This could be a project area, or product, or related business activity. And a cell name tree exists for each major business organization over which administration of information systems is shared, for example, a site in a multi-site enterprise.

The name trees are loosely related in a hierarchy, although this is not necessarily a strict hierarchy, and are interconnected by various bindings. For a given host, the default cell is bound into the host's root context at the both **cell** and at **":."**. The default workgroup is bound into the host's root context at **workgroup**. Each host in a workgroup is bound by its host name in the **hosts** context of the workgroup name tree. Also, each host in a cell is bound by its host name in the **hosts** context of the cell name tree. Similarly, each workgroup in a cell is bound by its workgroup name in the **workgroups** context of the cell name tree.

A program can resolve to an object on any host in any workgroup. For instance, consider the case where you have two hosts, "host-A" and "host-B", in the same workgroup. In addition, assume you have object "object-O" bound in the **applications/LifeInsurance/Claim** context on host-B. If a program on host-A wants to find object-O on host-B it can resolve to it from its local root context with the path

```
cell/workgroup/hosts/host-B/application/LifeInsurance/Claim/object-O
```

As another example, consider the case where there is a third host, "host-C", which is in a different workgroup, "workgroup-X". Also assume you have an object, "object-P", also in the **applications/LifeInsurance/Claim** context on host-C. You can navigate from the local root context on host-A to object-P with the path

```
cell/workgroups/workgroup-X/hosts/host-C/application/LifeInsurance/  
Claim/object-P
```

This could have also been done with the path

```
cell/hosts/host-C/application/LifeInsurance/Claim/object-P
```

Because host-C is also a member of the same cell.

Integration of system name spaces

In the current implementation, the system name space as built by Component Broker has only one cell and does not provide an inter-domain root as shown in Figure 5 on page 185. As a result, the name space enables you to traverse only through the cell, workgroups, and hosts defined and managed by a single System Manager. This is referred to as your Component Broker network.

It is possible to integrate the name space of your Component Broker network with the name spaces of other Component Broker networks, or even with non-Component Broker name spaces. One way to do this is for your program to obtain an object reference to a name context in another name space (for example, using a stringified object reference) and bind it into a name context within your Component Broker network. However, the Component Broker System Manager provides a mechanism to do this as a configuration option rather than your having to provide a program to accomplish the binding between different name spaces. This is done through an object called a Remote Name Context Binding.

Details of how to configure a Remote Name Context Binding are provided by the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*. You are required to provide:

- IP address and port used to bootstrap to the remote host
- The name path to a remote name context (from the local root of the remote host)
- The name path to a local name context (from the local root of the host on which you configured the Remote Name Context Binding)
- Whether the local context is to be bound into the remote context or the remote context is to be bound into the local context
- The name used to bind one context into the other

For example, suppose you wanted to bind a host tree for a host in another Component Broker network into a workgroup called CB390hosts in your own Component Broker network. The information you supply would be similar to the following:

- Bootstrap information: **bighost1.pok.ibm.com, port 900**
- Remote name context: **host**
- Local name context: **./workgroups/CB390hosts**

- Binding direction: **remote into local**
- Name: **bighost1**

Another example would be if you wanted to tie several Component Broker networks together using an inter-domain root. In this case, you would first have to create a name context within one of the Component Broker networks and bind it at “...” in the local root of one of the hosts. You can then bind the cell context for each of the Component Broker networks into the inter-domain root as follows:

- Bootstrap information: **domainmanager.austin.ibm.com, port 900**
- Remote name context: ...
- Local name context: .:
- Binding direction: **local into remote**
- Name: **cell3**

Then for each of the hosts in all of the Component Broker networks you would specify:

- Bootstrap information: **domainmanager.austin.ibm.com, port 900**
- Remote name context: ...
- Local name context: /
- Binding direction: **remote into local**
- Name: ...

Naming contexts

Names are bound to objects in a naming context. Naming contexts are collections of named objects. Within the naming context, each name must be unique. Naming contexts can be used to collect named objects that are related by some common business purpose. This can include, for instance, the customers, policy, and product-derivatives objects that an insurance agent has produced as part of a prospecting analysis.

In some sense, naming contexts are very analogous to file directories that have been expanded to include more than just file-objects. In the same way that a UNIX file directory can contain other directories, so can a naming context. Thus a naming context that is formed to group related named business objects can in turn be grouped into a larger category of related groups.

As each naming context is an object, its implementation is independent of any other naming context object. You can, in fact, introduce different implementations of naming contexts and bind them together within the same name tree. This is referred to as a *federated name tree*. The tree forms a loose

federation of naming context implementations. In this way the name tree can be backed by different qualities of service for the variety of requirements that may be needed.

The naming contexts that form the trunk of the system name space may represent an intolerable potential single point of failure. These contexts may need to be backed by a replicated data store, even at the expense of some performance penalty. Other naming contexts may change more frequently and therefore may need to be backed by a high-performance data store, even at the risk of some potential failure.

Component Broker provides one implementation of naming contexts, based on the DCE Cell Directory Service.

Object names

Objects are named within a naming context. Naming contexts are named within other naming contexts. All names are relative names, and always identify an object relative to a given naming context. An object can either have a *simple name* (the object name within its naming context) or a *compound name* (the object name within a higher-level naming context whose path leads to the object).

The name of an object within its naming context is a simple name. In the example name tree depicted in “Naming objects in the distributed object system” on page 181, “M” is a simple name referring to the object bound with the name “M” in its naming context. This is a one-part name that is divided into two fields: an *id* and a *kind* field. Both the *id* and the *kind* fields make up the object name and both contribute to the uniqueness of the name in a naming context. However, the *kind* field is optional and if not specified is dealt with as a null string.

A simple name can be combined with other simple names to form a compound name. For this reason, a simple name is also referred to as a *name component*.

When a naming context is bound in another naming context, the objects it contains can be referred to from higher-level naming context with a compound name. A compound name is a multi-part name, composed of the names of the intermediate naming contexts, plus the name of the object within its naming context. Referring to the example name tree depicted in “Naming objects in the distributed object system” on page 181, the compound names “B::F::M” and “F::M” can refer to the same object relative to different naming contexts. The components of a compound name are themselves the simple names of each intermediate naming context or target object in the path to the target object.

Any naming context operation accepts either a simple or compound name; the difference is strictly in whether you supply a one or multi-part name. If you supply a compound name in any of the naming context operations, the naming context recursively resolves the intermediate naming contexts before finally performing the operation on the naming context in which the target object is bound.

The `CosNaming::NamingContext` interface defined by the OMG requires that all names be supplied in the form of a sequence of structures of strings. More specifically, the IDL for a name is as follows:

```
module CosNaming
{
    typedef string Istring;
    struct NameComponent
    {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
};
```

As you can imagine, constructing a name can be somewhat cumbersome. For instance, to construct the compound name for "B::F::M" requires something like the following code in C++:

```
CosNaming::Name_var myName;
myName = new CosNaming::Name;
myName.length=3; // myName will have 3 components.
myName[0].id="B"; // Component 1
myName[0].kind="";
myName[1].id="F"; // Component 2
myName[1].kind="";
myName[2].id="M"; // Component 3
myName[2].kind="";
// Perform some naming context operation with myName ...
```

As you can see, this can make your code somewhat awkward. Component Broker introduces support for name strings. This allows you to form your names into a single string where the components are delimited by a forward (/) or back-slash (\), and where the *id* and *kind* fields of a name component are delimited by a period (.). With name strings, the name in the preceding example can be coded as:

```
const char* myName = "B/F/M";
// Perform some naming context operation with myName ...
```

The name components "B", "F", and "M" are delimited with forward slashes. Since all of these name components have a null *kind* field, then the *kind* field can be omitted and initializes to null automatically.

For every operation introduced by the `CosNaming::NamingContext` interface, Component Broker has augmented the interface with a corresponding operation, distinguished with the `_with_string` suffix, that takes a name string. These operations then use the `NamingStringSyntax::StandardSyntaxModel` object to convert the name string to a `CosNaming::Name` sequence.

The `NamingStringSyntax::StandardSyntaxModel` object is a local only helper object that can be used directly on the Component Broker server. While it is used under the covers of the Component Broker naming context, it can also be used for other purposes as well.

As described previously, name components can be composed of two fields: an *id* and a *kind*. This separation was introduced by the OMG to help create a separation between a name and its semantic. The *kind* is intended to add descriptive power to the name. Traditionally, operating systems and certain applications establish naming conventions to group related names, usually as part of a name extension. The Naming Service does not assign its own meanings to the *kind* field, but both the *id* and *kind* contribute to the uniqueness of the name.

The *kind* field could be used, for instance, to distinguish prospecting folders for new accounts versus established customers. Similarly, the *kind* field could also be used to distinguish different policy scenarios as follows: Given the existence of several prospecting folders such as Heidi Sutter.new prospect, Becky Newcombe.new prospect, and Jordan High.established prospect, and several policy scenarios such as early retirement.base line, *career change.variation*, and late retirement.variation, a compound name could be formed to Becky's career change policy scenario with the following name string:

```
prospecting folders/Becky Newcombe.new prospect/career change.variation
```

Notice that names can have embedded blanks, and the *id* and *kind* field for each name component is separated by a period.

Binding an object with a name

The following procedure demonstrates how you can bind an object in the system name space. This is useful for giving an object a readable name.

You can only bind an object in an existing naming context. If you use a compound name, all of the naming contexts identified in the name-path must already exist before the object can be bound. The name that you bind must be unique within the naming context. If the name you use is not unique within the target naming context then a `CosNaming::AlreadyBound` exception is raised.

1. Decide where you want to bind the object name: Where you decide to bind your object in the system name space can result in the requirement to create other naming contexts to build an appropriate name tree for the new object binding. The scope of visibility that you want the new binding to have determines whether the binding should be placed within the Host, Workgroup or Cell name-trees. If necessary, use the procedure described in “Creating a new naming context” on page 200 to create a new naming context.
2. Resolve to the target naming context: The target naming context is the naming context in which you want the object to be bound. You can resolve to the target name context with the resolve() operation by means of one of the following choices:
 - If you already have a reference to the target naming context you can skip to step 3 without doing anything further.
 - If you already have a reference to a naming context that is superior to the target naming context then you can invoke the resolve() operation on the superior naming context, supplying the name of the target naming context. This principle can be applied recursively for any of the successively more superior naming contexts.
 - If you do not already have a naming context, you can get the root to the system name space using the ORB::resolve_initial_references(“NamingService”) operation (the CORBA standard approach) or by obtaining it from the CBSeriesGlobal::nameService static function.
 Having acquired the root naming context, you can resolve directly to the intended naming context by invoking the resolve() operation and passing the complete path of intermediate naming context names as a compound name.
3. Bind the object: Once you are positioned at the target naming context, you can bind the object with a new name using the bind() or bind_with_string() operation on the target naming context.

You can combine steps 2 and 3 by supplying a compound name that includes the path to the target name context and the name of the object being bound in the name argument on the bind() or bind_with_string() operations. In this case, step 2 is performed implicitly.

The example that follows shows how to create a binding to a Policy object at the root of the system name space with the name “myPolicy”:

```
// Declare a reference to the Policy object
Policy_var myPolicyObject;
// Create an instance of Policy object in the normal manner
```

```

...
// Bind my Policy object at the root of the system name space.
CBSeriesGlobal::nameService()->bind_with_string(
    "myPolicy", myPolicyObject);

```

The previous example is not particularly interesting because it does not consider scoping or visibility. The following, more interesting, example binds the object in the cell name tree for use by the life insurance application. Specifically, the following example creates a binding to a Claim object with the name "myClaim" in the "LifeInsurance/Claims" naming context.

```

// Declare a reference to the Claims naming context,
// and to a Claim object.
IExtendedNaming::NamingContext_var claimsNamingContext;
Claim_var myClaimObject;

// Create an instance of Claim object in the normal manner
...
// Resolve to the "Claims" naming context in the cell name tree.
claimsNamingContext =
    CBSeriesGlobal::nameService()->resolve_with_string(
        "cell/applications/LifeInsurance/Claims");
// Bind my claim object in the Claims naming context.
claimsNamingContext->bind_with_string("myClaim", myClaimObject);

```

The last two statements in the previous example could be combined to form the following single statement:

```

CBSeriesGlobal::nameService()->bind_with_string(
    "cell/applications/LifeInsurance/Claims/myClaim", myClaimObject);

```

Resolving a named object

This procedure demonstrates how you can resolve an object in the system name space. This is useful for finding an object with a readable name. Only an object from an existing naming context can be resolved. If you use a compound name, all of the naming contexts identified in the name-path must already exist before the object can be resolved. The name you are resolving should already have been bound.

1. Resolve to the naming context: The *target naming context* is the target naming context in which you want resolve the object, that is where you expect the object to have been bound. You can resolve to the target name context with the resolve() operation using the following choices:
 - If you already have a reference to the target naming context you can skip to step two without doing anything further.
 - If you already have a reference to a naming context that is superior to the target naming context, you can invoke the resolve() operation on the superior naming context, passing in the name of the target naming context. This principle can be applied recursively for any of the successively more superior naming contexts.

- If you do not already have a naming context, you can get the root to the system name space using the `ORB::resolve_initial_references("NamingService")` operation (the CORBA standard approach) or by obtaining it from the `CBSeriesGlobal::nameService` static function.

Having acquired the root naming context, you can resolve directly to the intended naming context by invoking the `resolve()` operation and passing the complete path of intermediate naming context names as a compound name.

2. Resolve the object: Once you are positioned at the target naming context, you can resolve the object by its name using the `resolve()` or `resolve_with_string()` operation on the target naming context.

You can combine steps 1 and 2 by supplying a compound name that includes the path to the target name context and the name of the object being resolved in the *name* argument on the `resolve()` or `resolve_with_string()` operations. In this case, step 1 is performed implicitly.

The following example shows how to resolve a binding to a Policy object at the root of the system name space with the name "myPolicy":

```
// Declare a reference to the Policy object, and
// a general CORBA object for intermediate use.

CORBA::Object_var myObject;
Policy_var myPolicyObject;

// Resolve my Policy object at the root of the system name space.
myObject = CBSeriesGlobal::nameService()->resolve_with_string("myPolicy");

// Initially, the object coming back from the name context is a
// general CORBA object. It must then be narrowed to a Policy object.
myPolicyObject = Policy::_narrow(myObject);

// Verify that the narrow was successful
if (CORBA::is_nil(myPolicyObject) {
    cout << "The narrow operation failed" << endl;
    // do error processing here
}
```

The next example resolves an object in the cell name tree used by the life insurance application. Specifically, the following example demonstrates how to resolve a binding to a Claim object with the name "myClaim" in the "LifeInsurance/Claims" naming context.

```
// Declare a reference to the Claims naming context,
// a general CORBA object for intermediate uses
// and to a Claim object.
IExtendedNaming::NamingContext_var claimsNamingContext;
CORBA::Object_var myObject;
Claim_var myClaimObject;
```

```

// Resolve to the "Claims" naming context in the cell name tree.
// Narrow to Naming Context interface
myObject = CBSeriesGlobal::nameService()->resolve_with_string(
    "cell/applications/LifeInsurance/Claims");
claimsNamingContext = IExtendedNaming::NamingContext::_narrow(myObject);
if (CORBA::is_nil(claimsNamingContext) {
    cout << "The narrow operation failed" << endl;
    exit(-1);
}

// Resolve my claim object in the Claims naming context.
myObject = claimsNamingContext->resolve_with_string("myClaim");

// Initially, the object coming back from the name context is a
// general CORBA object. It must then be narrowed to a Policy object.
myClaimObject = Claim::_narrow(myObject);

// Verify that the narrow was successful
if (CORBA::is_nil(myClaimObject) {
    cout << "The narrow operation failed" << endl;
    // do error processing here
}

```

The previous two resolve statements can be combined into the single statement:

```

myObject = CBSeriesGlobal::nameService()->resolve_with_string(
    "cell/applications/LifeInsurance/Claims/myClaim");

```

Creating a new naming context

This procedure demonstrates how you can create new naming contexts bound in the system name space. This is useful for creating a context for binding related objects, for instance, a prospecting analysis folder or a list of brokerage services.

You can only create a new naming context from an existing naming context. This means you can only create new naming contexts within the system name space. You can get a reference to any naming context in the system name space by first getting the local root context using `ORB::resolve_initial_references("NameService")` and subsequently resolving to the naming context where you want to create (and bind) a new naming context.

1. Decide where you want to create the new naming context: Your new naming context is bound in the system name space, but you need to decide where. This may result in needing to create other naming contexts to build an appropriate name tree for the new naming context. Also, be sure to consider the scope of visibility that you want the new context to have. This determines whether the naming context should be placed within the Host, Workgroup, or Cell name-trees.

2. Resolve to the target naming context: The *target naming context* is the naming context in which you want the new naming context to be bound. You can resolve to the target name context with the `resolve()` operation using one of the following choices:
 - If you already have a reference to the target naming context, you can skip to step 3 without doing anything further.
 - If you already have a reference to a naming context that is superior to the target naming context then you can invoke the `resolve()` operation on the superior naming context, passing in the name of the target naming context. This principle can be applied recursively for any of the successively more superior naming contexts.
 - If you don't already have a naming context, you can get the root to the system name space using the `ORB::resolve_initial_references("NamingService")` operation (the CORBA standard approach) or by obtaining it from the `CBSeriesGlobal::nameService` static function.

Having acquired the root naming context, you can resolve directly to the intended naming context by invoking the `resolve()` operation and passing the complete path of intermediate naming context names as a compound name.
3. Create and bind the new naming context: Once you are positioned at the target naming context, you can create a new naming context and bind it with a new name at the same time using the `bind_new_context()` or `bind_new_context_with_string()` operation on the target naming context.

You can combine steps 2 and 3 by supplying a compound name that includes the path to the target name context and the name of the new naming context in the *name* argument on the `bind_new_context()` or `bind_new_context_with_string()` operation. In this case step 2 is performed implicitly.

A `bind_new_context` that is passed a compound name is defined as follows:

```

namingContext_X->bind_new_context(<nc1;nc2;...;ncn>)

```

and is identical to:

```

(namingContext_X->resolve(<nc1;nc2;...;ncn-1>)->bind_new_context(<ncn>)

```

where *nc1*, *nc2*, ..., *ncn-1*, and *ncn* are NameComponents. All the NameComponents prior to *ncn* need to be created first and exist in the system name space before `bind_new_context` can create *ncn*.

The following example creates the new naming context named "myNamingContext" using a compound name and binds it starting at the root of the system name space:

```

// Declare references to the new naming context.
IExtendedNamingContext::NamingContext_var tempNamingContext;
IExtendedNamingContext::NamingContext_var myNewNamingContext;

// Bind a new naming context named "Depts" at the root of the system
// name space.
tempNamingContext = CBSeriesGlobal::nameService(
    )->bind_new_context_with_string( "Depts" );

// Bind a new naming context using compound name "Depts/myNamingContext"
// start at the root and assign the resulting new naming context to my
// reference.
myNewNamingContext = CBSeriesGlobal::nameService(
    )->bind_new_context_with_string(
    "Depts/myNamingContext" );

```

Since the `bind_new_context_with_string` operation was used, the returned naming context is of type `IExtendedNamingContext::NamingContext`, and therefore does not need to be narrowed to this interface. If the `bind_new_context` operation was used, then the returned naming context would be of type `CosNaming::NamingContext` and would have to be narrowed if you wanted to use it as an `IExtendedNaming::NamingContext`.

A more interesting example would consider scoping and visibility to create a new naming context in the cell name tree for use by the life insurance application. Specifically, the following example names the new naming context "Claims" in the "LifeInsurance" naming context. By extension, it also creates the "LifeInsurance" naming context in the "applications" naming context.

```

// Declare a reference to the new naming contexts, and the application
// naming context.
IExtendedNaming::NamingContext_var aNewNamingContext;
IExtendedNaming::NamingContext_var applicationsNamingContext;

// Resolve to the "applications" naming context in the cell name tree.
applicationsNamingContext = CBSeriesGlobal::nameService(
    )->resolve_with_string("cell/applications");

// Create and bind the LifeInsurance naming context.
aNewNamingContext =
    applicationsNamingContext->bind_new_context_with_string(
        "LifeInsurance");

// Create and bind the Claims naming context.
aNewNamingContext =
    aNewNamingContext->bind_new_context_with_string("Claims");

```

The last statement utilized the `aNewNamingContext` variable. On the right side of the assignment it was used to reference the 'LifeInsurance' naming context. Having created and bound the new naming context to LifeInsurance, `aNewNamingContext` was then reassigned to refer to the Claims naming context.

As with any object that you bind to a naming context, new naming context names must also be unique within the target context. If the name you assign to the naming context you are creating is not unique within the target naming context then a `CosNaming::AlreadyBound` exception is raised.

Listing the contents of a naming context

The procedure that follows demonstrates how you can list out the contents of a naming context using the `CosNaming::NamingContext::list()` operation. This is useful if you want to determine what bindings exist or to perform some function iteratively over each entry of the naming context. The operation must target an existing naming context.

1. Determine the maximum number of entries you want to handle at a time: When you perform the `list()` operation in step 3, all of the entries in the target naming context are set aside in an iterator object. You are iterating through this object one or multiple entries at a time, getting that many entries back in a sequence (a subset of the iterator) through which you can then sub-iterate. You need to determine the number of entries you want to get back in the sequence at a time.

Memory and performance should both be considered when determining how many entries you want to handle at a time. Memory for the subset sequence is allocated based on the number of entries requested. To conserve memory, request fewer entries at a time.

Normally, the naming context you are listing, and therefore the iterator that is returned from the list request exists in a different process than your program, often on a different machine. Each request for another subset of entries is affected by network latency. The higher the number of requests required to get all of the entries in the iterator, the greater the impact on the performance of your program. Requesting fewer entries at a time increases the number of requests you have to make to the iterator to get all of the entries in the iterator. If performance is an issue, request more entries at a time. Requesting 1000 entries at a time may be a reasonable number for many situations.

2. Resolve to the target naming context: The target naming context is the naming context whose contents you wish to list. You can resolve to the target name context with the `resolve()` operation, using one of the following choices:
 - If you already have a reference to the target naming context you can skip to step 3.
 - If you already have a reference to a naming context that is superior to the target naming context, then you can invoke the `resolve()` operation on the superior naming context, passing in the name of the target naming context. This principle can be applied recursively for any of the successively more superior naming contexts.

- If you do not already have a naming context, you can get the root to the system name space using the `ORB::resolve_initial_references("NamingService")` operation (the CORBA standard approach) or by obtaining it from the `CBSeriesGlobal::nameService` static function.

Having acquired the root naming context, you can resolve directly to the intended naming context by invoking the `resolve()` operation and passing the complete path of intermediate naming context names as a compound name.

3. List out the contents of the naming context into an iterator: Once you are positioned at the target naming context, you can list its contents using the `list()` or `list_with_string()` operation on the target naming context. To keep your logic relatively simple do not extract any of the entries into a subset sequence; leave that to be included within your while loop. However, at the cost of some additional complexity, you can go ahead and extract the initial subset of entries on this request to reduce (by one) the number of times that you have to return to the iterator.
4. Get the next subset of entries from the iterator: You can do this with either the `next_one()` or `next_n()` operation on the iterator, depending on whether you want just one entry or more than one entry at a time.
5. Process a subset of the entries: Iterate through the sub-set of entries performing your function on each entry as appropriate for your application.
6. Repeat steps 4 and 5 until all of the entries in the iterator are exhausted: The `next_one()` and `next_n()` operations return a value of `True` if there are still more entries left in the iterator after the operation returns. Steps 4 and 5 should be repeated as long as these operations continue to return `True`, and 5 should be performed a final time when `false` is returned.

You can combine steps 2 and 3 by supplying a compound name that includes the path to the target name context and the name of the new naming context in the *name* argument on the `list()` or `list_with_string()` operation. In this case step 2 is performed implicitly.

The example that follows demonstrates listing out the contents of a naming context, 1000 entries at a time:

```
// Declare the claims naming context, a bindings
// list, bindings iterator, and sundry control variables.
IExtendedNaming::NamingContext_var claimsNamingContext;
IExtendedNaming::BindingStringList_var bindingsList;
IExtendedNaming::BindingStringIterator_var bindingsIterator;
CORBA::ULong i;
CORBA::Boolean continueFlag = 1;
const CORBA::ULong maximumEntries = 1000;
CORBA::Object_var tempObj;
```



```

// Resolve to the claims naming context
tempObj = CBSeriesGlobal::nameService()->resolve_with_string(
    "cell/applications/LifeInsurance/Claims");
claimsNamingContext = IExtendedNaming::NamingContext::_narrow(tempObj);
if (CORBA::is_nil(claimsNamingContext) {
    cout << "The narrow operation failed" << endl;
    exit(-1);
}

// List the contents of the claims naming context into an iterator.
// Don't extract any of the entries yet.
claimsNamingContext->list_with_string(0, bindingsList,bindingsIterator);

// Process the returned entries in bindingsList
while (continueFlag == 1) {
    // Get the next sub-set of entries
    continueFlag = bindingsIterator->next_n(maximumEntries,
        bindingsList);
    // Process each entry in bindingsList one at a time
    for (i=0; i < bindingsList->length; i++) {
        // Process bindingsList[i] ...
    }; // end for
}; // end while

```

Unbinding an object from a naming context

This procedure demonstrates how you can unbind an object from the system name space. You can use this to remove a binding, perhaps as a part of destroying the object itself. You can only unbind an existing object binding. If you use a compound name, all of the naming contexts identified in the name-path must already exist before the object can be unbound.

1. Resolve to the target naming context: The target naming context is the naming context in which you want to unbind an object. You can resolve to the target name context with the resolve() operation using one of the choices that follow:
 - If you already have a reference to the target naming context, you can skip to step 2.
 - If you already have a reference to a naming context that is superior to the target naming context, you can invoke the resolve() operation on the superior naming context, supplying the name of the target naming context. This principle can be applied recursively for any of the successively more superior naming contexts.
 - If you don't already have a naming context, you can get the root to the system name space using the ORB::resolve_initial_references("NamingService") operation (the CORBA standard approach) or by obtaining it from the CBSeriesGlobal::nameService static function.

Having acquired the root naming context, you can resolve directly to the intended naming context by invoking the `resolve()` operation and passing the complete path of intermediate naming context names as a compound name.

2. Unbind the object: Once you are positioned at the target naming context, you can unbind the object using the `unbind()` or `unbind_with_string()` operation on the target naming context.

You can combine steps 1 and 2 by supplying a compound name that includes the path to the target name context and the name of the object being unbound in the `name` argument on the `unbind()` or `unbind_with_string()` operation. In this case, step 1 is performed implicitly.

The example that follows demonstrates unbinding an object at the root of the system name space. It assumes that a binding to a Policy object at the root of the system name space with the name "myPolicy" already exists. The `unbind()` operation requires only the name of the binding as an argument.

```
// Unbind my Policy object at the root of the system name space.
CBSeriesGlobal::nameService()->unbind_with_string("myPolicy");
```

The example that follows demonstrates unbinding an object in the cell name tree, specifically a Claim object with the name "myClaim" in the "LifeInsurance/Claims" naming context.

```
// Declare a reference to the Claims naming context.
IExtendedNaming::NamingContext_var claimsNamingContext;
CORBA::Object_var tempObj;

// Resolve to the "Claims" naming context in the cell name tree.
tempObj = CBSeriesGlobal::nameService()->resolve_with_string(
    "cell/applications/LifeInsurance/Claims");
claimsNamingContext = IExtendedNaming::NamingContext::_narrow(tempObj);
if (CORBA::is_nil(claimsNamingContext) {
    cout << "The narrow operation failed" << endl;
    exit(-1);
}

// Unbind my claim object in the Claims naming context.
claimsNamingContext->unbind_with_string("myClaim");
```

The last two statements can be combined in to the single statement could be combined as follows:

```
CBSeriesGlobal::nameService()->unbind_with_string(
    "cell/applications/LifeInsurance/Claims/myClaim");
```

Two caveats apply when unbinding an object from a naming context:

- The name that you unbind must already exist. If the name you specify does not exist then a `CosNaming::NotFound` exception is raised.

- Naming contexts can be unbound from a superior naming context only if the target naming context had been previously bound using the `bind`, `rebind`, `bind_context` or `rebind_context` operations (or their `_with_string` variations). In other words, for two naming contexts A and B, if you create a new naming context, called C, in A (using `A->create_new_context_with_string("C")`, for instance) and later bind C in B, then you can unbind C from B. However, you cannot unbind C from A. You can only remove C from A by destroying C, and only after C has been emptied of any bindings it may have contained.

Local root naming context and the bootstrap host

The local root context is at the top of the name tree for every host. This is an entry in the system name space and is the reference returned from the `ORB::resolve_initial_references("NameService")` method and from the `CBSeriesGlobal:nameService()` method.

In some sense, the local root context is like the root of a UNIX file system. Workgroup and cell name trees are bound to the root of the host name tree (the local root context) in the same way that remote, distributed file systems are mounted to the root of the local file system.

Each host machine (each machine with one or more server processes) has its own name tree. Thus, each host has a local root context. When the `resolve_initial_references()` method is invoked on a host machine (in a client or server process on that machine), that method returns a reference to the local root context for the name tree on that host.

However, client machines (those without any server processes) do not have a local name tree. When invoked on a client machine, the `resolve_initial_references()` method must return a reference to a root naming context on a remote host machine. It does this using a bootstrapping mechanism.

For this to work, you must identify the name of the host whose local root context you want to use. This host then is referred to as the *bootstrap-host* for that client machine, the host from which the client is bootstrapped in to the distributed system. The name tree local to that host, in effect, ends up being shared by all of the client machines that identified that host as their bootstrap-host.

You identify your bootstrap-host when you install Component Broker on the client machine. The install utility prompts you for your bootstrap-host, and records that information for later use by the run time. In addition, the use of Client Styles defined with the System Manager allow for specification of a bootstrap-host other than the one designated at install time.

Since a number of client machines can resolve their local root context from a common bootstrap host the local name tree on that host effectively becomes a shared resource for those machines. It also becomes a potential single point of failure. If the bootstrap-host is down then the client cannot resolve its initial Naming Service reference, and thus cannot resolve any other resource from the system name space. Modification of the bootstrap host specification within the Client Style can be used to redirect the client to a different bootstrap host when needed to bypass a failing bootstrap host.

To prevent the bootstrap-host from becoming a performance bottle-neck, the number of client machines that target the same bootstrap-host should be limited. As a rule of thumb, no more than 5-10 client machines should use the same bootstrap-host. The actual limit depends on a number of factors such as the reliability and capacity of the bootstrap-host, and the nature of your client programs, therefore you may be able to significantly exceed this rule of thumb in some cases.

Another approach to avoiding the potential single point of failure and performance bottleneck problems is to configure your client as a server machine when you install Component Broker. This results in one or more server processes being configured, and a host name tree being automatically built for your machine. However, this also increases the footprint requirements for your client machine as well.

Absolute and relative names

As with traditional file systems, the Naming Service handles both absolute and relative paths. In fact, all compound names are handled as relative names since they are always used in the context of a specific naming context. However, the Component Broker Naming Service has been implemented to recognize a leading "/" (forward-slash) or "\" (back-slash) as a special case.

If the first character in a name string is a "/" (forward-slash) or "\" (back-slash) any naming context resolves the local root context, and continues to resolve the remaining name from the local root context. Thus, resolving the name `applications/LifeInsurance/Claim` from the local root context navigates to the same object as resolving `/applications/LifeInsurance/Claim` from the hosts context in the cell name tree.

The special case mechanism of using "/" or "\" to refer to the local root will generally give you the result you want if you have been traversing the name tree by starting from your local root. However, be aware that this mechanism really means to start at the local root relative to the name context on which the operation was invoked, which is not necessarily your local root. Therefore, this mechanism should be avoided. In order to avoid this potential

inconsistency, it is preferable to use `CBSeriesGlobal::nameService()` to refer to your local root rather than using the leading “/” or “\”.

Summary of the naming context interface

The complete Component Broker naming context interface is described in the combination of the `CosNaming::NamingContext` and `IExtendedNaming::NamingContext` interfaces. The `CosNaming::NamingContext` interface is the standard interface for naming contexts and introduces the following operations:

- `void bind(name, obj)`
- `void rebind(name, obj)`
- `void bind_context(name, naming_context)`
- `void rebind_context(name, naming_context)`
- `Object resolve(name)`
- `void unbind(name)`
- `NamingContext new_context()`
- `NamingContext bind_new_context(name)`
- `void destroy()`
- `void list(how_many, binding_list, binding_iterator)`

The `bind()` operation can be used to bind a name to any managed object, and the `bind_context()` operation can be used to bind a name to a sub-context. The `rebind()` operation can be used to change the object to which a name is bound, and the `rebind_context()` can be used to do the same for a sub-context. The `unbind()` operation can be used to remove a binding between a name and an object. The `unbind()` operation removes the binding from the naming context, but does not remove or destroy the object itself.

The `resolve()` operation can be used to find what object has been bound to a name. The `new_context()` operation is generally used to create a new naming context which is not bound to any other naming context.

The `bind_new_context()` operation can be used create a new naming context which is bound to a name. The `destroy()` operation can be used to remove the naming context. This is functionally equivalent to the `remove()` operation on any managed object. The naming context must not contain any bindings when it is destroyed. Neither the `remove()` nor the `destroy()` operation removes any bindings contained in the naming context: these must be un-bound separately using the `unbind()` operation. The `list()` operation can be used to return an iterator for all of the bindings in the naming context.

In addition, Component Broker extends the `CosNaming::NamingContext` interface with the sub-interface `IExtendedNaming::NamingContext`. The `IExtendedNaming::NamingContext` interface introduces the following operations:

- void bind_with_string(name, obj)
- void rebind_with_string(name, obj)
- void bind_context_with_string(name, naming_context)
- void rebind_context_with_string(name, naming_context)
- Object resolve_with_string(name)
- void unbind_with_string(name)
- NamingContext bind_new_context_with_string(name)
- void list_with_string(how_many, binding_list, binding_iterator)

All of these operations accept names in the form of a name-string and perform exactly the same functions as their counterparts introduced in the `CosNaming::NamingContext` interface.

To support converting between name-strings and the standard `CosNaming::Name` name-structures, Component Broker introduces the `NameStringSyntax::StringName` and accompanying `NameStringSyntax::StandardSyntaxModel` interfaces with the following operations:

- `NameString` name_to_string(name)
- `CosNaming::Name` string_to_name(name)

Implementing the Naming Service

The Component Broker Naming Service has very few configuration requirements. Most of the configuration work that needs to be performed is done automatically during installation and server initialization

Component Broker naming contexts map to DCE/CDS directories. Using the DCE Director, you can navigate and examine the Component Broker system name space. See the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide* for information on the structure of the Component Broker system name space within DCE/CDS.

You have control over which bootstrap-host is used by which client machine by specifying this when the client machine is installed. Component Broker administrators should plan out their system topology, denoting which hosts will be server hosts (containing one or more server processes) and which will be client machines (containing no server processes). This can be configured with the Component Broker System Management application, see the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide* for more information. As new client machines are added to the topology, the administrator should explain which bootstrap-host that particular client should be using so that this can be entered in during client installation.

Integrity of the system name tree

The integrity of the name space is compromised whenever the state of the name space and the state of the objects in the system do not agree, for whatever reason. For example, there may be objects which exist that are supposed to be bound into the name tree for which the bindings no longer exist. This leads to the object not being found by the requester when the name tree is used to locate it. Another problem occurs when a binding in the name space exists and the object to which it refers no longer exists. When the reference is found from the name space, the subsequent usage of the object by the requester fails because the object no longer exists. In addition, there are other more subtle inconsistencies which can also cause problems.

One approach to handling the potential integrity problem would be to back up the entire system state and name tree state at a point in time when there were no integrity problems. However, this is impractical in any environment that has more than a few hosts in it, and is intolerable in a production environment. The entire system would have to be quiesced at one time, backed up and then restarted. Even if this was acceptable, a subsequent minor inconsistency in the name tree would require reverting the entire system back to the previous state when it was last backed up.

Another approach is to subdivide the environment so that it can be backed up a piece at a time, such as an individual host or even an individual server. The problem with this approach is that the structure of the name tree is such that the bindings in the name tree cannot be accurately associated with a particular host or server.

The Component Broker System Manager provides facilities to address these issues. The system name space can be resynchronized with the state of objects in the running system when an integrity problem is encountered. See the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide* for a more in depth explanation of the potential situations that can be encountered and the facilities provided to resynchronize the name space with the rest of the object environment.

The string syntax object

The `IExtendedNaming::NamingContext` interface introduced by Component Broker utilizes a string syntax form of the `CosNaming::Name` (which is a sequence of structures) that is used by the `CosNaming::NamingContext` interface. The string syntax is a much easier form to use than the `CosNaming::Name` and simplifies the code you need to write to invoke methods on a naming context object. It also provides a simpler way to document and discuss specific name paths, such as we have done in the other sections of this Naming Service documentation.

For the majority of your usage of string syntax names, you will simply define the appropriate string and pass it to an `IExtendedNamingContext` object using one of the `xxxx_with_string` methods. However, Component Broker also provides an object which you may choose to use to do conversions between the string form and `CosNamingName` form of names if needed. The string syntax object is a helper, local only object that can be instantiated strictly to convert names back and forth between their two forms. This object was introduced to let you specify names as a string, using embedded delimiters and separators, and easily convert these into a name-structure by parsing the string based on a particular syntax model.

Architecture of the string syntax object: The interfaces introduced by Component Broker are intended to provide an architectural basis by which many different syntaxes can be introduced for the representation of the string form of a name. However, the current implementation of Component Broker only supports a single syntax, which is modeled after the X/Open Federated Naming (XFN) specification standard Composite Name String Syntax. We will refer to this syntax supported by Component Broker as the Default Syntax Model. The extensible architecture introduced by Component Broker enables two forms or variation from the default syntax model:

1. Using a totally different parsing algorithm
2. Using the same parsing algorithm but with a customized set of characters

The following is a diagram of the hierarchy of interfaces used to provide this architecture:

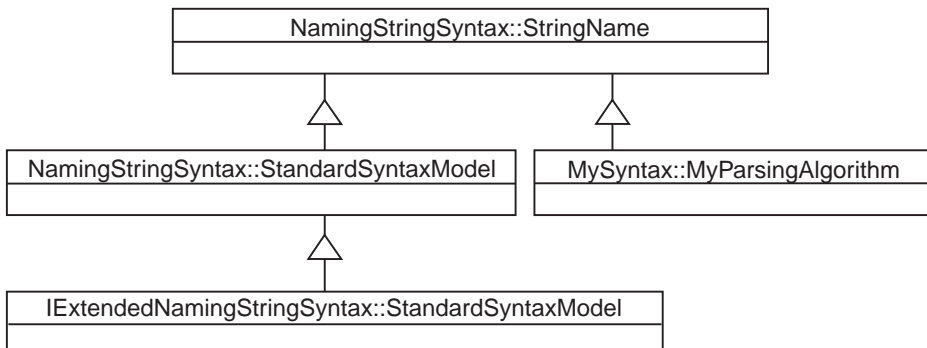


Figure 8. String syntax object architecture

The `NamingStringSyntax::StringName` is a base-interface that introduces two methods for converting names between their two forms. At this level, the actual syntax of the string form of the name is not defined. The `NamingStringSyntax::StandardSyntaxModel` interface introduces the string syntax which is parsed similar to the XFN syntax, but with the ability to customize the characters used by the parsing algorithm. The

`IExtendedNamingStringSyntax::StandardSyntaxModel` interface defines the actual implementation used by Component Broker, and it customizes the characters such that the syntax is similar to the XFN syntax. The `MySyntax::MyParsingAlgorithm` shows how the `NamingStringSyntax::StringName` interface could be extended with a totally different set of parsing rules (this is not an actual interface in Component Broker).

The `NamingStringSyntax::StringName` interface introduces two methods used for conversion between the two forms of names. The methods are:

- `string_to_name`
- `name_to_string`

The `NamingStringSyntax::StandardSyntaxModel` interface introduces attributes which are used to define the customizing aspects of the parsing rules. The attributes are:

- `syntax_direction`
- `syntax_absolute_prefix`
- `syntax_reserved_names`
- `syntax_delimiter`
- `syntax_separator`
- `syntax_begin_quote`
- `syntax_end_quote`
- `syntax_escape`
- `syntax_code_set`
- `syntax_locale_info`

The `IExtendedNamingStringSyntax::StandardSyntaxModel` interface introduces the `_create` static function which is used to instantiate an instance of the local only helper object configured as the default syntax model.

Standard syntax model grammar: This section describes the full grammar for the standard syntax model. In this syntax model, name strings can be used to specify a compound name, that is, a name that represents a traversal path through the name tree. The parsing rules for the strings are customized by values specified as attributes. Component Broker implements this grammar with a single set of values for the attributes, which is referred to as the default syntax model.

As an overview of the default syntax model, the grammar is straight forward. Name components are delimited by either a forward-slash (/) or a back-slash (\). The *id* and *kind* fields are separated by a period (.). Special characters such as quotes, double quotes, forward-slashes, back-slashes, and periods can be escaped in unquoted strings. Alternately, name components containing any of these special characters can be quoted with either single or double quotes.

The standard syntax model rules for parsing string names are defined as:

1. Absolute-prefix, reserved-name, delimiter, separator, begin-quote, end-quote and escape strings are distinguished elements.
2. An absolute-prefix at the beginning of the string is parsed as is. That is, it forms its own name component.
3. The escape string immediately preceding a delimiter, separator, begin-quote, end-quote or escape string escapes those elements even within a quoted string.
4. In an unquoted string, an unescaped delimiter separates two name components.
5. Within a name component, unquoted reserved names are parsed as is. That is, they are not separated even if they contain a separator.
6. If an escaped delimiter occurs within an unquoted name component, the delimiter is treated as a character value in the name component.
7. Within an unquoted name component, the first unescaped separator separates the *id* and *kind* fields of the name component.
8. If an escaped separator occurs within an unquoted string, the separator is treated as a character value in the name component.
9. An unescaped begin-quote preceding an *id* or *kind* field of a name component must be matched by a unescaped end-quote at the end of the field. If there are multiple values for begin-quote and end-quote, a specific begin-quote value must be matched with its corresponding end-quote value. An unmatched quotation raises an exception.
10. Other quotes (not including the end-quote used to close the quotation) embedded in a quoted string are treated as character values in the *id* or *kind* field.
11. Begin-quotes or end-quotes within an unquoted name component *id* or *kind* field are treated as simple characters and do not need to be matched.
12. If an escaped or unescaped delimiter occurs within a quoted string, the delimiter is treated as a character value in the name component *id* or *kind* field.
13. If an escaped or unescaped separator occurs within a quoted string, the separator is treated as a character value in the name component *id* or *kind* field.
14. If an escaped escape occurs within a quoted or unquoted string, the escape is treated as a character value in the name component *id* or *kind* field.

The following attributes apply to the standard syntax model and are described with the default values which make up the default syntax model (the only model currently supported by Component Broker).

syntax_direction

Indicates the direction in which the parsed string is loaded into components of the CosNaming::Name. (Default value: kLeftToRight)

syntax_absolute_prefix

The string(s) that define absolute prefixes. Absolute prefixes are used to denote an absolute name that is applied to a local root context. (Default value: "/", "\")

syntax_reserved_names

The string(s) that define reserved names. Reserved names are not parsed, even if they contain separator characters. (Default value: ":", "...")

syntax_delimiter

The string(s) that are used to separate name components. If more than one character is specified then either can be used interchangeably. (Default value: "/", "\")

syntax_separator

The string(s) that are used to separate the *id* and *kind* fields of a name component. If more than one character is specified then either can be used interchangeably. (Default value: ".")

syntax_escape

The character that is used to escape the parsing rules. Only one character can be specified. (Default value: "\")

syntax_begin_quote

The string(s) that define the beginning of a quoted string. If more than one character is specified then either can be used interchangeably, but they must be matched with a corresponding `syntax_end_quote` character (as determined by their index position within the character set). (Default value: "" (double quote), "" (single quote))

syntax_end_quote

The string(s) that define the end of a quoted string. The number of characters specified must match the number of characters specified in `syntax_begin_quote`. If more than one character is specified then either can be used interchangeably, but must match the corresponding `syntax_begin_quote` character used. (Default value: "" (double quote), "" (single quote))

syntax_code_set

Defines the code set used for parsing the string. (Default value: kISOLatin1)

syntax_locale_info

Defines the locale used for parsing the string. (Default value: kUS_ENG)

The preceding syntax rules and attributes are represented by the following modified BNF:

```

<name_string> ::= [ <absolute_prefix> ] <relative_name>;
<relative_name> ::= { <name_component> <delimiter> } *
    <name_component>;
<name_component> ::= <reserved_name>
    <name_id> [ <separator> [ <name_kind> ] ];
<name_id> ::= <component_string>;
<name_kind> ::= <component_string>;
<component_string> ::= <unquoted_string>
    | <begin_quote> <quoted_string> <end_quote>;
<unquoted_string> ::= <opening_char> <unrestricted_char> *;
<quoted_string> ::= <nonclosing_char> +;
<opening_char> ::= any character, including
    <escape_sequence>, except
    <begin_quote>, <end_quote>, <delimiter>, or
    <separator>;
<unrestricted_char> ::= <opening_char> | <begin_quote> | <end_quote>
    | <delimiter> ;
<nonclosing_char> ::= any character, including
    <escape_sequence>, <begin_quote>,
    <delimiter>, or <separator>,
    except <end_quote> corresponding
    to <begin_quote> of
    <quoted_string>;
<escape_sequence> ::= <escape> <delimiter>
    | <escape> <separator>
    | <escape> <begin_quote>
    | <escape> <end_quote>
    | <escape> <escape>;
<absolute_prefix> ::= defined in syntax_absolute_prefix
    attribute;
<delimiter> ::= defined in syntax_delimiter attribute;
<reserved_name> ::= defined in syntax_reserved_names attribute;
<separator> ::= defined in syntax_separator attribute;
<begin_quote> ::= defined in syntax_begin_quote attribute;
<end_quote> ::= defined in syntax_end_quote attribute;
<escape> ::= defined in syntax_escape attribute;

```

Converting between a name-string and a name-structure: This procedure is used to convert a name-string to a name-structure, or vice versa, using the string-syntax object. This is useful when you want to limit yourself to using only the standard CosNaming::NamingContext interface, but want to allow the user to specify names in the form of a string. The string-syntax object can be used to convert the user-specified string-name to a name-structure as required by the standard CosNaming::NamingContext() operations. The string-syntax object is a local only object and can only be created on a CORBA C++ client or in a Component Broker server (from a C++ or Java business object). The usage scenario is as follows:

1. Create an instance of a string-syntax object: The string-syntax object is a local only object and should be created using the normal Component Broker lifecycle model for local only objects. That is, invoke the `_create`

static member function on the
IExtendedNamingStringSyntax::StandardSyntaxModel class.

2. Convert the name-string to a name-structure: Given a name-string conforming to the default syntax model grammar, you can convert it to a name-structure (a `CosNaming::Name`) using the `string_to_name()` operation.
3. Or, convert the name-structure to a name-string: Conversely, you can convert a name-structure to a name string that conforms to the default syntax model grammar using the `name_to_string()` operation.

The following example demonstrates conversion of
`/cell/applications/LifeInsurance/Claims/myClaim`, from a name-string into a name-structure:

```
// Declare a string-syntax object, and a name-structure
IExtendedNamingStringSyntax::StandardSyntaxModel_var anSSO;
CosNaming::Name_var aNameStructure;

// Create a string-syntax object
anSSO = IExtendedNamingStringSyntax::StandardSyntaxModel::_create();

// Set and convert the name-string to a name-structure
aNameStructure=anSSO->string_to_name(
    "cell/applications/LifeInsurance.app/Claims/myClaim");
```

At this point, the name-structure is composed of the five elements with the *id* and *kind* field for each element set as follows:

```
1) id = "cell"
   kind = ""
2) id = "applications"
   kind = ""
3) id = "LifeInsurance"
   kind = "app"
4) id = "Claims"
   kind = ""
5) id = "myClaim"
   kind = ""
```

The corresponding *kind* field in each case is an empty string (i.e. `""`). Note that if the name-string is not formed in accordance to the standard syntax model, an `IllegalStringSyntax` or `UnMatchedQuote` exception is raised.

The following example demonstrates conversion of a name-structure into a name-string:

```
// Declare a string-syntax object, a name-string and a name-structure
IExtendedNamingStringSyntax::StandardSyntaxModel_var anSSO;
string aNameString;
CosNaming::Name_var aNameStructure;

// Create the string syntax object and name structure
```

```

anSSO = IExtendedNamingStringSyntax::StandardSyntaxModel::_create();
aNameStructure = new CosNaming::Name;

// Initialize the name-structure
aNameStructure->length=5;// The name-structure will have 5 elements

aNameStructure[0].id =CORBA::string_dup("cell");
aNameStructure[0].kind=CORBA::string_dup("");

aNameStructure[1].id =CORBA::string_dup("applications");
aNameStructure[1].kind=CORBA::string_dup("");

aNameStructure[2].id =CORBA::string_dup("LifeInsurance");
aNameStructure[2].kind=CORBA::string_dup("app");

aNameStructure[3].id =CORBA::string_dup("Claims");
aNameStructure[3].kind=CORBA::string_dup("");

aNameStructure[4].id =CORBA::string_dup("myClaim");
aNameStructure[4].kind=CORBA::string_dup("");

// Convert the name-structure to a name-string
aNameString=anSSO->name_to_string(aNameStructure);

```

At this point, *aNameString* should be `cell/applications/LifeInsurance.app/Claims/myClaim`.

One aspect of conversion between name strings and name structures to be aware of is the proper handling of names which contain imbedded delimiter or separator characters. This is most often seen with host names. For example, given `myHost.austin.ibm.com` as a host name within a name string, the dot (".") between `myHost` and `austin` will be interpreted as a separator. Therefore, the string: `cell/hosts/myHost.austin.ibm.com/applications` would be converted to:

```

1) id = "cell"
   kind = ""
2) id = "hosts"
   kind = ""
3) id = "myHost"
   kind = "austin.ibm.com"
4) id = "applications"
   kind = ""

```

However, when a host binds itself into the hosts context of the cell name tree, the entire host name is used for the *id* field, such as:

```

id = "myHost.austin.ibm.com"
kind = ""

```

This does not match element three in the above name, and therefore the appropriate string syntax for the above string would be `cell/hosts/'myHost.austin.ibm.com'/applications` which results in a name of:

```
1) id = "cell"
   kind = ""
2) id = "hosts"
   kind = ""
3) id = "myHost.austin.ibm.com"
   kind = ""
4) id = "applications"
   kind = ""
```

Chapter 8. Security Service

This section includes the following topics:

- Security in the distributed object system
- Principals, credentials, and secure associations
- Authentication and end-users
- Security Service objects
- Security and servlets in WebSphere Enterprise

Security in the distributed object system

A fundamental concern in distributed systems in general is how to protect data and business assets available through the information system. This is no less true in distributed, object-oriented systems. Valuable information exists in business objects. This information can be manipulated and accessed remotely and therefore must be protected from unauthorized use.

Component Broker provides the Security Service to help protect these assets. The Security Service is used primarily to prevent end users from accessing information and resources that they're not authorized to use. This predominantly covers distributed objects, but by extension includes any of the information and resources from other non-OO, or non-distributed sources that those business objects use.

In many cases Component Broker is used to wrapper legacy information system resources, such as business applications and enterprise data. Often, those resources have always been centralized resources, held in a physically secure environment or with restricted access over controlled access channels.

A key objective of object-oriented programming and business re-engineering is to provide for the abstraction of business resources that enables them to be used more readily in new applications. This has the effect of proliferating access to those legacy resources. Access may be increased to resources that have been traditionally (either intentionally or due to the limitations of technology) more restricted. Thus, the solution has the potential for undermining the protection that these resources require and have traditionally enjoyed.

The Security Service must compensate for any protections that may be otherwise lost due to the increased accessibility of business objects in a distributed object system. At the same time, any benefit an application programmer receives by using Component Broker should not be taken away

by the Security Service, except to prevent unauthorized access to resources. By extension, this means that if security policies for a set of legacy resources have already been established and are in use in existing production systems, then it should be possible to use those same policies to protect resources in the object system; it shouldn't be necessary to re-specify existing security policies or to keep two sets of policies in sync.

Object systems tend to introduce many more independent objects than equivalent procedural systems. Procedural systems tend to wrap up individual objects into larger-grained artifacts, such as resources managers or database tables. The presence of so many objects introduces the potential for administrative scalability issues that present their own security exposures: when administration becomes overwhelming, administrators just stop administering and objects remain unprotected. The Security Service guards against this threat by factoring security policies across a server, thus forming an administrative boundary for controlling unauthorized access to the objects contained within that server, and to the resources that are used by that server.

Component Broker security provides support for authenticating users to prevent unauthenticated access to secure servers. It can certify to business objects the principal on whose behalf any given method request is being made. The Security Service also provides support for protecting message traffic between clients and server, and between servers and servers.

Principals, credentials, and secure associations

Both users and servers can be authenticated to the system. Either users or servers can have identities, meaning either can be identified and access-controls exercised for any resources that are accessed on their behalf. Any entity that can be identified and authenticated in the system is referred to as a *principal*. A principal can be either a client-user or a server process. It can also be other software or abstract entities if those things can be associated with an identity and they have the means for authenticating themselves.

When a principal has been authenticated, the Security Service creates a credential for that principal. A credential is an object that represents the authenticity of a given principal. As such, it represents the principal, but only after the principal has been certified as being authentic.

In a secure server, all activities occur on behalf of a given principal. This is achieved in the following manner: When a principal is authenticated at a client (a client-principal), a credential is formed for that client. The credential is associated with the process. The credentials are communicated to the server along with any method requests that originate from that server, and the

thread of execution in the server is tagged with the credentials of the client-principal that originated the request.

The Security Service is able to efficiently and safely communicate the credentials for the client-principal by establishing a secure association between the client and the server. Each client-server forms a unique association, even when the server acts as a client to another server. The secure-association is also used to protect any message traffic between the client and the server processes.

Manipulating credentials

As discussed in “Principals, credentials, and secure associations” on page 222, when a principal is authenticated, a credential is formed to represent that principal. At a client, the credential formed when you authenticate the principal becomes the default credential for the process. It is theoretically possible for each object to have its own-credentials, but in this release of Component Broker any object’s own-credentials are synonymous with the default-credential for the process. In the absence of a thread-specific credential the default-credential is used for any activities performed in the client, or for any method requests invoked on a remote object.

If another principal is authenticated, the new credential replaces the previous as the process’ default credential. However, unless the previous credential is destroyed, it remains a valid credential, at least until the credentials expire or are explicitly removed by a program. Once a credential is formed, it can be obtained by a local program.

Developing applications that use the Security Service

VisualAge C++

For client applications that will run as a pure Component Broker C++ clients, the Security Service client library, `somsccl1.lib`, should be linked in your client application makefile.

For server applications that will run inside a Component Broker C++ server process, the Security Service server library, `somscsl1.lib`, should be linked in your server application makefile.

Microsoft Visual C++

The Security Service client library is not available for ActiveX clients due to a packaging problem. Even though the Security Service APIs are not available, ActiveX clients can still be enabled with security.

Getting a current object

This procedure demonstrates how you can obtain a security Current object with which you can obtain or manipulate the credentials that you want to use in your program.

You can obtain a Current at either the client or in the server. However, you can only get a Current object if the Security Service run time has been installed and the ORB has been initialized.

Obtain a Current object using the following steps:

1. Obtain a reference to the CORBA::ORB object. You can obtain a reference to the CORBA::ORB object by invoking the CORBA::ORB_init static function (the CORBA standard approach) or by obtaining it from the CBSeriesGlobal::orb static function (this static function is set by the CBSeriesGlobal::Initialize() method).
2. Get a CORBA::Current. Use the CORBA::ORB::resolve_initial_references() method to get a Current object, passing in the name of the security current: "SecurityCurrent".
3. Narrow to the security Current interface. Narrow the Current to the security Current interface. In fact the Security Service introduces two derivatives of the Current interface: SecurityLevel1::Current, and SecurityCurrent. You should narrow to the SecurityCurrent interface as a matter of practice.

The following example demonstrates how to obtain a SecurityCurrent object:

```
// Declare a reference to the CORBA::Current and the
// SecurityCurrent objects.

CORBA::Object_var object;
SecurityCurrent_var securityCurrent;

// Use the CBSeriesGlobal::orb to get a CORBA::Current

object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SecurityCurrent");

// Narrow to the SecurityCurrent.

securityCurrent = SecurityCurrent::_narrow(current);
```

Acquiring the security attributes of a credential

This procedure demonstrates how you can acquire the security attributes of a credential. This is used to determine the security name and host identity of the principal that invoked the current method request, including the host where the principal is logged in.

This procedure is performed on a Credentials object. The security name and host name are security attributes that have been introduced by Component Broker. Therefore, they are identified by the IBM_BOSS_FAMILY_DEFINER, in attributes family 2. The security run time must be installed and the ORB must be initialized.

Acquire the security attributes of a credential using the following steps:

1. Get a credentials object. Use the procedure described in “Acquiring a credential on a thread” on page 227.
2. Get the desired attribute. Use the SecurityLevel2::Credentials::get_attributes() method to get the desired attributes from the credentials. Component Broker introduces two security attributes, CredAttrSecName, and CredAttrHostName. These are defined in family 2 where IBM_BOSS_FAMILY_DEFINER is the definer of this family.

The following example demonstrates how to acquire the received-credentials, and then acquire the principals security name and host name:

```
#include <IExtendedSecurity.hh>
#include <CBSeriesGlobal.hh>
// Declare a reference to the CORBA::Current, SecurityCurrent,
// Credentials and CredentialsList objects, and Atributes, AttributeList,
// and AttributeTypeLists.

CORBA::Object_var object;
SecurityCurrent_var securityCurrent;
SecurityLevel2::CredentialsList_var receivedCredentialsList;
SecurityLevel2::Credentials_var receivedCredentials;
::CORBA::String securityName;
::CORBA::String hostName;
Security::AttributeList_var attributesList;
Security::AttributeTypeList attributesTypeList(2);

// Use the CBSeriesGlobal::orb to get a CORBA::Current

object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SecurityCurrent");

// Narrow to the SecurityCurrent.

securityCurrent = SecurityCurrent::_narrow(current);

// Get the received-credentials list

receivedCredentialsList = securityCurrent->received_credentials();

// Pull out the received-credentials from the first (0th) position

receivedCredentials = (*receivedCredentialsList)[0];"

// Get the principal's security name and host name.
```

```

// Set the length to 2
attributesTypeList.length(2);

// Create an entry for security name
attributesTypeList[0].attribute_family.family_definer=
    IExtendedSecurity::IBM_BOSS_FAMILY_DEFINER;
attributesTypeList[0].attribute_family.family=2;
attributesTypeList[0].attribute_type=
    IExtendedSecurity::CredAttrSecName;

// Create an entry for host name

attributesTypeList[1].attribute_family.family_definer=
    IExtendedSecurity::IBM_BOSS_FAMILY_DEFINER;
attributesTypeList[1].attribute_family.family=2;
attributesTypeList[1].attribute_type=
    IExtendedSecurity::CredAttrHostName;

// Get the attributes

attributesList = receivedCredentials->get_attributes(
    attributesTypeList);

// Extract the security name

securityName = CORBA::string_dup(
    (::CORBA::String) &(*attributesList)[0].value[0]);

// Extract the host name

hostName = CORBA::string_dup(
    (::CORBA::String) &(*attributesList)[1].value[0]);

```

The following is the proper way to create a valid Security::AttributeTypeList for get_attributes() of a Credentials object:

```

org.omg.Security.AttributeType[] attrTypeList =
    new org.omg.Security.AttributeType[2];

org.omg.Security.ExtensibleFamily attribute_family0 =
    new org.omg.Security.ExtensibleFamily((short)8, (short)2);

org.omg.Security.ExtensibleFamily attribute_family1 =
    new org.omg.Security.ExtensibleFamily((short)8, (short)2);

attributeTypeList[0] =
    new org.omg.Security.AttributeType(
        attribute_family0, com.ibm.IExtendedSecurity.CredAttrSecName.value );
attributeTypeList[1] =
    new org.omg.Security.AttributeType(
        attribute_family1, com.ibm.IExtendedSecurity.CredAttrHostName.value);

```

Acquiring a credential on a thread

This procedure demonstrates how you can acquire a credential on a thread of execution. This is used to determine the identity of the principal that issued the request, the identity of the server, or the identity used for any further method requests on downstream servers.

Any given thread of execution at either the client or the server may be associated with one of the following credentials:

Received-credential

The received-credential identifies the principal for whom this request is being performed. In the server, the received-credential is the credential received with the currently executing method request. In the client, the received credential is the client's own-credential (since no up-stream method request drove the current thread of execution).

Invocation-credential

The invocation-credential is the credential that accompanies any further down-stream method requests made from this thread of execution. In the server, when delegation is enabled, the invocation-credential is automatically set to the received-credential.

Own-credential

The own-credential is also known as the process' default-credential. This credential identifies the principal associated with the process. In the server, this is the server-principal. In the client, this is the client-principal. Note, that in the server, the own-credential can become the invocation-credential when delegation is disabled

To perform this procedure you need to decide which credential you want, the security run time must be installed and the ORB must be initialized.

To acquire a credential on a thread of execution, use the following steps:

1. Obtain a security Current object: Get a security Current object.
2. Acquire the desired Credential: Use the `SecurityCurrent::received_credentials` attribute, or the `SecurityCurrent::get_credentials()` method to get the credentials. Use the former attribute if you want the received-credentials. Use the latter method if you want the invocation- or own-credentials.
3. If acquiring received-credentials, select the first credentials from the credentials list: The `SecurityCurrent::received_credentials` attribute returns a `CredentialsList`. Component Broker only carries one received-credentials. The received-credentials is in the first position of the credentials list.

Acquire the **received-credentials** using the following steps:

```

#include <IExtendedSecurity.hh>
#include <CBSeriesGlobal.hh>
// Declare a reference to the CORBA::Current, SecurityCurrent,
// and Credentials and CredentialsList objects.

CORBA::Object_var object;
SecurityCurrent_var securityCurrent;
SecurityLevel2::CredentialsList_var receivedCredentialsList;
SecurityLevel2::Credentials_var receivedCredentials;

// Use the CBSeriesGlobal::orb to get a CORBA::Current

object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SecurityCurrent");

// Narrow to the SecurityCurrent.

securityCurrent = SecurityCurrent::_narrow(current);

// Get the received-credentials list

receivedCredentialsList = securityCurrent->received_credentials();

// Pull out the received-credentials from the first (0th) position
receivedCredentials = receivedCredentialsList[0];

```

Acquire the **invocation-credentials** using the following steps:

```

#include <IExtendedSecurity.hh>
#include <CBSeriesGlobal.hh>
// Declare a reference to the CORBA::Current, SecurityCurrent,
// and Credentials objects.

CORBA::Object_var object;
SecurityCurrent_var securityCurrent;
SecurityLevel2::Credentials_var invocationCredentials;

// Use the CBSeriesGlobal::orb to get a CORBA::Current

object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SecurityCurrent");

// Narrow to the SecurityCurrent.

securityCurrent = SecurityCurrent::_narrow(current);
// Get the invocation-credentials

invocationCredentials = securityCurrent->get_credentials(
    Security::InvocationCredentials);

```

Acquire the **own-credentials** using the following steps:

```

#include <IExtendedSecurity.hh>
#include <CBSeriesGlobal.hh>
// Declare a reference to the CORBA::Current, SecurityCurrent,
// and Credentials objects.

```



```

CORBA::Object_var object;
SecurityCurrent_var securityCurrent;
SecurityLevel2::Credentials_var ownCredentials;

// Use the CBSeriesGlobal::orb to get a CORBA::Current

object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SecurityCurrent");

// Narrow to the SecurityCurrent.

securityCurrent = SecurityCurrent::_narrow(current);

// Get the own-credentials

ownCredentials = securityCurrent->get_credentials(
    Security::SecOwnCredentials);

```

Access control

Component Broker, in this release, supports a coarse-grained access control model and a fine-grained access control model. The access to business objects is controlled at the server level based strictly on whether the client principal has been authenticated in the coarse-grained access control model. More specifically, the enterprise determines whether your business object must be secured or not. If so, an enterprise administrator should configure the server where your objects reside to be secure using the Component Broker system management facility.

Any objects created in a secure server are protected. The client principal must be authenticated prior to being allowed to invoke method requests against objects in the secure server. The same policy applies to all principals for any object in the server. The fine-grained access control model in Component Broker allows the enterprise administrators to control access to the individual method on each object. The fine-grained access control model is provided by the Component Broker authorization service which is based on the IBM Policy Director authorization service, and so fine-grained access control can only be enabled if the IBM PolicyDirector has been installed in your configuration. In fact, while the fine-grained access control model we're introduced here is much finer-grained than what we had in the coarse-grained access control model, it is just one refinement of control. The fine-grained access control model, for example, will not allow the customers to define their own required-rights on individual method (beyond those that are assigned by default by the system), nor will it allow the customer to apply different authorization policies to different object instances of the same class of object. Conversely, it is sometimes useful to simply control who can access the objects in a given server.

Finer grained control over who can access functions in your object can be obtained by acquiring the credentials of the principal on whose behalf the request is being made, and testing that against access policies that are relevant to your application. For instance, if you are developing a claims-inquiry function, you might want to test for whether the person requesting the inquiry is a beneficiary of the claim. If so, then you might allow the requestor to receive personal information about the claimant that you do not allow an adjuster to see.

Other security considerations for the server

Component Broker can protect your business objects in a secure server by authenticating any access to them. However, this protection can be undermined if rogue or untrusted software is loaded in to the server. Since the implementations for business objects are captured in dynamic link libraries (DLLs), then an attacker can subvert the security of the system by changing or superceding the implementation of your business objects, perhaps handing out valuable information to other, unauthorized parties by replacing your DLLs.

To prevent this, you should protect the file system in which DLLs and EXEs (in general, any executable) are stored, and prevent the server from loading DLLs from anywhere but that protected file system. The LIBPATH for your server should be set to only those directories containing the DLLs that you want loaded in your server. This includes the Component Broker server EXE, the Component Broker system DLLs those that are needed to run the server process, and your business object DLLs.

▶ WIN

The default locations for Component Broker DLLs are the following directories:

- \Program Files\IBM\ComponentBroker\bin
- \OPT\Digital\dce\bin

▶ AIX

The default locations for Component Broker shared library files are the following directories:

- /usr/lpp/CBServer/bin
- /usr/lpp/CBServer/lib
- /usr/lpp/dce/bin

The system administrator may elect to install these files elsewhere. In any event, you should set the access controls on the directories containing the

Component Broker DLL files so that only the servers can read files in those directories, and so that only selected administrators can put DLLs or other executable files in those directories.

Depending on how you have implemented your business objects, if you make use of other data files that affect the way your business objects perform their tasks, such as configuration files or policy rules files, then an attacker could modify these types of files as well to subvert your business objects. The above principle regarding the protection of the file system should be applied to any directories that contain data files that affect the behavior of your business objects.

Authentication and end-users

Security systems are generally composed of several related concepts. This includes things like authorization, confidentiality, auditing and administration. While all security concepts are fundamentally built on the foundations of a cryptographic system, most security concepts are preconditioned by establishing who end users are, and certifying they are who they claim to be. Certifying that a person is authentic is known as authentication. Authenticated end users have identities. Those identities can be used to determine what resources the end-user is allowed to access, and how. Message traffic can be encrypted in a way that only an authenticated end-user can decipher. Identities of anyone challenging the integrity of the security system can be obtained.

Authentication of a person can be based on a variety of techniques that challenge what that person possesses, what they know, or who they are, or any combination of the three. For instance, you can assert who a person is by the fact that they possess a key to your house or employee-badge to your enterprise. This is typically a weak authentication scheme, as it can be easily defrauded by stealing the possession. The strongest technique is to base authentication on something that biometrically defines a person such as a finger-print, voice-print, DNA-print, or signature. Typically biometric-based authentication can be very reliable, but also somewhat intrusive. Biometric-based authentication schemes also typically require specialized hardware. The middle road is to base authentication on a secret that only that person should know, such as a password or your mother's maiden name. This is only as strong as it is difficult to guess the secret. However sufficiently obscure passwords can be reasonably difficult to guess making this technique quite strong. The first release of Component Broker supports password-based authentication relying on a secret that only that end-user should know.

When a User is not an End-User

Not all users are in fact people. More specifically, not all activity in the system is performed strictly on behalf of a specific person. Other entities in the system need to be authenticated too.

For instance, before using a particular server, you might like to be certain that the server is legitimate, that it is the server you expect to be working with. Component Broker provides support for authenticating servers certifying that the server is what it claims to be. To achieve this, server processes are given identities and are required to prove themselves to the system. Obviously, biometric or possession-based authentication schemes would not be very practical, and so servers are expected to authenticate themselves based on a password. Administration of a user's identity and password is discussed further in "Server key-tab file". For any method request between a client process and a server process (or between server processes) Component Broker attempts to authenticate both the the client to the server, and the server to the client. This is known as mutual-authentication.

User IDs and passwords

Authentication is based on user-identities (user-ids) and passwords. The end-user is expected to enter a user-id and a password to log-in to the system. If the password is valid for the given user-id, then the user is deemed to be authentic. Component Broker offers a variety of ways of supplying the user ID and password (logging in), including the following:

- "Logging in with environment variables" on page 236
- "Key-tab File" on page 233

Server key-tab file

A key-tab file can be used to store a user-id and password for a particular principal. While this can be used for client-principals (end-users), it is normally used to identify server-principals. A key-tab file is a convenient mechanism for storing log-in information; information that only a server should possess in order to authenticate itself to the security system.

When placed in a secure file system and protected with restrictive access policies, the key-tab file itself can be maintained safely. The policies governing access to the key-tab file should be set so that only the server's identity (the local identity under which the server runs) can access the key-tab file. If the server is activated automatically, then the local identity of the server is the same as the ORB daemon that starts it. If the server is started manually by an administrator, then the server's local identity is the identity of the administrator that starts it.

As the log-in information is kept in a well-known place, the server can be authenticated automatically. Also, the log-in information can be automatically updated thus helping to preserve the integrity of the log-in information and at the same time facilitating server automation. (See the *DCE Administration Reference*, particularly the subcommands of the **rgy-edit** Security Service commands for more information on key-tab file management.)

Key-tab File

Establishing a server's identity can most often be accomplished with the use of a key-tab file. The key-tab file is essentially used to retain the server's identity and password. By being isolated in this way, it can be administered more easily: for instance, the password can be automatically changed periodically to decrease the vulnerability of the server's identity.

However, since this vital information is stored in a file it must be protected to prevent other user's of the server host from accessing it and exploiting it for unauthorized purposes. This is achieved by putting the key-tab file in a protected file system, and setting the access controls on the file so that only the server can access it. This in turn requires that the server be started by an administrator or daemon process whose operating-system identity, inherited by the server, has the authority to access the file. The inherited operating system identity is used to provide access to the key-tab file from which the server's distributed system identity is established.

Each server on the same host can have its own entry in the key-tab file and establish its own distributed system identity. Nonetheless, any server started by the server activation daemon inherits the same operating system identity. The key-tab file should be set so that the daemon's operating system identity can access the file. This form of authentication is only exercised if this option has been enabled in the configuration, and the key-tab file has been created and can be accessed by the server process. This option can also be used by client processes and is subject to the same conditions and procedures as server processes. The server must be authenticated from a key-tab file.

Protecting the key-tab file

This procedure demonstrates how you can protect your server's key-tab file from unauthorized tampering. A key-tab file can be used for authenticating a server without requiring a local administrator to log-in for the server. Thus the key-tab contains sensitive security information, specifically the server's user-identity and password. It is essential for the integrity of the server that this file be protected.

The primary means of protecting the key-tab file resides with the file system where it is stored. The file permissions for the key-tab file should be set so that only the server can access it.

The server actually assumes the local operating-system identity of either the administrator that manually started the server, or more often, the identity of the ORB daemon that started the server automatically. In turn, the ORB daemon assumes the local operating system identity of either the administrator that manually started the ORB daemon (somorbd.exe), or more likely the Component Broker system management agent that started the ORB daemon. Finally, the system management agent assumes the local operating-system identity of the administrator that manually started the agent (bgmain.exe and bgsrvctl.exe), or the local operating-system identity of the administrator registered with Component Broker in the NT Registry, if the Component Broker is started automatically at system start-up.

Thus, the local-operating system identity of your server depends on your configuration and the process you use for starting the server. Most often, it assumes the identity of the administrator registered with Component Broker in the NT Registry. However, it could be the identity of another administrator if portions of the Host system are started manually. It is up to you to resolve which administrator's identity is used ultimately to start the server.

Knowing the local operating system identity under which your server runs is important because it is this identity that you must enable to access the key-tab file. Any other identity that you enable to this file has the ability to see and possibly change the DCE user ID and password for the server. Thus, the permissions for the key-tab should be as restrictive as possible.

Component Broker automatically creates a single default key-tab file per host. The server principal information for every server on that host is entered into that key-tab file. Thus, if different servers on the host are started under different local operating systems (OS) identities, then each of these identities needs to be enabled to access that key-tab file.

If this is unacceptable, that is if you have some servers that are started manually by different administrators and you do not want them to all have access to the default key-tab file, then you must use the DCE administration tools (rgy_edit) to create a unique key-tab file for that server and protect it separately. In doing so, you need to supply the name of the server for which an account should be created in the DCE user registry. Component Broker establishes the principal name for the server; you can obtain this from the Component Broker system management facility, but the corresponding account for the server in the DCE user registry should be created automatically. In addition, you need to modify the Component Broker system

management configuration properties for that server to indicate the name of the key-tab file so that Component Broker can find it to log-in the server during server start up.

In Windows/NT, files can only be protected if they are installed in the NT file system (NTFS).

► WIN

Protect the server key-tab file using the following steps:

1. Log in to the host machine. Sign on to the host machine where your server resides, either as the administrator that normally starts the host machine, or the administrator that manually starts the part of the Component Broker whose local operating system identity is assumed by the server.
2. Create the key-tab file. If necessary, create a unique key-tab file for the server. Otherwise, assume the default key-tab file, `v5srvtab`.
3. Use explorer to navigate to the key-tab file and set the permissions:
 - a. From the Windows NT Start menu, select **Programs > Windows NT Explorer**.
 - b. In NT Explorer, select: **C: > Opt > Digital > dcelocal > krb5**.
 - c. Right mouse click on the key-tab file, **v5srvtab** or the key-tab file you created in step 2.
 - d. Select **Properties**.
 - e. Select the **Security** tab.
 - f. Click **Permissions**.
 - g. Ensure only System and the group representing your administrator (that is, Administrators) has Full Control (All). Everyone should only have Execute (X). All other user groups and users should have No Access, or be removed from the ACL.
 - h. Select **OK** to accept the file permissions.
 - i. Select **OK** to close the Properties window.

► AIX

Protect the server key-tab file using the following steps:

1. Log in to the host machine. Sign on the host machine where your server resides as root.
2. Create the key-tab file. If necessary, create a unique key-tab file for the server. Otherwise, assume the default key-tab file, `v5srvtab`.
3. The key-tab file should already have the correct permissions. DCE on AIX sets the permissions automatically. If necessary, change directory (`cd`) to

/krb5, and use **chown** to set the owner to your administrator's ID or root, and use **chmod** to set the permissions so that only the owner can read or write the key-tab file.

Logging in with environment variables

You can specify the user ID, password, and cell in the environment variables *SCSPRINCIPAL*, *SCSPASSWORD* and *SCSCELLNAME*, respectively. The ID, password and cell values only have significance in the scope established for these variables. Depending on how these variables are created, the scope can be simply just the client process they are created in, or it can be any client process on the client machine, unless specifically modified in an individual process. These variables are then used by Component Broker to automatically log-in the specified user for any requests that are initialized from that client. This mode of log-in is only exercised if this option is enabled in the configuration, and the variables have been created and assigned a value in the client environment.

Note: This approach to logging in a user can be somewhat vulnerable to attack. A common practice is to use this option by setting the environment variables in a shell-script. If you hard-code the user-id and password variables in the shell-script, not only are you exposing the values to anyone else who has access to the file-system on which the script is stored, but if someone else is allowed to invoke the script they can use it to masquerade as the person identified in the shell script. Caution should be taken when using this log-in approach.

Using environment variables to establish authenticity

This procedure demonstrates how you can create a credential for a principal by supplying the principal's userid and password in environment variables. The values supplied in these variables are used to log-in the specified user when a credential is first required, when this option has been enabled in the configuration of the process.

The credentials produced for the principal specified in the environment variables are always local to the process where the log-in occurs. However, the same environment variables are used in all processes that fall within their scope. For instance, if you set these environment variables in the start-up script for the machine (for example, *config.sys*) then they are available to all processes started on the machine, unless explicitly nulled out in specified processes before they are used to log-in.

Note: This process is useful for certain circumstances such as when a shell-script is used outside of the application program to perform a log-in of the principal. However, this approach should be used with some care to avoid exposing the userid and password to potential

attacks. For instance, it may be dangerous to set the environment variables from hardcoded values in a shell-script. Shell-scripts are stored in a file and therefore can be attacked if the shell-script file is not protected in a secure file system.

To log in to Component Broker, you need a user ID that has been registered with DCE. You should use the DCE administration tools to create an account for the corresponding user ID. DCE must be installed.

The following steps demonstrate how to create a credential for a principal by supplying the principal's userid and password in environment variables:

1. Set the SCSRPCPRINCIPAL environment variable with the userid of the principal you want logged-in. The userid must have been registered with the DCE user registry.
2. Set the SCSPASSWORD environment variable with the password of the principal you want logged-in. The password must be valid for the principal identified in SCSRPCPRINCIPAL.
3. Set the SCSCELLNAME environment variable with the cell in which you want to authenticate the principal specified in SCSRPCPRINCIPAL. Component Broker supports a single cell, and so the SCSCELLNAME should be set with the cell used for the Component Broker installation.

The following example is an excerpt from an NT command file. This example specifies the user ID and password information for Jane Austin (user ID jaustin) in the Winchester cell.

```
set SCSRPCPRINCIPAL=jaustin
set SCSPASSWORD=sensible
set SCSCELLNAME=winchester
```

The actual log in of the principal specified in the environment variables occurs well after the variables themselves are set. No error indication occurs when these variables are set, even if the principal name, password, or cell name are invalid. The first indication of this type of error generally occurs on the first method request initiated (or received) by the process and is reported back to the application with a NO_PERMISSION exception on the request. Any errors that occur while logging-in the user are reported in the Component Broker error log. If you suspect an error has occurred during log-in, perhaps due to the principal or password supplied in the environment variables being incorrect, then you should check in the Component Broker error log.

Message protection

Component Broker authentication services can provide you protection from unauthorized access to your server. However, you may also need to protect the method requests themselves as they flow over the network. This is

particularly true in untrusted networks where the network flows over unguarded or publicly accessible wires. For instance, it is relatively easy to tap into a local area network (LAN) to monitor network traffic. An attacker can attach a monitor to see the content of every message. If your method requests pass monetary, personal, or business information that is sensitive, this can be viewed and even changed by the attacker. Component Broker provides different qualities of protection (QOP) for message traffic between client and server processes, including the following:

Integrity Protection

The message is digitally signed. Component Broker verifies the signature of the message to ensure that no one has modified the content of the message en-route. This can prevent the message from being tampered with, but does not prevent an eavesdropper from viewing the message.

Confidentiality Protection

The message is encrypted. Component Broker encrypts the message to ensure that no one can view the message en-route. This can prevent the message from being read, but it does not necessarily prevent the message from being modified, even if only to damage the message by setting it with an unencrypted value.

Confidentiality and Integrity Protection

The message is encrypted and digitally signed. This can prevent the message from being read or modified.

None (Out-of-Sequence and Replay Protection)

The message is neither encrypted nor digitally signed. This leaves the message vulnerable to being read and modified. However, as with all of the previous modes, the message is always out-of-sequence and replay protected. That is, Component Broker always assigns a secure-sequence number to all messages and detects when two messages are received out of order or if the same message is replayed. This prevents an attacker from reordering method requests so that, for instance, a withdrawal is processed before a deposit, when in fact the deposit was intended to be processed before the withdrawal. Likewise, this protection prevents an attacker from sending the same withdrawal method twice.

Out-of-Sequence and Replay protection is performed on all messages to a secure server.

Message protection is performed using a *session key*, which is exchanged as part of creating a secure association between a client-principal and a server. These session keys are always unique to every client-principal and server pair, and so different keys are used even if method requests are passed between two servers operating on behalf of different client-principals. All message

confidentiality is performed with the Data Encryption Service (DES) using a 56-bit key (plus 8-bit parity). All message integrity is performed with MD5.

The desired quality of protection is set at the secure server as a configuration option in the server image. The selected quality of protection specified for the server is used for all method requests invoked on objects in that server, irrespective of where the requests originate.

Note: Both integrity and confidentiality qualities of protection make use of a cryptographic algorithm to sign or encrypt the message. If you select either of these qualities your system performance is affected and the rate at which you can invoke methods decreases. Encryption protection affects performance more than integrity. The extent to which your system is affected depends on the size and frequency of method requests that you perform in your application and the overall workload on your client or server machine. You should evaluate this effect in any risk/benefit analysis you perform.

Delegation

Delegation is a process of a principal acting on behalf of another principal. In Component Broker, delegation is done by transferring credentials of a requesting client between server processes. In the current release, only “simple-delegation” (impersonation) is supported. It means that every downstream method request on a secure server is performed strictly under the credentials of the requesting client.

Enabling delegation for a secure server

This procedure demonstrates how you can enable delegation for a secure server. When delegation is enabled, every downstream method request on the server is performed strictly under the credential of the requesting client. If delegation is disabled, the request is then performed under the credential associated with the process.

To enable delegation for a secure server, complete the following steps.

1. Define a new server group or a new free-standing server, or select an existing one. For a new server group or server, use the Component Broker system management facility to define the new server group or server. If you are enabling an existing group or server to be secure, locate the corresponding server in the system management facility.
2. Open the Properties notebook and tab to the Security Service page.
3. Enable Server delegation. Edit the delegate credentials property by setting it to simple.

Note: In the WLM environment, you should enable delegation for the controlled server group that you have defined. This ensures that every downstream method request from the server Group Control Point server is performed under the credential of the requesting client.

Security Service objects

The Component Broker Security Service introduces the following object types:

Object Type	Related Interface(s)
"Principal object"	CORBA::Principal
	IExtendedSecurity::Principal
"Credentials object"	SecurityLevel2::Credentials
	IExtendedSecurity::Credentials
"Current object"	SecurityLevel1::Current
	SecurityLevel2::Current
"LoginHelper object" on page 241	IExtendedSecurity::LoginHelper

Principal object

Important: Please do not use the principal object, it will be removed in a future release.

Credentials object

A Credentials object represents a principal that has been authenticated to Component Broker. The SecurityLevel2::Credentials interface is introduced by the CORBA Security Service, and includes the following:

- Credentials copy()
- get_security_features(direction)
- get_attributes(attributes)
- is_valid(expiry_time)
- refresh()

Component Broker combines the CORBA::Principal and the SecurityLevel2::Credentials interfaces in the IExtendedSecurity::Credentials interface.

Current object

The Current object provides your application access to the Security Service context associated with any given thread of execution. You can access any of the credentials associated with the thread of execution through a Current

object. The CORBA Security Service introduces SecurityLevel1::Current, and SecurityCurrent, which inherits from the SecurityLevel1::Current interface.

The SecurityCurrent interface adds the following methods:

- received_credentials()
- received_security_features()
- principal_authenticator()
- get_credentials(cred_type)

LoginHelper object

The LoginHelper object can be used to log-in a user programmatically. The IExtendedSecurity::LoginHelper interface introduces the following method:

```
SecurityLevel12::Credentials: request_login(security_name,  
                                             realm_name,  
                                             password,  
                                             creds,  
                                             auth_specific_data)
```

Security and servlets in WebSphere Enterprise

One of the primary clients to WebSphere Enterprise Servers is expected to be servlets. Servlets will leverage the WebSphere Java client and can be expected to do normal client tasks such as creating, finding and using business objects within a WebSphere system. In this scheme, servlets function as the middleman by translating HTTP requests from a user's browser to IIOP requests to downstream WebSphere Enterprise servers. Inside the servlet, the Java Client is used to flow IIOP requests to the WebSphere Enterprise Servers.

This section is focused on providing a description of using SSL security to protect the requests and responses between a servlet using the Websphere Java client and WebSphere Enterprise servers. In particular we will focus on the storing the client's DCE userid and password in a credential and associating that credential with a session.

Note: Servlets are not supported in the WebSphere Enterprise DCE Java client because the WebSphere Enterprise DCE Java client is not thread safe.

Configuring the WebSphere Enterprise Servers

Install and configure the WebSphere Servers for SSL security as described in "Administer Security in your Enterprise" chapter of the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*. For testing purposes only, we have provided a sample SSL keyring file called CBDevTest.kdb which contains a private key and a self-signed server certificate. You can use this keyring file or build your own keyring file and obtain your own server certificates. Building your own keyring is described in

the section, Create and Install Server Certificates, of the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*. We strongly suggest that you follow the procedures for obtaining your own server certificates when you deploy your business objects.

Note: For public-private key technology to be effective, each server should have it's own unique public-private key pair, and therefore, each server should have it's own SSL keyring file containing a certificate which uniquely identifies itself. Using the sample keyring file, *CBDevTest.kdb*, when you deploy your application is strongly discouraged as each WebSphere Enterprise Server will essentially have the same identity. This represents an unsecure environment and should be avoided.

Configuring the Java client servlet

Install the WebSphere Java Client on your Web Server machine and configure it for SSL security as described in the "Enable Java Client with SSL Security" section of the *WebSphere Application Server Enterprise Edition Component Broker Planning, Performance and Installation Guide*.

Installing the WebSphere Java Client adds the following two files, which are required by the servlet (since it is functioning as a WebSphere Java client):

somojd.zip

The domestic version of the SSL security java implementation zip file.

somojse.zip

The export version of the SSL security java implementation zip file.

CBDevTestCIKeyRing.class

The sample Java class with the client-side keyring for SSL support. It contains the server certificates from the *CBDevTest.kdb* file which you might have used to configure SSL on the server. This Java class will be accessed by the servlet, since it is acting as the "Java Client" in this configuration. If you have obtained your own server certificates, you will have to create your own Java keyring class and import the certificates for each server.

By default, the WebSphere Java Client SSL service displays a prompt to ask for the client's DCE userid and password to log the client in at the WebSphere Enterprise Server. While this default might work fine for a client workstation, this is not what we want when running the Java Client inside a servlet. Therefore, to prevent this dialog from appearing, change the *login source* attribute of the Client Style from "prompt" to "none". You can then use the properties file generated from this Client Style as the value of the `com.ibm.CORBA.ClientStyleImageURL` property.

Another option is to directly edit the properties file you want to use and change the following line:

```
com.ibm.CORBA.LoginSource=prompt
```

to

```
com.ibm.CORBA.LoginSource=none
```

Installing and configuring servlets in WebSphere Enterprise

After you've installed the servlet, you will also have to do the following to configure the WebSphere Java Client to run as a servlet:

- Update the Application Server Classpath to include two of the following files:
 1. `somojsd.zip` *or* `somojse.zip`
 2. `somojor.zip`
- The `somojsd.zip` *or* `somojse.zip` file should precede the `somojor.zip`.
- Add the “-nojit” option to the Java Compiler Directives.

Example code source

There are several steps, illustrated by the code snippet below, to programmatically creating a credential with a DCE userid and password and associating it with a servlet session. First, the `_LoginHelper` must be instantiated after the ORB has been created with `CBSeriesGlobal.Initialize()`. When initializing the ORB, you can pass `Properties` object into `Initialize()` which can define the `com.ibm.CORBA.ClientStyleImageUrl` property to point to your properties file.

Second, you must obtain an initial DCE userid and password and form a credential before the first request flows to the Name Server. Typically, the call to `CBSeriesGlobal.nameService()` initiates the first request.

Third, you will also want to catch `org.omg.CORBA.NO_PERMISSION` exceptions on calls to methods which create a remote object or on calls to methods to get a `FactoryFinder` for the remote object. Either may result in a request flowing to that application server for the first time. The client credentials, userid and password, are sent to the application server on the first request and the user is logged in at the application server at that time.

The following code snippet represents a base servlet class which provides some common functionality to other sibling servlets.

```
//Standard java imports
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.text.*;
```

```

import java.net.*;
import java.lang.reflect.*;

//WebSphere Imports start here

// Java Client
import com.ibm.CBCUtil.CBSeriesGlobal;
import com.ibm.CORBA.iiop.ORB;

//For Factory Finder & Naming Context
import com.ibm.IExtendedNaming.*;
import com.ibm.IExtendedLifeCycle.*;

//Security
import org.omg.SecurityLevel2.*;

/**
 * Base class to support commonly used WebSphere Enterprise Servlet
 * functions. This servlet does not override any functions dispatched by
 * service(), such as doPost, doGet, etc. It is expected that you will
 * have other servlets which will override those methods accordingly.
 * such servlet might override doPost() to receive a form with a userid
 * and password and call createCredentials() below to create a credential
 * and associate it with a session.
 *
 */
public class CBHttpServlet extends HttpServlet {

//***** CHANGE THE FOLLOWING CONSTANTS FOR YOUR INSTALLATION *****

    final protected String BOOTSTRAP = "your.cb.bootstrap.server";
    final protected String CLIENTSTYLE =
        "file:<DRIVE>:/your/ClientStyle.properties";
    final protected boolean USE_SSL = true;

//*****

    // ORB Instance
    protected ORB orb;

    //Factory Finder
    protected FactoryFinder factoryFinder = null;

    //Session
    protected HttpSession session = null;

    //LoginHelper class for programmatic logins
    com.ibm.IExtendedSecurity._LoginHelper _loginHelper = null;

```



```

/**
 * Called once for initialization
 */
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    Properties props = new Properties();
    String[] args = new String[0];
    props.put("com.ibm.CORBA.BootstrapHost", BOOTSTRAP );

    if (USE_SSL)
        props.put(
            "com.ibm.CORBA.ClientStyleImageUrl","file:" + CLIENTSTYLE);

    // Initialize the Java Client environment
    try {
        CBSeriesGlobal.Initialize(args, props);
    }
    catch ( java.lang.Exception e ) {
        System.out.println(
            "exception caught from CBSeriesGlobal.Initialize, exiting...");
        e.printStackTrace();
    }

    // Save the Java Client ORB instance in a class variable for
    // future use
    orb = (com.ibm.CORBA.iiop.ORB) CBSeriesGlobal.orb();

    //
    // It is important to note that to do programmatic logins, the
    // _LoginHelper object must be instantiated and a credential
    // created for that client before CBSeriesGlobal.nameService() is
    // called. The call to nameService() flows the first request to
    // the Name Server and if no credentials are set for the user the
    // Security Service will assume that the request is intended to
    // be unauthenticated. Unauthenticated requests to a secure server
    // will unauthenticated with a NO_PERMISSION exception.
    //
    // Also the _LoginHelper must be created after ORB has been created.
    //
    //
    // The following line is provided as a convenience for those who
    // would like to instantiate a _LoginHelper during the init().
    //
    // _loginHelper =
        com.ibm.IExtendedSecurity._LoginHelperHelper._create();
}

/**
 * Process the request with the userid and password. One way this
 * method might be used is to call it from another servlet which
 * overrides doPost() and receives the form with the userid and

```

```

* password. The method returns true if the credentials have been
* created.
*/
public boolean createCredentials( HttpServletRequest req )
{

    if (USE_SSL) {

        // Create a _LoginHelper if we don't have one already.
        if( _loginHelper == null ) {
            _loginHelper =
                com.ibm.ExtendedSecurity._LoginHelperHelper._create();
        }

        String[] values;
        String userID = null;
        String password = null;

        //Get the Session object
        session = req.getSession(true);

        //Retrieve the userid from the form fields
        values = req.getParameterValues("userid");
        if (values != null) {
            userID = values[0];
        }

        values = req.getParameterValues("password");
        if (values != null) {
            password = values[0];
        }
        System.out.println("login: " + userID + "-" + password);

        //Create the credentials and set the session data value
        try {
            org.omg.SecurityLevel2.Credentials myCreds =
                _loginHelper.request_login(userID,
                    (String)null,
                    password,
                    (org.omg.SecurityLevel2.CredentialsHolder)null,
                    (org.omg.Security.OpaqueHolder)null );

            // Put credentials in session object for future use
            session.putValue("your.package.or.class.name.credentials",
                myCreds);

            return true;
        }
        //
        // Note that when using SSL, request_login() will fail only if the
        // password is provided but no userid is provided. The userid and
        // password are passed to the WebSphere Enterprise Server on the
        // first method request and are validated at the server at that
        // time, not during the request_login method. An example of how
        // to catch the NO_PERMISSION exception is shown in

```

```

        // getFactoryFinder().
        //
        catch ( org.omg.SecurityLevel2.LoginFailed lf) {
            System.out.println(
                "request_login() failed with " + userID + " - " + password);
            return false;
        }
    } else {
        return true;
    }
}

/**
 * The credentials for a user were stored in that user's session object
 * by the login servlet. This method retrieves them from the session
 * object and uses them as input to the SecurityLevel2.Current object.
 * Setting them in the Current object causes the Java client ORB to flow
 * them on subsequent method requests.
 */
public boolean setCBCredentials(
    HttpServletRequest req, HttpServletResponse res)
{
    boolean ret = false;

    if (USE_SSL) {
        session = req.getSession(true);
        //create session if not already created

        //The credentials are retrieved from the loginServlet
        org.omg.SecurityLevel2.Credentials myCreds =
            (org.omg.SecurityLevel2.Credentials)session.getValue(
                "your.package.or.class.name.credentials");

        //Get the current security level
        org.omg.SecurityLevel2.Current sL2Curr = null;

        try {
            sL2Curr = org.omg.SecurityLevel2.CurrentHelper.narrow(
                orb.resolve_initial_references("SecurityCurrent") );
            if (sL2Curr != null) {
                sL2Curr.set_credentials(
                    org.omg.Security.CredentialType.
                    SecInvocationCredentials, myCreds );
                ret = true;
            }
        }
        catch (Exception exc) {

            // Some problem occurred while trying to set the credentials.
            // Post an appropriate error to the user and proceed
            // accordingly.
            //

```

```

        // ...
    }
} // if use_ssl
else {
    ret = true;
}

return ret;
}

/**
 * Get the FactoryFinder. The first request to the Name Server
 * flows on CBSeriesGlobal.nameService(). The _LoginHelper and
 * client's credentials should have been created before this
 * method is called. If no credentials are set for the user, the
 * Security Service will assume that the request is intended to be
 * unauthenticated. Unauthenticated requests to a secure server
 * will typically fail with a NO_PERMISSION exception.
 */
public FactoryFinder getFactoryFinder()
{
    org.omg.CORBA.Object obj = null;
    NamingContext iExtendedNC = null;

    //If null, try to obtain it.
    if (factoryFinder == null) {
        try {
            iExtendedNC = CBSeriesGlobal.nameService();
            obj = iExtendedNC.resolve_with_string(
                "host/resources/factory-finders/host-scope");
            factoryFinder = FactoryFinderHelper.narrow(obj);
        }
        catch (org.omg.CORBA.NO_PERMISSION np) {

            // The DCE login at the Name Server failed. One option is
            // to post an html page which explains that the login
            // failed and ask for the userid and password again.
            //
            // ...

        }
        catch (Exception e) {

            // Some other problem occurred. Post an appropriate
            // error to the user and proceed accordingly.
            //
            // ...

        }
    }
    return factoryFinder;
}

```

```
//  
//  
// Other methods common to sibling servlets follow.  
//  
//  
  
} // CBHttpServlet
```

Java client programming note

Each Security Java client application program has to invoke `CBSeriesGlobal::initialize()` to initialize the ORB which is the default ORB that will be referenced by the Security Service. `CBSeriesGlobal::orb()` will return the object reference of the default ORB.

Please do not invoke the `CORBA::ORB_init()` method to initialize the ORB because it will create a new instance of ORB which is different from the default ORB.

For more information of the `CBSeriesGlobal` interface, please refer to the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*.

Chapter 9. Transaction Service

The Transaction Service enables programmers to implement transactions using standard object-oriented interfaces in a distributed environment.

The provision of a Transaction Service has always been an essential part of the software for mainframe systems because these systems run business critical applications. Now that such applications are also being written for a distributed environment, the software in this environment must also support a Transaction Service.

One of the features of business critical applications is that they often make a number of updates (changes) to the data to complete a task. At the same time, a number of different tasks are normally running that are also making updates to this data. The most difficult part of writing a business application is to keep the updates that belong to a single task together, while keeping them separate from the updates being made by other tasks. This is hard enough when the system is running properly; once you start considering all the different ways the system may completely or partially fail, the effort required to design and write the application becomes truly daunting.

This is where the Transaction Service is useful. The objects within an application group the updates required for a single task into a transaction and the Transaction Service ensures that either all of these updates occur or none of them do (this is called atomicity). Provided that the application has correctly grouped the updates in the transaction, then the data is always updated consistently. If the application uses the Transaction Service in conjunction with the Concurrency Service these updates are also not affected by updates being performed for other tasks. Finally, if persistent objects, or a database, are used to store the data, these updates will be permanent (durable) even if the system crashes.

An example of a transaction

Suppose you have two bank accounts, *A* and *B*, with balances of 300 and 100 dollars, and that you want to transfer 50 dollars from account *A* to account *B*. You might do this as follows:

1. $A = A - 50$
2. $B = B + 50$
3. Result: $A = 250$, $B = 150$

The result is correct. Suppose, though, that an error such as a system crash occurs just after account *A* is updated. If you are not using a transaction, the following would occur:

1. $A = A - 50$
2. System crash and restart
3. Result: $A = 250, B = 100$

Account *A* has been correctly debited but, because processing was interrupted by the system crash, the corresponding credit to account *B* has not been made. Apparently, 50 dollars have been lost. You can avoid this type of problem by using transactions.

You can implement the same bank transfer as a transaction:

1. Begin transaction
2. $A = A - 50$
3. $B = B + 50$
4. Commit transaction
5. Result: $A = 250, B = 150$

As expected, the result is correct. More importantly, if a system crash occurs after *A* is updated, the result is still correct:

1. Begin transaction
2. $A = A - 50$
3. System crash, system restart, transaction recovery
4. Result: $A = 300, B = 100$

This time, because the transaction was not committed, any changes made to the accounts before the system crash are discarded and both accounts return to their original state. The transaction can then be run again to get the correct results. Similarly if *B* is updated, but the transaction is not committed, the changes are discarded.

Top-level and flat transactions

The most common type of transaction used by applications is the top-level transaction. For example, the first transaction created by an application is always a top-level transaction.

Top-level transactions are independent of one another; the updates that the application associates with one top-level transaction (see “Lifetime of a transaction” on page 253) appear as an atomic update to other top-level transactions. They are also recoverable. This means that if some (or all) of the objects associated with the transaction terminate while the top-level

transaction is still active (running) then the transaction waits until these objects have been recreated to allow them to participate in determining the outcome of the transaction.

An application can set a time limit for a top-level transaction. If it does not complete within this time limit, the transaction is said to have timed out and the Transaction Service causes it to rollback.

A top-level transaction is often referred to as a *flat transaction*.

The CORBA specification also describes another type of transaction, called a *subtransaction*, often referred to as a *nested transaction*. Component Broker does not currently support subtransactions.

Lifetime of a transaction

Applications use transactions to group related updates to data such that all of the updates occur or none do.

Typically, an application:

1. Starts a transaction.
2. Makes the updates and associates them with the transaction.
3. Terminates the transaction

When an application terminates a transaction, it can request that the transaction is either rolled back or committed. If the application requests rollback, all of the updates it has made are undone. If the application requests that the transaction is committed, the Transaction Service checks that each object involved in the transaction is able to make its updates permanent. If all objects indicate that they can, the transaction is committed. Otherwise the updates are undone just as if the application requested rollback. The result of a transaction (that is, whether it committed or rolled back) is referred to as its outcome.

Transaction scope and context

The Transaction Service allows multiple objects to participate in a transaction. These objects can be distributed across multiple operating system processes and threads and each object can be working with more than one transaction at once. To control which transaction an object is working on at a particular point in the code, the Transaction Service provides a *transaction context*. This is a collection of Transaction Service objects that represents the transaction.

The *scope* of a transaction is made up of all the locations within your application where the transaction context is in use. In general, the scope of the transaction increases over the lifetime of the transaction as the transaction context is passed from object to object.

The Transaction Service provides two mechanisms for passing transaction context:

- The most common method is *implicit propagation*, where the transaction context is associated with a thread and is available to each method called within this thread that understands transactions. If a remote method is called, the Transaction Service automatically passes the transaction context to the thread in the remote server process where the method is run.
- An alternative method is for the application to pass the transaction context as a parameter on method calls. This is called *explicit propagation* and allows access to the transaction context without associating it with the remote object's thread. Explicit propagation is not used as widely as implicit propagation because it requires Transaction Service objects to be passed as parameters to application objects and prevents the Transaction Service from exploiting its own internal performance enhancements.

Recoverability

One of the main reasons for using the Transaction Service is that it removes the need for an application to manage the correction of its data if some, or all, of the operating system processes involved in a series of updates abnormally terminate. If the application associates the series of updates in a transaction, the Transaction Service is able to coordinate the objects responsible for these updates to ensure that either all or none of these updates occur, even in the event of system failures.

To do this, the Transaction Service requires that the objects that represent the updates made to the data are recoverable. This means they are capable of surviving, and of preserving their internal state, across a system or software failure. In the Component Broker programming model, these objects must be managed objects with persistent references and persistent state. These objects must also inherit from particular CORBA defined interfaces to allow the Transaction Service to call them when it is coordinating the updates. Refer to "The Transaction Service objects and interfaces" on page 267.

Recoverability is a quality of service that is provided by certain Component Broker application adaptors. The RDB adaptor is one such example.

Transaction outcomes

When the Transaction Service is terminating a transaction, it uses the two-phase commit (“The two-phase commit process” on page 257) or one-phase commit (“The one-phase commit process” on page 258) process to ensure the objects that have made updates to data during the transaction take the same action, committing (making permanent) the updates or rolling back (undoing them). The decision that is made, and the action the objects actually take, is referred to as the *transaction outcome*.

A top-level transaction (“Top-level and flat transactions” on page 252) can have one of the following outcomes:

Commit

All of the updates have been completed successfully and the changes are now permanently recorded in the data.

Rolled back

None of the updates requested for the transaction have been made. The data is left in the same state as if the transaction had not run.

Heuristic Mixed

Only some of the updates for the transaction have been permanently recorded in the data. The rest have been undone. In an ideal world, this outcome would never occur. However if:

- an application does not implement its resource objects (“The Transaction Service objects and interfaces” on page 267) correctly, or
- one of more objects are temporarily unable to contact the other objects involved in the transaction,

They may be forced to end the transaction by taking a heuristic decision (“Heuristic decisions” on page 258) to release critical locks or resources. If this decision is inconsistent with the action chosen by the other objects then the outcome of the transaction is heuristic mixed.

In addition, if the top-level transaction is not able to contact all the objects, it can report *heuristic hazard*. This is a temporary result that changes to one of the outcomes described previously when all objects involved in the transaction are consulted.

The following table summarizes how the actions of the objects responsible for data updates within a transaction are aggregated into a transaction outcome.

Table 11. Transaction outcomes

Action of Objects						Transaction Outcome
Commit	Rollback	Heuristic Commit	Heuristic Rollback	Heuristic Hazard	Heuristic Mixed	
yes						Commit
yes			yes			Heuristic Mixed
yes			yes	yes		Heuristic Mixed
yes			yes	yes	yes	Heuristic Mixed
yes			yes		yes	Heuristic Mixed
yes				yes		Heuristic Hazard
yes				yes	yes	Heuristic Mixed
yes					yes	Heuristic Mixed
	yes					Rollback
	yes	yes				Heuristic Mixed
	yes	yes		yes		Heuristic Mixed
	yes	yes		yes	yes	Heuristic Mixed
	yes	yes			yes	Heuristic Mixed
	yes			yes		Heuristic Hazard
	yes			yes	yes	Heuristic Mixed
	yes				yes	Heuristic Mixed
		yes				Commit
		yes		yes		Heuristic Hazard
		yes		yes	yes	Heuristic Mixed

Table 11. Transaction outcomes (continued)

Action of Objects						Transaction Outcome
Commit	Rollback	Heuristic Commit	Heuristic Rollback	Heuristic Hazard	Heuristic Mixed	
		yes			yes	Heuristic Mixed
			yes			Rollback
			yes	yes		Heuristic Hazard
			yes	yes	yes	Heuristic Mixed
			yes		yes	Heuristic Mixed
				yes		Heuristic Hazard
				yes	yes	Heuristic Mixed
					yes	Heuristic Mixed

The two-phase commit process

Two-Phase commit is a protocol used by the Transaction Service during transaction termination to enable all updates to data associated with the transaction to be made permanent (committed) or undone (rolled back).

In the first phase of the two-phase commit protocol, the Transaction Service sends a “prepare” message to each of the objects representing updates to the data made during the transaction. These objects respond by voting either to “Commit”, “Rollback” or “ReadOnly”.

A vote for “ReadOnly” means the object is not interested in the outcome of the transaction. It is therefore no longer involved in the two-phase commit.

A vote to “Rollback” causes the whole transaction to roll back. This object is not called again.

A vote to “Commit” means the object guarantees that it is in a state where it can either make its updates permanent or undo them, even if the system crashes before the Transaction Service can pass a message indicating which of these action to take. Because an object that has voted “Commit” is waiting for a decision from the Transaction Service, it is said to be “in doubt”.

If the Transaction Service receives a “Commit” or “ReadOnly” vote from all objects, it sends the “Commit” message to all objects that voted commit. These objects should make their updates permanent. They are not called again.

If the Transaction Service receives a “Rollback” vote from one or more objects it sends a “rollback” message to each object that voted “Commit”. These objects should undo their updates and they are not called again.

If all resources follow the protocol correctly (and none of the servers have a transaction retry limit set), the Transaction Service guarantees that all objects take the same action so that the updates for the transaction appear atomic. However, in exceptional circumstances, it might be necessary for some of the objects involved in the transaction to take a heuristic decision that might result in a loss of atomicity. This is referred to as *heuristic damage* or a *heuristic mixed outcome*.

If only one object is responsible for data updates in a transaction, the Transaction Service can use the one-phase commit optimization (refer to “The one-phase commit process” on page 257.)

The one-phase commit process

If all of the data updates for a transaction occur in a single object, the Transaction Service can optimize the termination of a transaction by using a one-phase commit protocol with this object rather than the two-phase commit (refer to “The two-phase commit process” on page 257.) With a one phase commit, a single message is sent to the object controlling the data updates and it reports whether it committed or rolled back. This result becomes the outcome of the transaction (refer to “Transaction outcomes” on page 255.)

Heuristic decisions

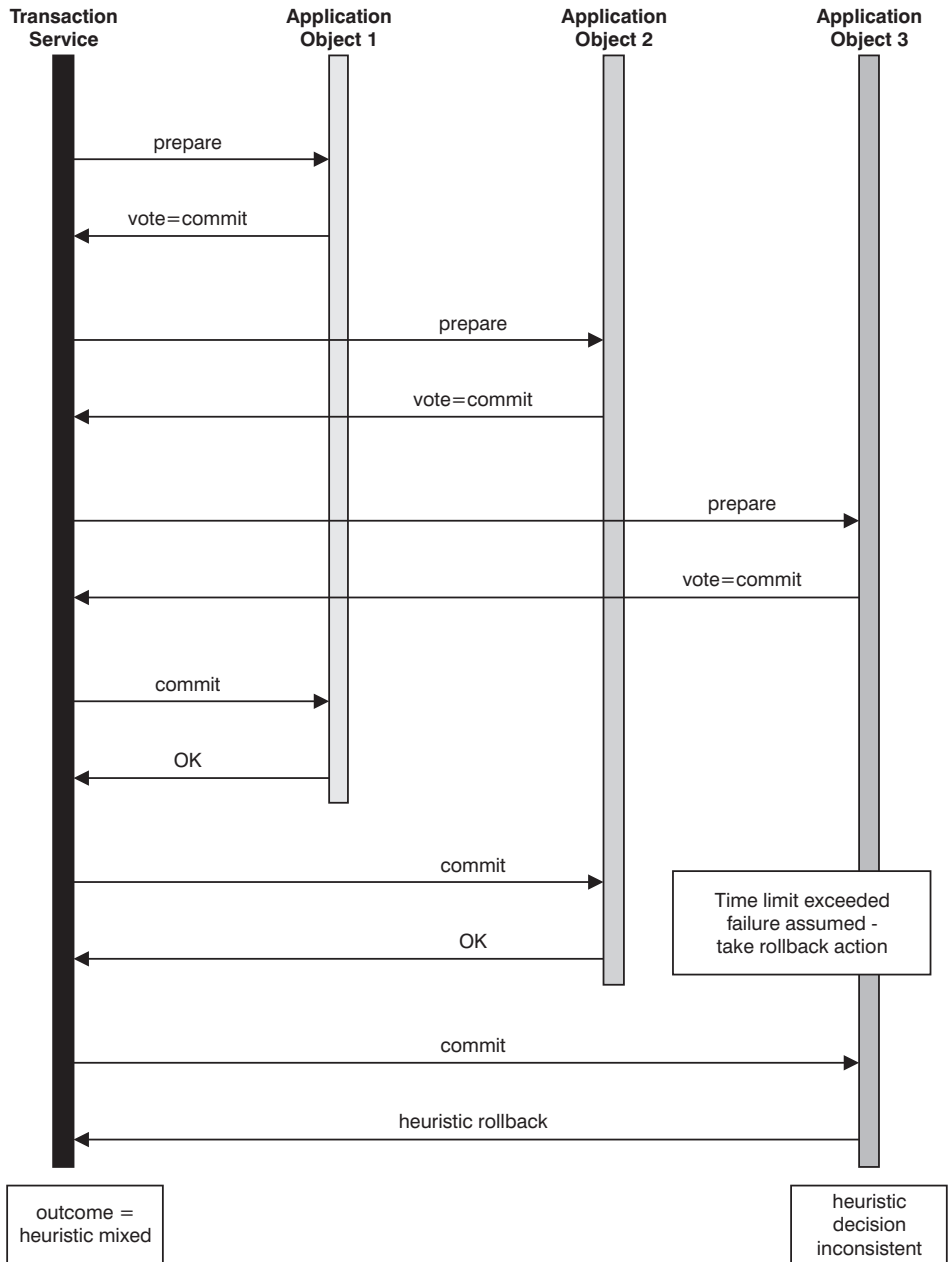
If all objects involved in making updates to data within a transaction correctly follow either the two-phase commit or one-phase commit protocol selected by the Transaction Service, and none of the servers have a transaction retry limit set, the Transaction Service guarantees that all objects take the same action during the completion of a transaction.

However, in exceptional circumstances, it might be necessary for some of the objects involved in the transaction to “guess” the outcome of the transaction in order to release critical locks or resources. This guess is called a *heuristic decision*.

If, at the end of the transaction, the heuristic decision matches the action taken by the other objects then the transaction is still atomic. However, if this guess is inconsistent with the action chosen by the other objects in the transaction, some of the updates will have been made permanent while others were undone. This is referred to as heuristic damage or a heuristic mixed outcome.

No heuristic decision is safe. Even if all objects are programmed to take the same heuristic decision on detection of a potential failure (for example a timeout), heuristic damage can still occur as it is possible that only some of the objects are affected by the failure and will take a heuristic decision.

Here is an example:



This example shows that object 3, which has voted to commit and takes a heuristic decision to roll back, can cause heuristic damage. The heuristic damage occurs because the other objects within the transaction do not detect a

failure and so do not make the heuristic decision. Therefore atomicity can only be guaranteed by the Transaction Service if heuristic decisions are not used.

Transaction retry limits

If a server containing objects that are involved in a transaction terminates unexpectedly during the two-phase commit or one-phase commit process, the outcome of the transaction cannot be resolved until the failing server is restarted. While the server is unavailable the rest of the objects involved in the transaction might be holding locks to critical resources. This could affect many users and even bring the entire system to a halt.

For some applications, an indefinite halt is a more serious problem than the possible loss of data integrity if the Transaction Service ignored the objects in the failing server. The Transaction Service can be configured to limit the number of times it attempts to contact a server during the two-phase commit or one-phase commit. If this limit is reached, the Transaction Service uses a pre-configured value as the “action” taken by the unavailable objects. This is factored into the transaction outcome as if the objects had reported it as normal.

The retry limit and the heuristic action assigned to an unavailable object are configured in each server using the *retry restricted*, *commit retry limit* and *heuristic direction* attributes. These values must be used with care as any type of heuristic decision can result in a loss of data integrity.

Transaction time limits (timeouts)

An application can set a time limit for its transactions. This time limit applies in all operating system processes and threads that are part of the transaction’s scope. It is specified as the transaction is started and runs until the call is made to terminate (commit or rollback) the transaction.

When the time limit is reached, the transaction is said to have *timed out* and if the call to terminate the transaction has not been made, the Transaction Service rolls back the transaction.

As this rollback can occur while the application is still using the transaction, it might receive responses from the Transaction Service indicating that the transaction is no longer valid because it has rolled back.

Time limits are useful mechanisms for recovering from deadlocks, or other failures that introduce delays into the transaction, especially if the transaction

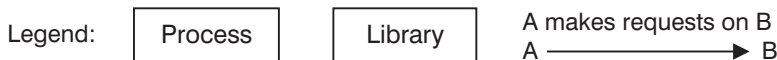
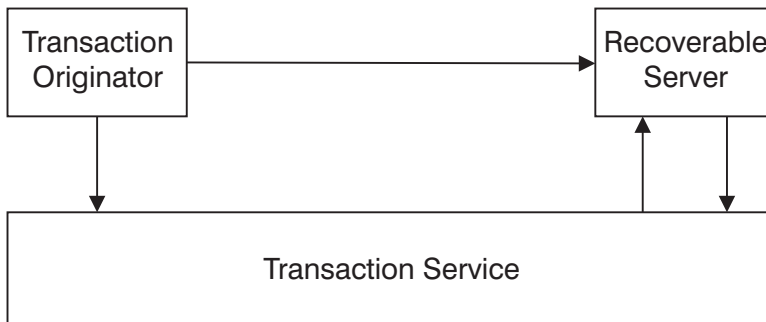
is holding critical locks. They are safe to use and do not result in heuristic decisions because the time limit is ignored once the two-phase or one-phase commit process starts.

Application programming using the Transaction Service

Architecture and design of a Transaction Service application

This section describes some suggested architectures for applications that use the Transaction Service. The Transaction Service itself places no restrictions on the architecture of applications (refer to “Chapter 9. Transaction Service” on page 251 for more information.)

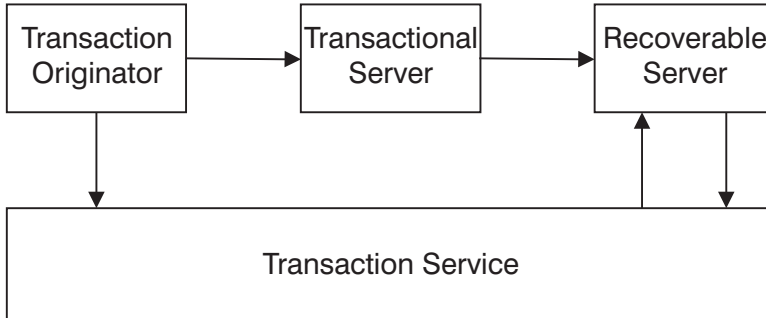
The following diagram shows the basic elements that make up a Transaction Service application. These are the transaction originator and a recoverable server. The transaction originator is the process that starts the transaction. Usually it is a non-recoverable client but it can also be a server process. Once it has started the transaction, the transaction originator invokes methods on objects in the recoverable server. These objects manage the updates to data during the transaction.



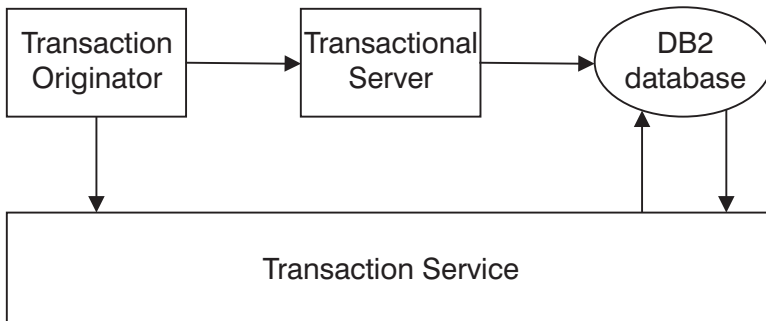
The objects containing the recoverable data need not all reside in the same recoverable server as indicated in the previous diagram. The transaction originator might invoke methods on recoverable objects residing in more than one recoverable server and these can all be involved in the same transaction.

Another alternative architecture, which is illustrated in the following diagram, involves an intermediate server that contains no recoverable data. This type of

server is called a transactional server (refer to “Transactional server” on page 265 for more information.) Typically, the transactional server would contain the business logic and the recoverable server would be responsible for data updates.



As an alternative, the recoverable server can be replaced with an XA database. The following diagram illustrates this.



The Transaction Service inter-operates with the DB2 XA Resource Manager to ensure that data contained in the database is involved appropriately in the application’s transactions.

Non-recoverable client

The Transaction Service requires the Component Broker server environment to manage transactions. However, it also offers a client library that allows other processes to access the Transaction Service. This library is called “somosai”, for the IBM VisualAge C++ compiler, and “somosam” for the Microsoft Visual C++ compiler. Component Broker also provides a Java Client SDK zip file, somojor.zip, with Java client implementations of the Transaction Service interface. Processes that use the C++ libraries or Java zip file are called

non-recoverable clients. Note that, in the case of a Java client that runs on a machine capable of also supporting a Component Broker server, the SOMOJOR.zip file must be at the start of the CLASSPATH otherwise the business object (server) Java bindings will be loaded when a CORBA Object Service interface is called and a CORBA::NO_IMPLEMENT exception will be thrown.

A non-recoverable client can use the Transaction Service objects and interfaces in the normal way to start and end transactions. It can also propagate the transaction to objects located in servers. The Transaction Service ensures that the objects it needs to be located in a server are created in the server.

The restrictions for using the client library are:

- It cannot be implemented within a Component Broker server process.
- It must be implemented in conjunction with at least one available Component Broker server process. The Component Broker server process is required to maintain transactional information about transactions started in the client.
- Non-recoverable client processes cannot receive transactional requests.

When the non-recoverable client originates a transaction, it needs to choose a Component Broker server on which to create the Transaction Service objects. The choice of this server can have a significant impact on performance. For example, if the client chooses a server that would not normally be involved with the transaction, performance will be adversely affected. To avoid this, the client will attempt to choose the same server as the first transactional object, by delaying the creation of the Transaction Service objects until needed by the transaction. In some circumstances, however, (for example, if the application asks for the name of a transaction), the client must create the Transaction Service objects before it can determine the location of the first transactional object.

When the client must determine the server before any transactional objects are used, the client uses the client style image stored in the System Management configuration data files. The *factoryFinder* attribute of the client style names the default factory finder that will be used by the client to find a transaction factory, and hence determine the server on which the transaction will be created. If no transaction factory can be found using this method, an arbitrary transaction factory will be chosen using the host-scope factory finder.

It is also worth noting that a Component Broker server cannot detect when a client process terminates. Therefore it is recommended that all transactions that are controlled from a non-recoverable client process are created with a time limit. This means that if the client process terminates unexpectedly while it has incomplete transactions running in the server, these will be rolled back

when the time limit expires. Without this safeguard, the incomplete transactions would hold any locks on your application's data until the server was shutdown.

Transactional server

A transactional server is a Component Broker server that contains objects that inherit from the `CosTransactions::TransactionalObject` interface. Thus, if these objects are called within the scope of a transaction, the object's work is considered part of the transaction.

If a server has objects that update data as part of a transaction, it is called a recoverable server rather than a transactional server even if it also contains objects that inherit from the `CosTransactions::TransactionalObject` interface. The reason for this distinction is that a server that manages updates to data has additional responsibilities when it is using the Transaction Service.

Note that in the Component Broker programming model, all managed objects inherit from `CosTransactions::TransactionalObject`.

Recoverable server

A Component Broker server may be recoverable or transactional. If the server is configured with a persistent data store such as DB2, and contains objects that manage updates to data during a transaction, then it is recoverable. If it is not configured with a persistent data store, then it is transactional.

Visibility rules

Transactions can be run concurrently from different applications, sometimes competing for the same data. Each transaction isolates the data updates it encompasses from concurrent transactions. No intermediate states of persistent data are visible outside the transaction that modifies those states. Only when a transaction has been successfully committed are the data updates encompassed within that transaction made visible outside the transaction. This is true of all transactions in Component Broker, although the manner in which data may be accessed concurrently by different transactions varies, depending on the container in which the data's managed object is configured. The behavior of components with regard to concurrent access to their state data may be categorized as follows:

DB2 Embedded SQL Components, MQ Components, and Shared Transient Transactional Components

Serialization of access to the state data of these types of components is accomplished by a concurrency lock which locks the component on a

transaction basis and is not unlocked until the transaction commits or rolls back. Other transactions are forced to wait for their turn to access the component.

DB2 Caching, Oracle, and PAA (APPC) Components

Each transaction is given its own copy of the persistent state of components of these types, and conflicts are resolved by the underlying Cache Service associated with the particular type of component. When conflicts occur, the Cache Service will pick a winning transaction and force the losing transaction to rollback. Multiple transactions can have read access to these components without causing conflicts.

Design a Transaction Service application

There are two parts of an application design that are affected by the use of the Transaction Service:

1. The objects that contain the business logic that understands how the updates to different pieces of data relate to the same user task.
2. The objects that actually perform the updates to the data.

Before carrying out this procedure, make sure you are familiar with:

- “Chapter 9. Transaction Service” on page 251
- “Lifetime of a transaction” on page 253
- “Transaction scope and context” on page 253

The following steps explain how to identify which parts of your application need to call the Transaction Service.

1. Identify all of the updates for each of the user tasks that your application performs.

Group these updates so that the data is consistent after each entire group have been performed. These form the basis of your application’s transactions.

2. Locate the points in the code where the processing starts and ends for each group of updates.

These are the places where the application should call the Transaction Service to start and end the transaction.

3. Identify all of the objects called between the start and end point of the transaction.

If they cause data to be updated for the transaction, or call other objects that update data for the transaction, they need to implement within the scope of the transaction. Methods that are called in the same thread as the transaction was started in, or in a thread with which the transaction has

been associated, are automatically included within the scope of the transaction. However, objects located in remote servers need to inherit from `CosTransactions::TransactionalObject` to inform the Transaction Service that the transaction needs to be implicitly propagated and automatically associated with the thread in the remote server where the method runs.

4. Identify the objects that actually make some or all of the updates for the transaction. Design resource objects for these objects.

See “Manage transactions in your application” on page 269 for further information.

The Transaction Service objects and interfaces

The following table shows the interfaces defined by CORBA for managing transactions. These interfaces represent the various roles and responsibilities required to make updates that are distributed across a number of servers appear atomic. Some of these roles are fulfilled by the Transaction Service and the others by the application itself, as shown in the following table. Thus the Transaction Service and the application cooperate when transactions are in use.

Table 12. Interfaces defined by CORBA for managing transactions

CORBA Interface	Implemented by
<code>CosTransactions::Current</code>	Transaction Service
<code>CosTransactions::Control</code>	Transaction Service
<code>CosTransactions::TransactionalObject</code>	Application Adaptor
<code>CosTransactions::Coordinator</code>	Transaction Service
<code>CosTransactions::Terminator</code>	Transaction Service
<code>CosTransactions::Resource</code>	Application Adaptor
<code>CosTransactions::RecoveryCoordinator</code>	Transaction Service
<code>CosTransactions::Synchronization</code>	Application Adaptor
<code>CosTransactions::TransactionFactory</code>	Transaction Service

There is a `CosTransactions::Current` object in every process. It is used by the application to start and end transactions.

When a transaction is started, the `CosTransactions::Current` object returns a `CosTransactions::Control` object. This is used by the application to represent the transaction context. The `CosTransactions::Current` object also associates the `CosTransactions::Control` object with the thread that created the transaction. This association is used to implicitly propagate the transaction context to any

object that is called in this thread if the object inherits from `CosTransactions::TransactionalObject`. (Note that in the Component Broker programming model, all managed objects inherit from `CosTransactions::TransactionalObject`.)

The `CosTransactions::Control` object also provides access to the `CosTransactions::Coordinator` object and `CosTransactions::Terminator` object created by the Transaction Service for the transaction. The `CosTransactions::Coordinator` object is responsible for keeping a record of application objects that implement the `CosTransactions::Resource` interface. These resource objects represent updates to data that the application has performed that belong to the transaction. Under the instructions of a `CosTransactions::Coordinator` object, they are able to make these updates permanent or they can undo them.

While the application is making the updates, it explicitly registers its resource objects with the `CosTransactions::Coordinator` object. This returns a `CosTransactions::RecoveryCoordinator` object for use by the resource object if the server process fails.

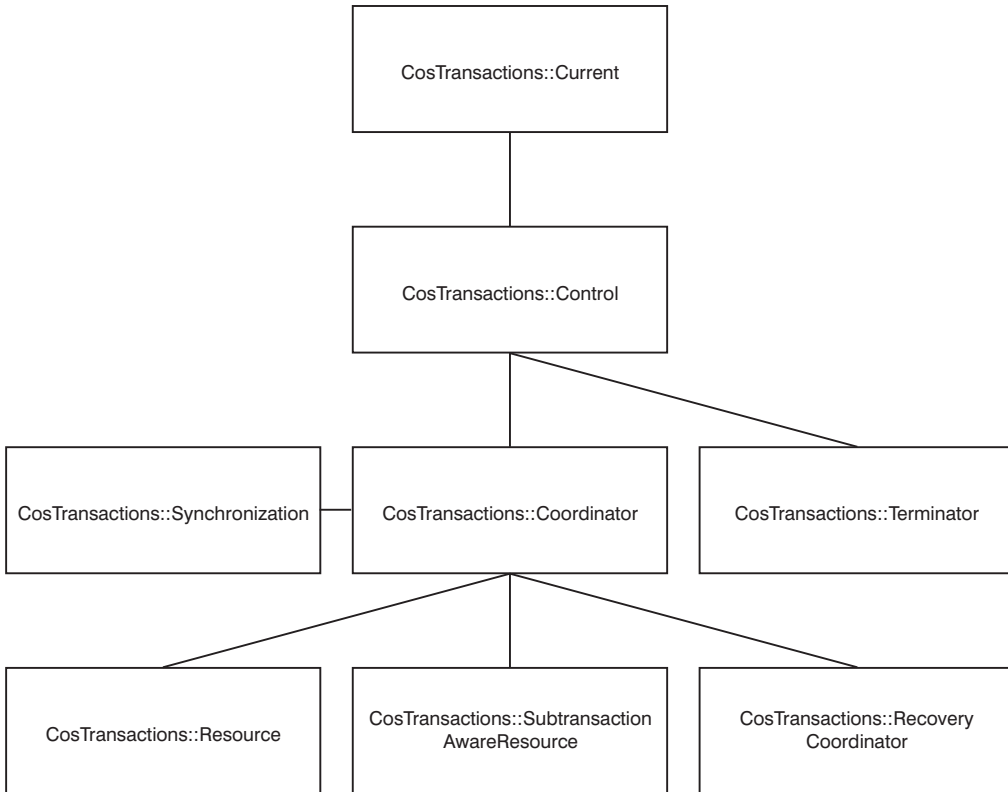
When the application has registered all its resource objects and has completed all the updates required for the transaction, it ends the transaction either by calling the `CosTransactions::Current` object, or by calling the `CosTransactions::Terminator` object extracted from the `CosTransactions::Control` object. It can request either that the transaction commits or that it rolls back (see “Lifetime of a transaction” on page 253). This results in the `CosTransactions::Coordinator` object making calls to the application’s resource objects. Using either the two-phase commit or one-phase commit process, the `CosTransactions::Coordinator` object ensures that the application’s resource objects are either all told to make their updates permanent, or all told to undo their updates even if some or all of the servers abnormally terminate during this operation.

If an application wishes to be called just before the resource objects are called during a commit operation and just after all of the resource objects are either committed or rolled back, it can create an object that implements the `CosTransactions::Synchronization` interface and register it with the transaction’s `CosTransactions::Coordinator` object. Such `CosTransactions::Synchronization` objects are normally only required when using a database such as DB2

When all of the application’s resource and `CosTransactions::Synchronization` objects have been called, the transaction is complete. The transaction’s `CosTransactions::Control`, `CosTransactions::Coordinator`, `CosTransactions::Terminator` objects and all the application’s resource and

synchronization objects are destroyed. The application can then use the `CosTransactions::Current` object to start another transaction.

The `CosTransactions::TransactionFactory` object provides an alternative operation for creating a transaction. It creates a transaction and returns a `CosTransactions::Control` object to an application without associating it with a thread. This might be required for some specialized applications that do not wish the transactions it is using to be associated with a thread.



Manage transactions in your application

Applications manage transactions primarily by using the `CosTransactions::Current` interface. The *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* describes all the Transaction Service interfaces and describes which ones are appropriate for use within the Component Broker Programming Model. Some of the later sections in this

chapter go beyond the Programming Model and are appropriate only for application adaptor writers. This material is provided for background information.

There is one `CosTransactions::Current` object in each operating system process, and it provides the operations to access the main facilities of the Transaction Service. More information on the use of these operations is given in the following steps.

Before carrying out this procedure, make sure you are familiar with:

- “An example of a transaction” on page 251
- “Lifetime of a transaction” on page 253
- “The Transaction Service objects and interfaces” on page 267
- “Design a Transaction Service application” on page 266

To manage transactions in your application, follow these steps:

1. Access the `CosTransactions::Current` interface. (See “Accessing the `CosTransactions::Current` Interface”.)
2. Set a time limit for all new transactions. (See “Setting a time limit for all new transactions” on page 271.)
3. Start a transaction using the `CosTransactions::Current` interface. (See “Starting a transaction using the `CosTransaction::Current` interface” on page 272.)
4. Pass the transaction to a remote object. (See “Implicitly propagate a transaction context to a remote object” on page 277.)
5. Update data.
6. End a transaction using the `CosTransactions::Current` interface. (See “Ending a transaction using the `CosTransactions::Current` interface” on page 278 .)

Accessing the `CosTransactions::Current` Interface

The `CosTransactions::Current` object is available in every process where the Transaction Service is located. It provides access to most of the operations you require to control the transactions used by your application.

The following information describes how to extract a reference to the `CosTransactions::Current` object.

Before carrying out this procedure, make sure you are familiar with “The Transaction Service objects and interfaces” on page 267.

To access the `CosTransactions::Current` object, follow these steps:

1. Get access to the ORB.
2. Get a reference to the CORBA::Object registered with the name "TransactionCurrent".
3. Narrow this CORBA::Object to a CosTransactions::Current object.

Here is an example:

```
#include <CBSeriesGlobal.hh>
#include <CosTransactions.hh>
// Function to return the CosTransactions::Current object for this process
CosTransactions::Current *get_CosTransactions_Current()
{
    CORBA::Object_var theCurrent;
    // Get access to the orb and retrieve the transactions current object
    theCurrent = CBSeriesGlobal::orb()->resolve_initial_references(
        "TransactionCurrent");
    // Narrow the to a CosTransactionsCurrent object
    return CosTransaction::Current::_narrow(theCurrent);
}
```

Setting a time limit for all new transactions

The CosTransactions::Current interface has a set_timeout() operation that enables your application to set a time limit for all transactions that are subsequently started.

The default time limit value is set to 300 seconds but is can be configured using the Systems Management interfaces. If the time limit has a value of "0", transactions subsequently started using the CosTransactions::Current interface does not have a time limit set.

Before carrying out this procedure, make sure you are familiar with:

- "Transaction time limits (timeouts)" on page 261
- "Accessing the CosTransactions::Current Interface" on page 270

To set a time limit for all new transactions, follow these steps:

1. Obtain a reference to the CosTransactions::Current object in this process.
2. Invoke the set_timeout() operation on the CosTransactions::Current object, passing the time limit required as a parameter in seconds.

```
#include <CosTransactions.hh> // CosTransactions module
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the set_timeout operation on the CosTransactions::Current object.
```

```

current->set_timeout(60 /* seconds */);
...
// Start a transaction
...

```

If a transaction exceeds the time limit set, the Transaction Service rolls back the transaction and removes the `CosTransactions::Coordinator` object. When the application subsequently calls a remote object or the `CosTransactions::Control::get_coordinator()` method, the `CORBA::TRANSACTION_ROLLEDBACK` exception is raised. If this occurs, the application should call `CosTransactions::Current::rollback()` to remove the `CosTransactions::Control` object and `CosTransactions::Terminator` object.

Starting a transaction using the `CosTransaction::Current` interface

To start a transaction, an application uses the `begin` operation of the `CosTransaction::Current` object. Invoking the `begin()` operation causes a transaction to be created and to be associated with the current thread of execution. The thread is then said to be running within the scope of the newly created transaction. (See “Transaction scope and context” on page 253 for more information.)

If a transaction was not associated with the current thread when the `begin()` operation was called, a top-level transaction is created. (See “Top-level and flat transactions” on page 252 for more information.)

Before carrying out this procedure, make sure you are familiar with:

- “Lifetime of a transaction” on page 253
- “Top-level and flat transactions” on page 252
- “Accessing the `CosTransactions::Current` Interface” on page 270

To start a transaction using the `CosTransactions::Current` Interface, follow these steps:

1. Access the `CosTransactions::Current` object.
2. Invoke the `begin()` operation on the `CosTransaction::Current` object.

Here is an example:

```

#include <CosTransactions.hh> // CosTransactions module...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);

```

```
...
// Invoke the begin operation on the CosTransactions::Current object.
current->begin();
...
```

If the Transaction Service is unavailable, it will throw one of the following standard exceptions:

CORBA::INITIALIZE

This means that a problem in recovery was detected while the server was starting. For example, the server was unable to open its log file. Error messages describing the problem were logged in the activity log when the problem was detected.

CORBA::PERSIST_STORE

This means that while running transactions the server found it was unable to use its log file. For example, because of insufficient disk space. Once the problem with the log file is fixed, the server will need to be restarted before new transactions can be created.

CORBA::INVALID_TRANSACTION

This means the Transaction Service is unable to start a transaction.

All of these exceptions are written to the activity log with a minor error code. This minor error code gives more detail on the cause of the problem.

Suspending a transaction from the current thread

When a thread is associated with a transaction, an application can use the `suspend()` operation of the `CosTransactions::Current` object to remove the association between the thread and the transaction. Following transaction suspension, the transaction still exists but is no longer automatically propagated to other objects in the application - that is, the current thread is no longer running within the scope of the transaction. (See “Transaction scope and context” on page 253 for more information.)

The `CosTransactions::Current::suspend` operation returns a `CosTransactions::Control` object reference that represents the transaction that has been suspended. This reference can be passed to another thread and used to resume the transaction later. See “Passing a transaction context to another thread” on page 275 and “Resuming a transaction on the current thread” on page 274 for more information.

Before carrying out this procedure, make sure you are familiar with:

- “Transaction scope and context” on page 253
- “The Transaction Service objects and interfaces” on page 267
- “Accessing the `CosTransactions::Current` Interface” on page 270

To suspend a transaction from the current thread, follow these steps:

1. Obtain a reference to the `CosTransactions::Current` object in this process.
2. Invoke the `suspend()` operation on the `CosTransactions::Current` object.

Here is an example:

```
#include <CosTransactions.hh> // CosTransactions module...
CosTransactions::Control_ptr control = NULL;
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references(
        "TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
...
// Invoke the begin operation on the CosTransactions::Current object.
current->begin();
...
// Suspend the association between the transaction and the thread.
control = current->suspend();
if (!control)
{
    // There was no transaction associated with this thread prior to
    // the suspend. Perform appropriate action.
    cout << "Error": No transaction prior to suspend" << endl;
}
}
```

Resuming a transaction on the current thread

If an application has a pointer to a `CosTransactions::Control` object that represents an active transaction, it can use the `resume()` operation of the `CosTransactions::Current` object to associate the transaction with the current thread of execution (in place of any previous transaction). Following the `resume()`, the current thread runs within the scope of the transaction.

An application can also pass a `NULL` pointer on the `resume()` operation to clear the thread of any association with transactions.

Before carrying out this procedure, make sure you are familiar with:

- “Transaction scope and context” on page 253
- “The Transaction Service objects and interfaces” on page 267
- “Accessing the `CosTransactions::Current` Interface” on page 270

To resume a transaction on the current thread, follow these steps:

1. Obtain a reference to the `CosTransactions::Current` object in this process.

2. Invoke the `resume()` operation on the `TransactionCurrent` object, passing the reference of the `CosTransactions::Control` object representing the transaction to be suspended.

Here is an example:

```
#include <CosTransactions.hh> // CosTransactions module...
CosTransactions::Control_ptr control = control_parameter;
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
...
// Resume the association between the transaction and the thread.
try
{
    current->resume(control);
}
catch(CosTransactions::InvalidControl)
{
    cout << "Error: control object passed to resume was invalid" << endl;
}
}
```

Passing a transaction context to another thread

The Transaction Service allows an application to pass a transaction between its threads. The application does this by extracting the `CosTransactions::Control` object using the `CosTransactions::Current` object and passing it to the other thread. The other thread can then use the `resume()` operation of `TransactionCurrent` to associate the transaction with the itself.

When extracting the `CosTransactions::Control` object from the `CosTransactions::Current` thread, the thread that is passing the transaction has two options:

- To use the `CosTransactions::Current` `get_control()` operation to retrieve a reference to the `CosTransactions::Control` object while leaving the transaction still associated with the thread, or
- To use the `CosTransactions::Current` `suspend()` operation to retrieve a reference to the `CosTransactions::Control` object. This removes the association between the current thread and the transaction.

Applications would use `get_control()` operation when both threads are to continue using the transaction, and `suspend` if the aim was to completely transfer execution for the transaction to another thread.

Before carrying out this procedure, make sure you are familiar with:

- “Transaction scope and context” on page 253

- “The Transaction Service objects and interfaces” on page 267
- “Accessing the CosTransactions::Current Interface” on page 270
- “Suspending a transaction from the current thread” on page 273
- “Resuming a transaction on the current thread” on page 274

To pass a transaction context to another thread, follow these steps:

1. Obtain a reference to the CosTransactions::Current object in this process.
2. Invoke the suspend or get_control() operation on the CosTransactions::Current object.
3. Put the pointer in a location where it can be accessed by another thread.
4. Start a new thread.
5. Within the new thread, retrieve the pointer to the Control object and use the CosTransactions::Current resume() operation to associate the thread with the transaction.

Here is an example:

```
#include <CosTransactions.hh> // CosTransactions module
...
CosTransactions::Control_ptr control = NULL;
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
...
// Suspend the association between the transaction and the thread.
global_control = current->suspend();
// Start a new thread
...

void new_thread(void *parameter)
{
    // Associate the transaction with my thread.
    try
    {
        current->resume(_global_control);
    }
    catch(CosTransactions::InvalidControl)
    {
        cout << "Error: control object passed to resume was invalid" << endl;
    }
    ...
}
```


Implicitly propagate a transaction context to a remote object

Having started the transaction, your application is ready to perform work as part of that transaction. This typically involves invoking methods on other objects that can be either local and remote.

When an application has a transaction associated with a thread, the Transaction Service automatically propagates the transaction to all objects invoked in the same thread and all remote objects invoked from this thread that inherit from the `CosTransactions::TransactionalObject` interface. These objects are then able use the `CosTransactions::Current` operations to control the transaction.

The Transaction Service does however place one restriction on the use of the transaction in servers other than the one that started the transaction: they cannot end the transaction. Only threads in the operating system process where the transaction was started can call any of these operations.

- `CosTransactions::Current::commit`
- `CosTransactions::Current::rollback`
- `CosTransactions::Terminator::commit`
- `CosTransactions::Terminator::rollback`

Before carrying out this procedure, make sure you are familiar with:

- “Lifetime of a transaction” on page 253
- “Transaction scope and context” on page 253

To pass a transaction context to a remote object, follow these steps:

1. Derive your remote object class from the `CosTransactions::TransactionalObject` interface. This will happen automatically if the remote object resides on a Component Broker server, as all managed objects inherit from `CosTransactions::TransactionalObject`.
2. If you invoke a method on the remote object from within the scope of a transaction, it propagates the transactional context.

Forcing a transaction to rollback

The `TransactionCurrent` interface has a `rollback_only()` operation, which enables your application to ensure that an active transaction rolls back even if the `TransactionCurrent commit()` operation is called. Applications use this operation if they detect an error that means all the updates could not be completed successfully.

Once the `rollback_only()` operation has been called, the application is no longer able to invoke methods on some remote transactional objects or register

resource objects with the transaction's `CosTransactions::Coordinator` object. See "Implicitly propagate a transaction context to a remote object" on page 277 for more information.

Before carrying out this procedure, make sure you are familiar with:

- "Lifetime of a transaction" on page 253
- "Accessing the `CosTransactions::Current` Interface" on page 270

To force a transaction to rollback, follow these steps:

1. Obtain a reference to the `TransactionCurrent` object in this process.
2. Invoke the `rollback_only()` operation on the `TransactionCurrent` object.

Here is an example:

```
#include <CosTransactions.hh> // CosTransactions module
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the rollback only operation on the TransactionCurrent object.
current->rollback_only();
...
```

Ending a transaction using the `CosTransactions::Current` interface

If the work has been done correctly inside the transaction, the application ends the transaction by committing the changes to the resources. Typically, this is done by invoking the `TransactionCurrent` `commit()` operation on a thread that is running within the scope of the transaction. A boolean parameter passed to the `commit()` operation indicates whether the application requires heuristic exceptions to be reported (a value of `TRUE` indicates that they should).

If the application cannot commit all the changes made to resources (for example, if an attempt to update a resource in the transaction resulted in an error code being returned), the application might choose to rollback the changes it made as part of the transaction. Typically, this is done by invoking the `TransactionCurrent` `rollback()` operation from a thread that is running within the scope of the transaction that is to be rolled back.

Before carrying out this procedure, make sure you are familiar with:

- "Lifetime of a transaction" on page 253
- "Accessing the `CosTransactions::Current` Interface" on page 270

To end a transaction using the TransactionCurrent interface, follow these steps:

1. Access the CosTransactions::Current instance by narrowing the CORBA::Current object.
2. Invoke the commit() or rollback() operation on the TransactionCurrent instance.

Here is an example:

```
#include <CosTransactions.hh> // CosTransactions module
...
::CORBA::Boolean rollback_required = FALSE;
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references(
        "TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);

// Invoke the begin operation on the TransactionCurrent object.
current->begin();

// Perform work for the transaction and set rollback_required to TRUE if
// an error is detected.
...

// Invoke commit or rollback depending on whether rollback_required is
// set. This must be called within a try...catch structure as the
// transaction service may raise an exception if an error occurs.

try
{
    if (rollback_required == TRUE)
    {
        current->rollback();
    }
    else // commit required
    {
        current->commit(/* report_heuristics = */ TRUE);
    }
}
catch (CORBA::TRANSACTION_ROLLEDBACK &exc)
{
    // The application called commit, but the transaction service rolled
    // the transaction back because an error was detected.
    ...
}
catch (CosTransactions::HeuristicMixed &exc)
{
    // The transaction service has reported that some or all of the resource
    // objects have made a heuristic decision. This has resulted in
    // heuristic damage.
    ...
}
```

```

}
catch (CosTransactions::HeuristicHazard &exc)
{
    // The transaction service has reported that not all of the resource
    // objects could participate properly in determining the outcome of the
    // transaction. There is a possibility of heuristic damage.
    ...
}
catch (CORBA::UserException &exc)
{
    // Another type of user exception has occurred.
    ...
}
catch (CORBA::SystemException &exc)
{
    // The application called commit, but the transaction service rolled
    // the transaction back because an error was detected.
    ...
}
catch (...)
{
    // A general exception has occurred.
    ...
}
...

```

If a transaction rolls back unexpectedly, then the problem may be:

- There are requests to remote objects still outstanding.
- The time limit for the transaction has been exceeded. See “Transaction time limits (timeouts)” on page 261.
- An object in your application has set the transaction to rollback. See “Forcing a transaction to rollback” on page 277.
- A server that houses some of your application’s objects has failed.
- A standard exception was raised and not caught during a remote method request. See “Handle exceptions”.

Handle exceptions

As with other CORBA Object Services, the Transaction Service may return a CORBA exception from a call. There are two types of CORBA exception

- User exceptions
- System exceptions

User exceptions can only be returned from an interface if it is specified on the definition of the interface. For an example, see the `CosTransactions::Current::begin` interface in the Programming Reference.

The Transaction Service user exceptions are:

- `CosTransactions::SubtransactionsUnavailable`

- CosTransactions::Inactive
- CosTransactions::NotPrepared
- CosTransactions::NoTransaction
- CosTransactions::InvalidControl
- CosTransactions::Unavailable
- CosTransactions::SynchronizationsUnavailable
- CosTransactions::HeuristicCommit
- CosTransactions::HeuristicRollback
- CosTransactions::HeuristicHazard
- CosTransactions::HeuristicMixed

System exceptions may be returned from any call. They are defined as part of the CORBA module. The system exceptions that are specific to the use of the Transaction Service are:

CORBA::TRANSACTION_ROLLEDBACK

If this exception is returned by either of the following methods:

- CosTransactions::Current::commit
- CosTransactions::Terminator::commit

The transaction has rolled back. If it is returned by any other method then the transaction has either been marked rollback only (“Forcing a transaction to rollback” on page 277) or is about to rollback. The transaction initiator (normally the client process) should then call either:

- CosTransactions::Current::rollback
- CosTransactions::Terminator::rollback

To complete the rollback process. (See “Ending a transaction using the CosTransactions::Current interface” on page 278.)

CORBA::INVALID_TRANSACTION

If this exception occurs, the state of the transaction does not allow the requested operation. If a transaction has been successfully started, the transaction initiator (normally the client process) should then call either:

- CosTransactions::Current::rollback
- CosTransactions::Terminator::rollback

To end the transaction. (See “Ending a transaction using the CosTransactions::Current interface” on page 278.)

CORBA::TRANSACTION_REQUIRED

If this exception occurs, the requested operation requires a transaction and no transaction exists. The application should start a transaction and retry the operation. (See “Starting a transaction using the CosTransaction::Current interface” on page 272 for more information.)

Whenever a system exception is raised, it is associated with a minor error code. These give more detail as to the cause of the error and are documented in Exceptions Defined by Minor Error Code, in the reference section.

Prevent memory leaks in the Transaction Service

Memory leaks occur when objects are created and not deleted, or storage is acquired and not freed. If memory leaks occur continuously in a process, the size of the process increases, making it run slower and eventually fail. The list below describes how to avoid memory leaks of Transaction Service objects.

Make sure that a process that starts a transaction always ends the transaction. (See “Starting a transaction using the `CosTransaction::Current` interface” on page 272 and “Ending a transaction using the `CosTransactions::Current` interface” on page 278 for more information.)

If your application uses an interface that returns a Transaction Service object such as:

- `CosTransactions::Current::get_control`
- `CosTransactions::Control::get_terminator`
- `CosTransactions::Control::get_coordinator`

then ensure it also calls `CORBA::release()` to release the object reference once it is no longer required.

Always specify a time limit (timeout) for transactions started from a client so that the transaction is rolled back if the client terminates before ending the transaction. (See “Transaction time limits (timeouts)” on page 261 and “Non-recoverable client” on page 263 for more information.)

Controlling the Transaction Service in a running system

This section includes the following information:

- The transaction service log
- Types of server start-up
- “Configuring a server to use the Transaction Service” on page 285
- Problem determination

The transaction service log

The Transaction Service creates a log for every server. This records information about transactions running in the server and is used for recovery.

Important:

The log here is not the same as a message log (such as the Component Broker activity log), audit trail or journal service. It is only used for transaction state information and is not in a readable format.

The name of a server's log is assigned by the Transaction Service the first time the server is started, and appears in one of the messages written to the Component Broker activity log as the server starts up. The name is of the form `somtr nnn` ; for example, `somtr0000`. This name is used as a prefix to all files that belong to the log. The log itself is split into two parts. Files that begin `somtr nnn .p` contain information about the databases and any other server that the local server is coordinating transactions on. Files that begin `somtr nnn .t` describe the transactions that are running in the server.

Each part of the log consists of three types of files, qualified by the file extension:

- .ctl** A control file (for example, `somtr000.t.ctl`).
- .csh** A cushion file (for example, `somtr000.t.csh`).
- .nnn** Any number of extent files (for example, `somtr000.t.001`, `somtr000.t.002`, and so on). The number of extent files used is dependent on the number and age of the running transactions in the server.

The amount of disk space required for the log depends on the number of transactions running in the server, the type of work involved in these transactions, and their duration. It is recommended that you initially allow about 1MB of disk space per server. This amount can be revised once the load on the server can be assessed. If the transaction log runs out of disk space, your server will stop processing transactions until you make more disk space available, and have then restarted the server.

The location of the log is defined in the *log directory* attribute found in the Transaction Service part of the server's server image. The directory specified must exist and be on a local file system. It is also advisable to use a file system where any other data on it is reasonably static. This will ensure there is always sufficient disk space for the log.

As transactions running within the server complete, their entry is removed from the log. There is no need to perform housekeeping functions on the log files. If a server fails because of insufficient space in the file system, make more space without deleting any of the log files. The only time it is safe to delete a log file is if the file is for a server that has been deleted.

A server stores the name of its log in the Component Broker System Management configuration data files. If you recreate your configuration data files, the server will no longer have the name of the log file and will be allocated a new log name. Therefore, before recreating your configuration data files, start up each server and ensure the Transaction Service starts successfully. Then, without allowing any new transactions to be started, shut the servers down. There will then be no partially-complete transactions in your servers. Now you can safely delete the log files and recreate the System Management configuration data files.

Types of server start-up

The first time a server is started, the Transaction Service creates a Transaction Service log (“The transaction service log” on page 282) for recording information about running transactions. The name of this log appears in a message written to the Component Broker activity log:

```
Opening new transaction service log <logfile> in server <serverName>  
The transaction service started successfully in server <serverName>
```

It is then possible to view a server’s log name using the System Manager User Interface. The name of the log file is also displayed in a message written to the Component Broker activity log at each subsequent start of the server.

```
Opening transaction service log <logfile> in server <serverName>  
The transaction service started successfully in server <serverName>
```

If the Transaction Service discovers that there were transactions running when the server was last terminated, messages appear in the Component Broker activity log indicating that the Transaction Service is recovering incomplete transactions:

```
Opening transaction service log <logfile> in server <serverName>  
Recovering incomplete transactions from the transaction service log for  
server <serverName>  
< a delay may occur here if other servers are not available >  
All recovered transactions are now complete in server <serverName>  
The transaction service started successfully in server <serverName>
```

The process of completing recovered transactions is called resynchronization. The server can only work with new transactions when resynchronization is complete.

If the Transaction Service cannot create or open its log, messages such as those shown below are logged in the Windows NT application:

```
The transaction service in server <serverName> could not open its log.  
Error was <errorCode>  
The transaction service is unavailable in server <serverName>
```


Any attempt to start a transaction in the server once this message has been logged results in a CORBA::INITIALIZE exception.

A description of these messages, and the correct action to take when they occur, can be found in the “Understand Transaction Service Messages” topic in the “Problem Determination” section of the *WebSphere Application Server Enterprise Edition Component Broker Problem Determination Guide*.

Configuring a server to use the Transaction Service

The default configuration for each application server is set so that the Transaction Service is enabled.

If you wish to change the default attribute settings, you should use the System Manager, as described in the following steps:

1. Edit the Server Model for the server.
2. In the Object Editor Notebook, select the **Transaction Service** tab. The following attributes relate to the Transaction Service:
 - a. *Retry restricted*
 - b. *Commit retry limit*
 - c. *Heuristic direction*
 - d. *Log directory*
 - e. *Log file size*
 - f. *Transaction timeout*
3. To save the changes and exit the Object Editor Notebook, click **OK**.
4. To apply the changes to the server, activate the configuration containing the Server Model.

Note: The Transaction Service will not be available in an application server if the *log directory* attribute specifies a directory that does not exist. This is because the Transaction Service creates log files in this directory that are required to preserve transaction integrity in the event of server failure. You must therefore check that the directory specified is valid. If the directory does not exist the Transaction Service will fail to initialize and it will not be possible to begin a transaction.

Problem determination

While it is running, the Transaction Service writes out a number of messages. For an explanation of the Transaction Service messages, please see the “Understand Transaction Service Messages” topic in the “Problem Determination” section of the *WebSphere Application Server Enterprise Edition Component Broker Problem Determination Guide*.

While the Transaction Service is running, Component Broker may also issue messages relating to XA Resource Managers. For an explanation of the XA

messages, please see the “Understand XA Messages” topic in the “Problem Determination” section of the *WebSphere Application Server Enterprise Edition Component Broker Problem Determination Guide*.

Summary of the Transaction Service

- The Transaction Service enables programmers to implement transactions through standard CORBA interfaces.
- A transaction simplifies applications that need ACID (atomicity, consistency, isolation and durability) properties.
- The Transaction Service provides support for top-level transactions only, and only top-level transactions can be used when accessing databases.
- Applications that use the Transaction Service normally have a client/server design and use managed objects to implement recoverable objects.

Chapter 10. Session Service



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

The Session Service is provided as part of the procedural application adaptor (PAA) software. The following sections describe what the Session Service is, and how to use it.

What is the Session Service?

PAA introduces the notion of a session to help manage resources within the context of a unit-of-activity scope. A session context is similar to a transaction context. For example, a session defines a scope and associated consistency and visibility semantics under which a set of activities is performed.

A session is started and ended by a client program (or something on its behalf). Sessionable objects that are invoked within the session context can register a sessionable-resource with a session coordinator. Sessionable-resources are then notified to participate in the completion of that session. In addition, the client can initiate intermediate checkpoints and resets of the session. These operations are discussed later in this chapter.

The scope of sessions

The Session Service provides primitives for applications in a distributed object environment to control the scope of a session and the session profile and arbitrary session properties that are relevant within the scope of that session.

The scope of the session is defined to exist between the point when the session is started and the point when the session is ended (with the `beginSession()` and `endSession()` methods). Within the bounds of the session, the application can suspend a session, copy or move a session to a different thread of execution, and resume the session. In general, an application can only end a session that is active on their thread of execution. Also, generally, a session can only be ended by the client (or server) thread and process that started the session. However, a session can time out any thread with access to the session context and force a reset (and termination of the session), and multiple threads in the top-level session can collaborate on the outcome of the

session, superseding either of the previous two rules. For more information about timeouts, see “The timeout value associated with a session” on page 289.

A session scope can be used by the server run time for a variety of purposes:

- Loose atomicity
- Loose consistency
- Isolation
- Reuse of resources
- Resource cleanup

In terms of atomicity, the Session Service does not guarantee atomic updates. However, the session can be used to synchronize the checkpointing of state among multiple, non-collaborating resources. Since all registered resources are synchronized on the session checkpoint event, the Session Service can be used to achieve loose-consistency among these otherwise disparate resources. Again, the consistency is not guaranteed, and no consistency is attempted between transactional resources and non-transactional resources operated on in the same session, except at the termination of the session. The Session Service will only attempt to synchronize transactional and non-transactional resources at the end of the session, essentially by preventing the session from being terminated until all outstanding transactions have completed.

In terms of isolation, the session provides the ability, outside a transaction context, to isolate particular instances of objects from other sessions; each session can have its own view of object data.

In terms of reuse, resources may be allocated during a session and carried forward from one method request to another in the same session context. An example of this is to reuse a CICS terminal from one method request to another. This would allow a pseudo-conversational set of CICS transactions to occur with the passing of COMMAREA data from one CICS transaction to the other. With resource cleanup, any resources allocated on behalf of the session can be cleaned up at the completion of the session.

Although this appears very similar to what a transactional context would define, it is different in one significant way: there is no commit/rollback support at all.

Sessions cannot be nested. There is at most one session context associated with a current thread at any time.

The relationship between transactions and sessions

The relationships between sessions and transactions are outlined briefly below. Specifically, a transaction is either completely inside one session, or is outside all sessions.

- Transactions can be started outside of the scope of a session. However, if that transaction is suspended it cannot be resumed in the context of an active session.
- A session cannot be started in the middle of an active transaction.
- Multiple transactions can be implemented within a single session context. However, if a transaction is started within a session context and then suspended, it cannot be resumed within a different session.
- A session cannot be terminated while any transactions are active within it. The transaction must be terminated before the session can be terminated. However, ending a session with the `Reset-Force` end-mode will automatically roll back any pending transactions before terminating the session.
- If the session is suspended on a thread of execution, then the active transaction, if any, is suspended on that thread of execution as well. When the session is resumed, the suspended transaction is resumed on that thread of execution as well.
- Sessions cannot be nested; a transaction can be involved with one session at most.

Note: Client application writers must not use the `TransactionFactory` directly to create a transaction, or the `transaction Terminator` directly to end the transaction. The checking of the relationships is done by the `transactions Current` and `sessions Current` objects, so these must be used.

The timeout value associated with a session

A session carries with it a timeout value which is established when the session is initially begun. This value defines the maximum duration of the session in seconds. If the timeout value is exceeded, the session is automatically ended (timed out). Setting the timeout value to zero (0) means that the session will not time out.

The timeout value is part of the session context and is propagated on remote method calls to other servers involved in the session. Any server involved in the session is capable of determining that the timeout value has elapsed and initiating the timeout of the session.

The action taken when a session timeout condition occurs is to cause the session to be ended with the `'EndModeResetForce'` end mode.

Resource priorities

When registering a Resource, a priority value is specified. The following table lists the priority values:

Range	Meaning	Recommended Use
0x0000 - 0x0FFF	Top priority	BO-supplied resources
0x1000 - 0x3FFF	High priority	
0x4000		MOs
0x4001 - 0xBFFF	Medium priority	
0xC000		Connections
0xC001		PAO cache
0xC002 - 0xFFFF	Low priority	

Using the Session Service

There are two perspectives on why a Session Service is useful:

- From the perspective of a client programmer, a session is useful for demarcating the beginning and end of related activity.
- From the perspective of a managed object, a session is useful for controlling the lifecycle of their objects when their objects cannot adhere to the strict ACID properties required in a transaction context.

Client use of the Session Service

A client programmer can use the Session Service to begin and end a session context. The session context:

- Has a beginning and an ending.
- Has state.
- Is associated with the thread of execution on which it was begun.
- Propagates implicitly with any method requests initiated within that session context.

The client program can associate a session profile with the session, and in doing so, indicates the way that it intends for resources to behave or be handled within that session context. This can include whether the application requires pessimistic or optimistic locking of data, whether it can tolerate deferred consistency, etc. In addition, the application can establish arbitrary name-value pairs (session properties), that act as global, distributed, environment variables. Session properties also propagate implicitly with any method requests initiated within that session context, and dissipate them at the end of the session context.

More importantly, the application can indicate when it is through with any resources that it was using within the session. The system can take this cue to clean up any transient or other resources that were allocated for the session that were not otherwise cleaned up as part of some other, more strict context-binding mechanism, such as a transaction context.

Finally, the application can use the session to control the checkpointing of persistent state for any resources it used within the session, in a loosely-consistent fashion. When the application issues a checkpoint request, a best-effort attempt will be made to transition object state into the persistent storage systems underlying those objects that are not controlled with a transaction context.

Client applications should begin a session when they want to perform a set of activities under a common session profile, and as a single logical activity. They should then perform their work, and then end the session when they have completed that activity and want to free up any system resources that were allocated for use within that activity. In between, the client application can issue a checkpoint request to push out any changes that have been made to those resources, to the back-end data systems that back up those resources. Having checkpointed the object-system, the application can also reset the state of the back-end data systems back into their object-system resources. In addition to controlling the session, the application can also control any number of transaction contexts, either completely within, or completely outside of, the session context.

When a client originates a session, it needs to choose a Component Broker server on which to create the Session Service objects. The choice of this server can have a significant impact on performance. For example, if the client chooses a server that would not normally be involved with the session, performance will be adversely affected. Unlike the Transaction Service client, the Session Service client creates a session as soon as the `beginSession()` method is called. At that point in time the client has no knowledge of which servers any of the business objects will be created on. In order to determine which server a session is created on, the Session Service client uses the client style image stored in the System Management configuration data files. The *factoryFinder* attribute of the client style names the default factory-finder that will be used by the client to find a session factory, and then determine the server on which the session will be created. If no session factory can be found using this method, an arbitrary session factory will be chosen using the host-scope factory finder.

The Session Service client library is part of the Object Services library, called “*somosai*”, for the IBM VisualAge C++ compiler, and “*somosam*” for the Microsoft Visual C++ compiler. Component Broker also provides a Java Client SDK zip file, *somajor.zip*, with Java client implementations of the Session

Service interface. Note that, in the case of a Java client that runs on a machine capable of also supporting a Component Broker server, the SOMOJOR.zip file must be at the start of the CLASSPATH otherwise the business object (server) Java bindings will be loaded when a CORBA object service interface is called and a CORBA::NO_IMPLEMENT exception will be thrown.

Managed object developer use of the Session Service

From the perspective of a managed object developer, transactional resources are generally orthogonal to sessionable resources. That is, a resource is either a transactional resource, and subject to the strict ACID properties that characterize transactional semantics, or the resource is a sessionable resource. Sessions, however, do not guarantee atomicity, consistency, or durability of updates, although they can be used to ensure isolation or visibility. Sessions operate with best-effort semantics.

Consequently, each managed object implementation must decide whether it can support the stricter ACID properties associated with a transactional object. If it can, then it should be implemented, and register a transactional-resource object with the Transaction Service. However, for objects that are implemented over many legacy procedural systems, such as CICS or IMS transactions, the strict ACID properties of a transactional resource may be difficult, if not impossible, to achieve. These objects should be implemented as sessionable objects, and register a sessionable-resource object with the Session Service.

Managed object developers can choose to implement their objects with the more stringent ACID properties of a transactional resource, or with the loose-consistency properties of a sessionable object. Typically this will be driven by the capabilities of the data system over which the managed object is implemented, and as instituted by the qualities of service support provided by the corresponding application adaptor. Depending on which is selected, the managed object will then be governed by either the session context and lifecycle, or the transaction context and lifecycle.

Visibility rules

Sessions can be run concurrently from different applications. Some complexity is involved when some of these concurrently running applications are accessing the same data from the same data store. Visibility rules define how the data interactions between these concurrent sessions are defined.

The Session Service is provided as part of a solution to provide Component Broker applications access CICS or IMS based applications on the third tier of a three-tiered architecture. Component Broker uses different mechanisms to attach to CICS and IMS applications, and these mechanisms have a slightly different behavior. Unfortunately, this behavior can reflect back into the client

application and provide slightly different results for concurrently running applications. The following sections describe the different behavior relative to the following categories:

- Attribute getters and setters - those methods that read or update the attributes of a business object.
- Create, retrieve, update and delete methods.
- Pushdown methods - some methods cause interactions with the data store. For example, a method may need the value of an attribute from the data store. This will cause a retrieve operation to be performed. These methods that can indirectly cause communication with the data store are referred to as pushdown methods. It is the responsibility of the application developer to document which attributes may be read or modified for each pushdown method.
- Checkpoint and end session methods.

It should be noted that although these methods may make changes visible to other sessions or other applications, these changes are not forced on these sessions or applications. These sessions or applications must take some action to refresh their state or read the changed data before the changes are evident in these other sessions or applications. Visibility should be viewed as making the changes *available* to other sessions or applications.

Using HOD to access to CICS and IMS applications

Host on Demand (HOD) is the mechanism used to provide access to CICS and IMS 3270 applications. The categories behave in the following manner:

Attribute getters and setters

When these methods are implemented, they only affect the session in which they are used. They are *not visible* to other sessions.

Create, retrieve, update and delete methods

When these methods are implemented, the changes are *visible* to other sessions.

Pushdown methods

When these methods are implemented the changes are *visible* to other sessions.

Checkpoint and end session methods

When these methods are implemented the changes are *visible* to other sessions.

When using this mechanism, many of the methods that are implemented result in changes that are *immediately visible* to other sessions or other applications.

Using ECI to access to CICS applications

ECI (External Call Interface) is one of the mechanisms used to provide access to CICS COMMAREA applications. The categories behave in the following manner.

Attribute getters and setters

When these methods are implemented they only affect the session in which they are used. They are *not visible* to other sessions.

Create, retrieve, update and delete methods

When these methods are implemented the changes are *not visible* to other sessions.

Pushdown methods

When these methods are implemented the changes are *not visible* to other sessions.

Checkpoint and end session methods

When these methods are implemented the changes are *visible* to other sessions.

When using this mechanism, many of the methods that are implemented will result in changes that are *not immediately visible* to other sessions or other applications.

Transactional access to CICS and IMS

Transactional access to CICS and IMS applications is also provided in the PAA APPC support. Transactional visibility rules are discussed in the Chapter 9. Transaction Service chapter.

Toward a more common client programming model

In some instances, it is important for the application developer to have a more common programming model that is more independent of the attachment mechanism being used. In these cases, the following technique could be implemented. It can be deduced from the previous information that checkpoint or endSession operations could be used to bring these two models closer together.

A simple approach would be to code more checkpoint operations throughout the client application. This would bring the visibility of these two mechanisms very close. However, this technique should only be used if a more common programming model is desired since it removes the capability of reversing a set of operations through the use of an endSession(EndModeReset) operation.

Session Service tasks

The following sections outline the fundamental tasks that you can perform using the Session Service.

Setting a time limit for all new sessions

The `ISessions::Current` interface has a `setSessionTimeout()` operation which enables your application to set a time limit for all sessions that are subsequently started.

The default time limit is set to zero. That is, sessions may run indefinitely.

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287
- “The timeout value associated with a session” on page 289

To set a time limit for all new sessions, follow these steps:

1. Access the `ISessions::Current` object by narrowing on the `CORBA::Current` object returned from the ORB.
2. Invoke the `setSessionTimeout()` operation on the `ISessions::Current` object, passing the new timeout value.

Here is an example:

```
CORBA::Object_var object;
ISessions::Current_var sessionCurrent;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "SessionCurrent");
sessionCurrent = ISessions::Current::_narrow(object);
// Set the session timeout.
sessionCurrent->setSessionTimeout(100);
```

Beginning and ending a session

Support for the session profile is provided for future expansion of the Session Service and is not enabled in the current release. Any value specified for the session profile is ignored and `beginSession` always returns `FALSE`.

Beginning a session

You begin a session by invoking the `beginSession()` operation on the `ISessions::Current` object. You should supply a text string representing the name of your application, or the session profile under which you want the session to operate. If the specified profile cannot be found, or if you specify an empty string, then a default session profile is used.

The session profile is intended to allow specification of certain expectations about how the session will behave. This information can be used in combination with the capabilities and policies of the running system to produce a set of execution decisions that optimize the total performance and throughput of the system.

Once the session has been started, you can perform any number of operations on business objects within the session. All operations invoked within the session are performed with the same session context.

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287
- “Client use of the Session Service” on page 290

To start a session, follow these steps:

1. Decide on the application name or session profile name under which you want the session to operate.
2. Get access to the ORB.
3. Get a reference to the CORBA::Object registered with the ORB with the name “SessionCurrent”.
4. Narrow the CORBA::Object to an ISessions::Current object.

Here is an example:

```
CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_narrow(current);
// Begin a session context.
ISessions::Current->beginSession("LifeInsuranceApplication");
```

If the Session Service is unavailable, it will throw the following standard exception:

CORBA::INITIALIZE

This means that the Session Service initialization failed. For example, because a factory could not be found to create the objects needed for the session context.

CORBA::UNKNOWN

This means that the Session Service encountered an unexpected error such that initialization could not continue.

These exceptions are written to the activity log with a minor error code. This minor error code gives more detail on the cause of the problem.

Ending a session

You complete the session by invoking the `endSession()` operation on the session `Current`. Normally, you specify the `EndModeCheckpoint` end-mode with this operation. This drives all the sessionable resources used within the session to save their state changes persistently, through embedded operations on the underlying data system.

To reset the session, end it without saving any of the changes that occurred since your last checkpoint (or since the beginning of the session if you did not perform any checkpoints). Specify the `EndModeReset` end-mode with the `endSession` request.

`EndModeCheckpoint` and `EndModeReset` have no bearing on any transactions issued within the session, other than to ensure that the session is terminated before the session can end. However, if you encounter severe errors in your processing, you can end the session with `EndModeResetForce`. This will force the session to be reset immediately, including rolling-back any outstanding transactions.

To end a session, follow these steps:

1. Access the `ISessions::Current` object as described previously. You may also use an `ISessions::Current` object obtained from a previous operation.
2. Decide on the way in which the session should end: checkpoint, reset or reset-force.
3. Decide whether the `endSession()` operation should return immediately in the case where other threads in the same process still have the session context active, or whether to wait until the session context has ended on all threads.
4. Invoke the `endSession()` operation on the `ISessions::Current` object, passing the end mode value and the wait flag value.

Here is an example:

```
CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_narrow(current);
// End the session context, returning immediately in the case
// that there are other threads with the session context active.
ISessions::Current->endSession(ISessions::EndModeCheckPoint,1);
```

Full example

The following example demonstrates how a single-thread client can begin a session, perform some work, and then end the session, checkpointing any non-transactional work that occurred within the session.

```
CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_narrow(current);
// Begin a session context.
ISessions::Current->beginSession("LifeInsuranceApplication");
    try {
        // ... do the methods that will be implemented under the
        // session ...
    }

    catch (ISessions::SessionResetForced) {

        // The session was forced to reset mid-stream, probably the
        // session timeout tripped, or a session resource
        // encountered a significant error and had to force the session
        // reset.

    };

// End the session context, including checkpointing any activity that
// occurred during the session.

try {
    ISessions::Current->endSession(EndModeCheckpoint,1);
}
catch (ISessions::DifferentEndModeForced) {

    // The end-session was forced to reset instead of checkpointing.

}
catch (ISessions::IncompleteProcess) {

    // Something failed to checkpoint. Identify the resource that
    // failed. Try opening a new session, driving
    // the set of changes to that resource again, and attempt to
    // checkpoint the session again.

}

catch (ISessions::SubThreadPending) {

    // Something in the work that was done must have spawned an
    // asynchronous thread which is operating within the session and
    // has not completed yet. Wait some more time to give the thread a
    // chance to complete, and try to end the session again. This can
```

```

        // be retried a number of times (although look out for the session
        // timeout tripping). Eventually, if the thread never ends, you
        // will have to force the session to reset.
    }

    catch (ISessions::TransactionPending) {

        // Something in the work that was done must have started a
        // transaction and has not ended it. This could be a programming
        // error, although it could happen in an asynchronous thread that
        // has not completed yet. Try taking the same action as with
        // SubThreadPending.

    };

```

Suspending and resuming a session

There may be occasions when you want to switch the session under which you are operating. You can do this by suspending the current session, and starting a new one. Later, you can resume the original session. Each session is represented by an `ISessions::Control` object. By maintaining separate references to each `Control`, you can multiplex multiple sessions. The sessions can be resumed and suspended each for the period under which you want to operate.

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287
- “Client use of the Session Service” on page 290
- “The relationship between transactions and sessions” on page 289

You can suspend a session with the `suspendSession()` operation on the session `Current`, using the following steps:

1. Access the `ISessions::Current` as described previously. You may also use an `ISessions::Current` object obtained from a previous operation.
2. Invoke the `suspendSession()` operation on the `ISessions::Current` object.
3. Store the returned `Control` object to be used later to resume the session context.

Here is an example:

```

CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
ISessions::Control_var sessionControl;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");

```

```
ISessions::Current = ISessions::Current::_narrow(current);
// Suspend the session context.
sessionControl = ISessions::Current->suspendSession();
```

You can resume the session with the `resumeSession()` operation on the session `Current`, using the following steps:

1. Access the `ISessions::Current` object as described previously. You may also use an `ISessions::Current` object obtained from a previous operation.
2. Identify the `ISessions::Control` object which represents the session context to be resumed.
3. Invoke the `resumeSession()` operation on the `ISessions::Current` object, passing the `Control` object.

Here is an example:

```
CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
ISessions::Control_var sessionControl; // obtained from elsewhere.
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_narrow(current);
// Resume the session context.
ISessions::Current->resumeSession(sessionControl);
```

Explicit and implicit propagation of session context

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287

If you begin a session, then the session will automatically be propagated with any method request you invoke on any other managed object, regardless of whether the targeted object is local or remote. This is referred to as *implicit session context propagation*.

There may be occasions when you want to propagate a session context between threads, outside of a method request. You can do this with explicit session propagation. You can get a session `Control` object, representing a particular session context, using the `getSessionControl()` operation on a session `Current`. You can then pass a reference to this `Control` object to another thread of execution and use that as input to the `resumeSession()` operation. This will register the other thread’s participation in the session. This is referred to as *explicit session propagation* and can be performed inside or outside a method request.

Note that methods with the following names do *not* have session context propagated on them:

- `_is_a`
- `_non_existent`
- `getSessionCoordinator`
- `getSessionName`
- `registerResource`
- `getSessionContext`
- `decrementUseCount`
- `incrementUseCount`
- `prepareToEndSession`
- `prepareToCheckpointSession`
- `prepareToResetSession`

Managed object writers should avoid using any of these names for their methods.

Notice that the following is a perfectly legal scenario: Thread-1 has an active session and suspends this session using the `suspendSession()` operation. This returns a `Control` object representing the suspended session. It then begins a new session and invokes a method on some object, passing the original session `Control` as an argument. The suspended session is explicitly propagated, while the currently active session is implicitly propagated.

The session remains active on that thread of execution, independent of the original thread, until that thread separately ends its involvement in the session, typically by suspending the session on that thread of execution.

Checkpoint and reset a session context

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287
- “Client use of the Session Service” on page 290

When appropriate, you can checkpoint your progress in a session. That is, you can issue the `checkpointSession()` operation on the session `Current`. As a consequence, any session resources registered in the session will be prompted to checkpoint their state to their corresponding persistent data system. In effect, any changes that you have made so far to the in-memory state of business objects you have used will be transformed to their persistent storage system.

After making some number of changes to the in-memory state of your business objects, you can reset the session to the last checkpointed state by invoking the `resetSession()` operation on the session `Current`. Consequently, any session resources registered in the session will be prompted to reset their state from their corresponding persistent data system. In effect, the persistent state of your objects will be transformed in to memory.

Note that certain operations that you perform on your business objects may result in implicit checkpointing, that is, forcing in-memory state to be transformed to persistent storage. Typically, this occurs in push-down methods, that is, methods that are encoded directly or indirectly to make use of legacy procedures and require or result in a certain amount of state transition in the course of those procedures. Thus, if you reset a session, the in-memory state may or may not revert back to the state you had the last time you explicitly checkpointed the session.

Note also, that in many cases where you are adapting to legacy procedures, transitions from memory to persistent storage, or vice-versa, may result in additional side-effects that may be relevant to your business logic. This phenomena may require additional adjustments to your business object implementations to consider the impact of those side-effects fully.

You can checkpoint a session context using the following steps:

1. Access the `ISessions::Current` object as described previously. You may also use an `ISessions::Current` object obtained from a previous operation.
2. Invoke the `checkpointSession()` operation on the `ISessions::Current` object.

Here is an example:

```
CORBA::Object_var object;
ISessions::Current_var ISessions::Current;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CbSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_narrow(current);
// Checkpoint the session context.
ISessions::Current->checkpointSession();
```

Full example of checkpointing and resetting a session

Checkpointing and resetting a session are simply a matter of issuing the checkpoint or reset methods as follows:

```
try {
    ISessions::Current->checkpointSession();
}

catch (ISessions::IncompleteProcess) {

    // Something failed to checkpoint. Identify the resource that
    // failed. Try opening a new session,
    // driving the set of changes to that resource again, and attempt to
    // checkpoint the session again.

}

catch (ISessions::SubThreadPending) {
```

```

// Something in the work that was done must have spawned an
// asynchronous thread which is operating within the session and
// has not completed yet. Wait some more time to give the thread a
// chance to complete, and try to end the session again. This can be
// retried some number of times (although look out for the session
// timeout tripping). Eventually, if the thread never ends, you will
// have to force the session to reset.

};

```

Registering sessionable resources

Normally, the Component Broker run time automatically registers relevant session resources in a session. However, any application can introduce and register their own session resources with the Session Service. A distinction is made between a `SessionableObject` and a session Resource. While these typically represent essentially the same entity, the interfaces are separated to help distinguish between publishing a desire to be used within a session, and actually registering as a session resource. If a resource wants to publish a desire to be used within a session, the `SessionableObject` only advertises that a session context should be created and flowed on any method request invoked on the object. The object itself may or may not be capable of responding to session-related events, but rather may depend on other objects that do. If a resource wants to actually register as a session resource, the session Resource is capable of responding to session-related events. The same object may inherit both interfaces, fulfilling both roles. Or the interfaces could be inherited separately by two different objects, usually two closely collaborating objects.

A `SessionableObject` should register its Resource with the session Coordinator as soon as it realizes that it has been invoked in a new session. Typically this is done by performing a test at the beginning of each method on the `SessionableObject`. If the request is issued within a session context, and the session is new to the `SessionableObject` (one that it has never recognized before), then it should register its corresponding Resource object with the session Coordinator.

A `SessionableObject` can determine whether it is being invoked within a session by invoking `getSessionControl` on an `ISessions::Current`. If the request raises the `NoSession` exception, then the request was not invoked in a session. Otherwise, the request should return a `ISessions::Control`. The sessionable object can then determine whether the session is the same as one they have seen before by getting the session Coordinator (by invoking the `get_coordinator` on the session Control), and then invoking `isSameSession` on the Coordinator.

The `isSameSession()` operation takes a `Coordinator` as an argument. The sessionable object should retain a reference to all of the `Coordinators` for each of the unique sessions it is currently participating in. Note that the same sessionable object could be invoked by multiple clients each under their own session context. Thus, the sessionable object can repeat the `isSameSession` request, passing in each of the `Coordinators` of the sessions it is participating in. If the session (`Coordinator`) is not the same as any it already knows of, then the session is new to it. The sessionable object should then register its `Resource` object and then add the `Coordinator` to its list of sessions in which it is participating. Later, when the session completes, issuing the `endResource()` operation on the `Resource`, the sessionable object (through a private collaboration with its `Resource`) should remove the `Coordinator` representing that terminated session from its list.

Before performing this task, make sure you are familiar with:

- “Resource priorities” on page 290
- “Client use of the Session Service” on page 290
- “Managed object developer use of the Session Service” on page 292

You can register a `Resource` object for the session using the following steps:

1. Access the `ISessions::Current` object as described previously. You may also use an `ISessions::Current` object obtained from a previous operation.
2. Obtain the `ISessions::Control` object which represents the session context using `getSessionControl` on the `ISessions::Current` object. If this operation throws the `NoSession` exception, then there is no session context with which to register the `Resource` object.
3. Obtain the `ISessions::Coordinator` object that represents the session context, using `getSessionCoordinator` on the `ISessions::Control` object.
4. Determine whether the `Resource` has already been registered with this session context, using `isSameSession` on the `ISessions::Coordinator`. If the `Resource` has already been registered, then there is no need to do so now.
5. Obtain a reference to the `Resource` object to be registered. This can be done by either creating a new instance, or by reusing an existing `Resource`.
6. Decide on the priority value for the registration.
7. Invoke the `registerResource()` operation on the `ISessions::Coordinator` object, passing the `Resource` object and priority value.

Here is an example:

```
CORBA::Object_var object;  
ISessions::Current_var ISessions::Current;  
ISessions::Control_var sessionControl;  
ISessions::Coordinator_var sessionCoordinator;  
ISessions::Resource_var myResource;  
int myPriority;
```

```

// Get the current for this thread of execution
// from the ORB and narrow to the session current.
object = CBSeriesGlobal::orb()->resolve_initial_references(
    "ISessions::Current");
ISessions::Current = ISessions::Current::_ (current);
// Obtain the session Control object.
// This will throw NoSession if there is no current session context.
try {
    sessionControl = ISessions::Current->getSessionControl();
}
catch (ISessions::NoSession) {

    //skip over the remaining code

}
// Obtain the session Coordinator object.
sessionCoordinator = sessionControl->getSessionCoordinator();
// Determine whether already registered.
// This will involve using ISessions::Coordinator::isSameSession.
...
// Register the Resource
sessionCoordinator->registerResource(myResource,myPriority);

```

Collaborating on session outcome using multiple concurrent threads

Before performing this task, make sure you are familiar with:

- “The scope of sessions” on page 287
- “Client use of the Session Service” on page 290

In some cases, particularly in window-based client programs, it is necessary to manipulate resources (business objects) from multiple threads, all within the same session context. The Session Service makes it relatively easy to do this. As described in Explicit and implicit propagation of session context the same session context can be propagated between threads by passing the session Control representing the session context between threads, and then resuming it on the subordinate thread using the resumeSession request on an ISessions::Current.

At least one thread must issue an endSession request to establish the outcome of the session. All of the other threads can simply issue an endSession request or a suspendSession request on a Current to end their involvement in the session. The session will not actually terminate until the last thread has issued either an endSession or a suspendSession request. If more than one thread issues the endSession request, the final outcome of the session is determined by the total consensus of all of the threads issuing the endSession request. For the session to be checkpointed, all of the threads must request to end with EndModeCheckpoint. If any thread requests to end with EndModeReset or EndModeResetForce, the session will be reset.

Note that collaboration of this sort can only be performed amongst threads in the process of the top-level coordinator.

Chapter 11. Query Service for AIX and Windows NT

The Query Service enables you to query for a set of objects that satisfy a set of conditions that you specify. Performing a query using the Query Service is conceptually very similar to performing query on a relational database. It differs in that the Query Service query is performed on a collection of objects rather than a collection of records, and the predicate is formed on the set of attributes and method return values for the object rather than on columns in the tables.

You will often find that you are dealing with very large collections in your business application. The collections that you deal with may have hundreds of thousands, millions, or even billions of object instances. Iterating through the entire collection of objects, looking for the one or few that you want to work with, can be enormously expensive, in terms of system resources. You can use the Query Service to preselect the set of objects that you want to work with. The Query Service will produce a subset of the original collection that satisfies the conditions that you set. If you only want to work with insurance policies that have coverage of more than a million dollars, you can form a query to return only those Policy objects that satisfy that condition, and then iterate on just those few.

The Query Service introduces a new query language: object-oriented SQL (OOSQL). OOSQL is an extension to the SQL language with additional constructs for operating on objects instead of tuples. OOSQL is described in detail in the Component Broker reference section.

Queries are actually performed by a query evaluator. The query evaluator understands the OOSQL grammar and how to apply it to collections of objects to form the requested result. In some cases, the query evaluator is able to push queries all the way down into the underlying datastore for a collection of objects. In this case, the query can be performed in the datastore and can yield significant performance improvements.

Certain kinds of collections, termed queryable collections, are able to support query operations directly. The collection, in turn, will locate an appropriate query evaluator and use that to form the results that it hands back to you.

Depending on the needs of your application, queries have two different kinds of results. One result type is a collection of objects of a single type. For instance, if you are performing a query on a collection of Policy objects, then you probably want a result set of Policy objects. Alternately you can project in

the query statement attributes of the object or expressions involving object attributes and this is returned in the form of a collection of data array records.

The Query Service locates collections referenced in the FROM clause of any query statement by looking up the referenced collection name in the system name space. Alternately the caller can specify a list of Name/Value pairs on the query method that define the collection in the FROM clause.

Object-Oriented Structured Query Language

The language for the Component Broker Query Service is object-oriented Structured Query Language (OOSQL). OOSQL is a Query Service over objects where the syntax of the query is expressed in standard or extended forms of the Structured Query Language (SQL).

Differences between OOSQL and SQL

This section contains a brief review of SQL showing how OOSQL differs from SQL. For more information on SQL see the SQL Reference Manual and Application Programming Guide in the DB2 product information.

SQL is a Structured Query Language designed for use with relational databases. Use the following employee and department tables, you can perform queries to find specific data.

Table 13. Employee

empid	name	deptno
12	'Dave'	42
14	'Andrew'	42
16	'Liz'	44
18	'Amy'	44
20	'Don'	44

Table 14. Department

deptno	name	mgrid
42	'Sales'	16
44	'Dev'	20

You can find all employees in department 42:

```
select empid,name from employee where deptno=42
```

You can find all employees whose manager has the ID of 16. This query, however, requires a join of both tables.


```
select e.empid, e.name from employee e, dept d
where d.deptno=e.deptno and d.mgrid=16
```

You can find all employees that are not managers:

```
select empid, name from employee where empid not in
(select mgrid from dept)
```

You can find the number of employees in each department:

```
select deptno, count(*) from employee group by deptno
```

OOSQL is an extension of SQL. Instead of tables used in SQL, data takes the form of collections of objects with attributes and methods.

empHome is a home collection of employee objects with the interface:

```
interface employee{
    attribute readonly long empid;
    attribute string name;
    attribute dept deptPtr;
}
```

deptHome is a home collection of dept objects with the interface:

```
interface dept {
    attribute readonly long deptno;
    attribute string name;
    attribute employee mgr;
    IManagedCollection::Iterator emps();
}
```

Unfortunately IDL does not tell you what kind of objects the method emps() returns. Assume that these are employee objects.

OOSQL queries equivalent to the previous queries would be:

```
select e.empid,e.name from empHome e where
e.deptPtr->deptno=42;
```

This query returns the values of *empid* and name for employee objects in department 42. It is called a data array query.

Some important points to remember:

- OOSQL queries always end with a semicolon.
- The FROM clause names the collection. (Later you will see how to query an unnamed collection.) The name of the collection is the name that the home uses in the DCE namespace. Be aware that home collections have two names, a home name and a factory finder name. The FROM clause always uses the home name, never the factory finder name.

- The (->), or (..) is a dereference operator in OOSQL. These operators function in a similar manner as the right arrow (->) operator in C++. The dereference operator can be used with data types that are object references. A path expression must always begin with a correlation identifier (the letter “e” in the previous example).

If you want to return object references instead of attribute values the query would be:

```
select e.OID from empHome e where e.deptPtr->deptno=42;
```

This is called a reference query. A reference query is a special case of a data array query where there is only one element in the SELECT clause and that element is an object reference.

A correlation name followed by a period and the keyword *OID* returns pointers to objects in the collection associated with the correlation name. If the correlation name is not one of the members in the collection associated with the correlation name, the correlation name and has the same semantics as correlation name preceded by the keyword *REF*. However only the right arrow (->) is used to traverse through *OID*. This feature was introduced to preserve compatibility with SQL in which column names can be unqualified. For example, to select pointers to employee objects with the name of 'Bob' from the empHome collection:

```
select e.OID from empHome e where name='Bob';
```

If the collection empHome has no member attribute “e”, then the previous query is the same as the following query:

```
select e from empHome e where name='Bob';
```

You can return attribute values and object references:

```
select e.empid, e.name, e.OID from empHome e
where e.deptPtr->deptno=42;
```

This is considered another type of data array query.

To find all the dept objects where the deptno is between 10 and 100 the query would be:

```
select x.OID from deptHome x where x.deptno between 10 and 100;
```

The query to find all non-manager employee objects would be:

```
select e.OID from empHome e where e.empid not in
(select d.mgr->empid from deptHome d );
```

A count of employees in each department would be performed as follows:

```
select e.deptPtr->deptno, count(*) from empHome e
group by e.deptPtr->deptno;
```

You can perform string searches using the SQL LIKE operator:

```
select e.OID from empHome e where e.name like 'Bob%';
```

“%” is the wild card character in SQL. SQL strings are delimited by single quotes where strings in C++ are delimited by double quotes.

Note: String searches are case sensitive in SQL.

If the dept interface also includes methods, I can also include methods in my queries.

```
interface dept {
    attribute readonly long deptno;
    attribute string name;
    attribute employee mgr;
    double compute_overtime();
    long compute_vacation(in long year);
};
```

Find all dept objects where the overtime is greater than 10 hours:

```
select d.OID from deptHome d where d.compute_overtime() > 10;
```

Find the deptno, name and vacation days of dept objects where vacation in 1996 was less than 50 days.

```
select d.deptno, d.dname, d.compute_vacation(1996)
from deptHome d where d.compute_vacation(1996) < 50;
```

The key points of OOSQL queries is that they:

- Are similar to SQL queries.
- Use collection names in the FROM clause (SQL queries use table names).
- Can return object references as well as attribute values (SQL queries can only return column values).
- Have a dereference operator (->), or (..) that can be used to follow object references.
- Can do joins, subselects, ordering and summarize data just like SQL.
- Can use object attributes and methods in the select and where clause. Only methods that return a value and have either no parameters or only input parameters can be use in a query statement.

Methods

OOSQL supports invocation of CORBA IDL methods in queries. (Methods are also referred to as member functions.) In an OOSQL query, a method name is followed by the method arguments within parentheses. For example,

$m(a_1, a_2, \dots, a_n)$ is a method with name **m** and arguments a_1, a_2, \dots, a_n . Following the C++ convention, methods with no arguments are followed by empty parentheses: $m()$.

Component Broker implements a dynamic run-time environment for method selection and invocation.

If a method argument is a null value, then the value returned by the method is also null.

Methods must be defined as having zero arguments or input only arguments and must not be defined with a return type of void.

The implementation of methods appearing in queries has limitations. Method arguments in OOSQL statements are checked for type correctness. Where possible, when method execution results in exceptions, a method failure message is generated and the query is terminated. In some cases, a programming error in a user's method might cause the query engine to halt loop or terminate abnormally.

The following table presents the conversion that is performed when an argument of a given type is passed to a method that is defined with a parameter of the same or different type.

The table shows for example, if you have the IDL interface

```
myInterface {
    attribute short s1;
    attribute string s2;
    long methoda(in long input);
}
```

the query statement

```
select e.methoda(e.s1) from myHome e;
```

is valid and the Query Service will convert $s1$ from short to long when calling the method. However the query statement

```
select e.methoda(e.s2) from myHome e;
```

is not valid because the string type attribute $s2$ cannot be passed to `methoda`.

argument type/ parameter type	pointer	short	long	float	double	string	wstring
pointer	NC	E	E	E	E	E	E
short	E	NC	C	C	C	E	E
long	E	C*	NC	C	C	E	E

argument type/ parameter type	pointer	short	long	float	double	string	wstring
float	E	C*	C*	NC	C	E	E
double	E	C*	C*	C*	NC	E	E
string	E	E	E	E	E	NC	C
wstring	E	E	E	E	E	E	NC

C Conversion. The argument type is converted to the parameter type.

C* Conversion with the possibility of an overflow, or an underflow due to a type conversion.

E Error. The argument type used in a method in a query is not applicable to the parameter type of the method.

NC No Conversion. The argument type and the parameter type are the same.

Inheritance

OOSQL supports interface inheritance as in the following example. Suppose the manager interface inherits from employee.

```
interface manager : employee {
    attribute dept manages_deptPtr;
    attribute long executiveLevel;
}
```

A query statement over manager can select inherited attributes just like noninherited attributes. (No special syntax is required.)

```
select m.no, m.name, m.executiveLevel from managerHome m;
```

Navigation

A navigation is specified by path expressions. Path expressions allow traversal through references, embedded structures, and collections to reach embedded members. The (->), or (..) operator is used to express traversal through embedded members. Path expressions can appear anywhere a member can. A path expression is q.m1..m2..mn where q is a correlation name defined for collection C, and m1 is a member of the element type of C, and m2 is a member of the type of m1 and so on. A member of mi can be a member or a method. A path expression evaluates to the value of the leaf of the expression.

Through embedded structures

Embedded structures members can be defined in terms of structures. Navigation allows traversing into the embedded members of structure

definitions. Use two periods (..) as an operator only for traversal through embedded structures. The right arrow (->) operator CANNOT be used with structures.

Example:

```
struct addressStruct {
    string street;
    string city;
    string state;
    string country;
    string zip;
}

interface employee {
    attribute long empid;
    attribute addressStruct address;
}
```

You can write the query statement

```
select e.address, e.empid, e.address..city from empHome e
where e.address..zip='95120' order by e.address..city;
```

returns the address struct, employee ID and city for employees in postal code 95120 sorted by city.

Through references

Reference members can also participate in navigational expressions. A reference that has a zero value is treated as a null reference. Only references to objects, can be traversed (that is, appear as other than leaf nodes in path expressions). Uninitialized or invalid references can cause OOSQL to terminate abnormally if these are part of a path expression that is traversed. The operator (->), or (..) is used for traversal through reference types. However, only the right arrow (->) is used to traverse through OID.

Through collections

Multi-valued relationship members can be used as collections in a FROM clause as in the query

```
select d.OID from deptHome d , (d.emps) e where e.empid > 1000 ;
```

This query joins each department object in the department Home collection with its related set of employee objects. The semantics for join are used to compute the final result set. For each employee whose *empid* > 1000 a reference to the department object is returned. If a department has a set of employees with *empid* equal to (20, 1001, 1009, 1020) then the OID for that department would be returned three times. If a department has no employees, then the OID for that department would not be returned.

Multi-valued members can also be used in predicates as in the query

```
select d.OID from deptHome d where d.emps .. empid > 1000 ;
```

This query has the same meaning as the query above. The path expression

```
d.emps .. empid
```

is converted into an equivalent join query.

Navigation through multi-valued relationship members creates implicit correlation names over each embedded collection in the path expression.

You must use two periods (..) as an operator when navigating through multi-valued members. The right arrow (->) operator cannot be applied to a multi-valued member.

Multi-valued members can also be used as a short hand notation for a sub-select expression as in the query

```
select d.OID from deptHome d where exists d.emps ;
```

The following query has the same meaning as the previous query:

```
select d.OID from deptHome d where exists (select 1 from (d.emps) e) ;
```

Both queries return references to department object where there is one or more employees in the department.

The next query illustrates using a quantified predicate with a path expressions. Remember that quantified predicates take as an argument a set of values.

```
select d.OID from deptHome d where 2000 < some d.emps..empid ;
```

This query is equivalent to

```
select d.OID from deptHome d where 2000 < some (
  select e.empid from (d.emps) e) ;
```

Note the difference here from the earlier examples. The earlier examples implicitly converted the path expression into a join query. In this example, the path expression when used with a quantified predicate converts the path expression into a sub-select. A join query will exclude departments with zero employees and an department with more than one employee may appear in the result multiple times. Sub-select queries used with quantified predicates have different semantics.

The previous query returns references to department objects where some of the employees in the department have an *empid* > 2000. If the department has

zero employees, it will not appear in the final result. A department will appear at most once in the final result even if there are multiple employees with *empid* > 2000.

Next let's look at path expressions used with aggregation functions. The following query is the sum of salaries for each department.

```
select d.deptno, sum(d.emps..salary)
       from deptHome d
       group by d.deptno ;
```

It is equivalent to:

```
select d.deptno, sum(e.salary)
       from deptHome d , (d.emps) e
       group by d.deptno ;
```

A department with zero employees will not appear in the result set.

The following query does a count of employees in each department. A department with no employees will not appear in the result.

```
select d.deptno, count(d.emps..id)
       from deptHome d
       group by d.deptno;
```

An equivalent query is:

```
select d.deptno, count(*)
       from deptHome d, (d.emps) e
       group by d.deptno ;
```

The following query does not count the number of employees in each department, but rather counts the number of sets of employees in each department. *Emps* represents the collection of employees in each department and there is exactly one such collection for each department.

```
select d.deptno, count(d.emps) from deptHome d group by d.deptno;
```

Multi-valued path expressions can also be used in the SELECT clause as in the query

```
select d.emps .. empid , d.emps .. name
       from deptHome d where d.deptno = 1 ;
```

The following query is an equivalent query that uses a join.

```
select e.empid, e.name from deptHome d, (d.emps) e
       where d.deptno = 1 ;
```

This returns a data array consisting of the *empid* and name values for employees in department 1. If there are no employees in department 1, the result of the query is empty because of the inner join operation.

In the following query, the *emps* is defined in the IDL schema as an Iterator, and will return a result set that consists of Iterators. If a department has multiple employees, there will only be one Iterator in the result for each department. If a department has no employees, then there will still be an Iterator in the result but the Iterator will return an empty set.

```
select d.emps from deptHome d where d.deptno in (1, 2) ;
```

Using a multi-valued relationship in a SELECT clause has the semantics of outer join. However applying the dereference operator of two periods (..) to the relationship changes the semantics to an inner join.

One final explanation about path expressions when you use a path expression to navigate between objects in different application servers. Path expressions can only be used when the related collections exist in the same server. Using a path expression to navigate between objects that reside in different servers will result in an exception.

Query optimizations

In general, the semantics of any query requires that the Query Service invoke the methods or attributes specified in the query expression, perform any mathematical operations on the resulting values as specified in the query expression, and determine whether the final result satisfies the specified predicate for each object in the collection. Due to the inherent encapsulation semantics of object-oriented programming, this requires that the Query Service activate each object instance in the collection and perform the evaluation one object at a time.

However, if the collection(s) you are querying is collected by a home, and that home is configured to store the state of its objects in a relational database then the Query Service can optimize its search. The query evaluator is smart enough to detect that the collection(s) it is operating on may be adapted to a relational data store. When this is the case, and when there is a straightforward mapping between the attributes specified in the query, and the underlying columns of the tuple in which the managed objects collected by the home are stored, then the query evaluator will transform the OOSQL expression into a standard SQL expression and “push down” the query to examine the underlying datastore. The Query Service supports “push down” for DB2 and Oracle data stores.

Even if only part of the OOSQL expression can be mapped in this way, the resulting push down will often yield an intermediate result set that is a fraction of the original collection size. Thus, the remaining query can iterate over the intermediate subset and significantly reduce the overhead that would be incurred if the evaluation had to iterate over every object in the collection.

If the collection being queried is collected by a home that is configured to store object state data in a procedural system such as CICS or IMS using the Component Broker Procedural Adapter a mapping between the object attributes and procedures may be created. If such a mapping exists then the query evaluator will transform the OOSQL expression into a series of procedural calls on the backend store. For example, if a CICS transaction T1 that given a partial employee name, will return a list of employees with that name, the OOSQL query

```
select e.OID from empHome e where e.name like 'Smith%';
```

will result in calling T1. The data returned by T1 will used to build a series of employee objects that will returned by the query operation.

Query push down on PAA backed home collections can also be partial. Given the OOSQL query

```
select e.OID from empHome where e.name like 'Smith%' and e.salary > 25000;
```

transaction T1 will be called to first build a intermediate result which will then be further refined to return only those objects where salary > 25000.

For more information about configuring query and PAA backed home collection, see the “Flow diagram” section in the *WebSphere Application Server Enterprise Edition Component Broker CICS and IMS Application Adaptor Quick Beginnings*.

DBMS pushdown rules

The query pushdown determines what parts of the OOSQL query are passed down to the underlying relational database management systems (DBMS) where the data resides. The parts of the OOSQL query that are not pushed down are evaluated in the memory. This section lists which of the OOSQL query constructs are pushed down and which are not for the DB2/390, DB2 Universal Database (UDB) and Oracle database management systems.

Simple expressions: All DBMS types: The expressions containing numeric, date, time, and timestamp functions, arithmetic operators and comparison operators appearing in predicates are pushed down to the DBMS. Comparison operators among {<, ≤, >, ≥} are not pushed down for string or Wstring types.

Joins: All DBMS types: Multiple tables from the same database can be pushed down in a single query if the table are related by an equi-join predicate.

Aggregates, group by clause, having clause: DB2/390, Universal Database (UDB): Aggregates, group by and having can be pushed down provided that all other clauses in the query can be pushed down.

Oracle: Aggregates, group by and having can be pushed down provided that all other clauses in the query can be pushed down and that only columns appear in the projection list.

Distinct: All DBMS types: Pushdown is disabled.

Order by: All DBMS types: Pushdown is disabled.

Union: All DBMS types: Pushdown is disabled.

Subqueries: All DBMS types: If the tables participating in a subquery are from the same database as the tables participating in the outer query and the body of the subquery can be pushed down, the following rules apply: Exists (existential) subqueries are pushed down. Subqueries that are basic predicates, and ANY (existential) and ALL (universal) subqueries are pushed down if the comparison operator is in the set {=, ≠} or if the comparison operator is among {<, ≤, >, ≥} and the operands of the operator are numeric, date, time, timestamp (arguments of type string or Wstring disable pushdown).

Projection (query without aggregates, group by and having): All DBMS types: Projection clauses in subqueries can be pushed down. The projection clause of the outer query is computed in the memory.

Projection (query with aggregates, group by and having): DB2/390.UBD: The projection list can be pushed down (see the following aggregate function list for details on when pushdown occurs).

Oracle: Pushdown is disabled if anything other than columns appear in the projection list.

Scalar functions:

char

Universal Database (UDB):

The CHAR function can be pushed down for arguments of type string, short, long, decimal, date, time, timestamp. Pushdown is not applied for arguments of type 4 byte float, double, duration due to formatting differences.

DB2/390:

Same as Universal Database (UDB) except that string arguments cannot be pushed down.

Oracle:

The CHAR function is pushed down using the Oracle to_char function for arguments of type short, long, and decimal. Note that these numeric types map to the Oracle number type with a certain precision and scale.

integer

Universal Database (UDB):

The INTEGER function can be pushed down for string and numeric arguments.

DB2/390:

Same as Universal Database (UDB) except that varchar arguments cannot be pushed down.

Oracle:

Pushdown disabled.

digits, float, decimal, year, month, day, hour, minute, second, date, time, timestamp, microsecond

DB2/390, Universal Database (UDB):

Pushdown enabled.

Oracle:

Pushdown disabled.

smallint, double

Universal Database (UDB):

Pushdown enabled.

DB2/390, Oracle:

Pushdown disabled.

Aggregate functions:

count, sum, avg

DB2/390, Universal Database (UDB):

Pushdown enabled.

Oracle:

Pushdown is enabled if aggregates are not in the projection list (e.g., in a having clause or in a subquery).

min, max

All DBMS types:

Pushdown is enabled for numeric, date, time, timestamp arguments, and disabled for string or Wstring arguments.

Miscellaneous rules:

Methods in queries

All DBMS types:

Query terms that are methods are not pushed down.

Queries with reference collections

All DBMS types:

Query terms that are reference collection attributes are not pushed down.

Queries with in-memory object building**All DBMS types:**

Query terms that are reference collection attributes are not pushed down.

Path expressions over home collections**All DBMS types:**

Path expression over home collections can be pushed down as join expressions.

PAA pushdown rules

The query pushdown for PAA determines what parts of the OOSQL query are passed down to the underlying PAA where the data resides. The parts of the OOSQL query that are not pushed down are evaluated in the memory. The candidate operations for pushdown depend on the underlying PAA application. This sections lists which of the OOSQL query predicates can be pushed down.

Simple numeric expressions

The simple numeric expressions of the form *attribute comparison-operator numeric-constant* where the comparison-operator can be one of the operators =, >, <, ≥, ≤.

Simple string expressions

The simple numeric expressions of the form *attribute comparison-operator string-constant* where the comparison-operator can be one of the operators =, LIKE.

All the other OOSQL query constructs are not pushed down.

The cast operator

The cast operator sets the type of a member or method in the body of a query. Thus casting provides type information to OOSQL when it is not available from the schema.

The keywords *CAST* and *AS* is used to specify the casting. The characters (%) and %) can also be used by following the member to be altered and specifying the type name between (%) and %).

Casting is a fragile operation, since casting a member to the wrong type may result in producing incorrect answer to the query or can cause OOSQL to fail. See Query over reference collections for an example of casting.

Query over reference collections

The syntax of a query over a reference collection is the same as query over a home collection.

There is one important difference when querying an object whose IDL definition contains an `ICollection` or `Iterator` type as an attribute or return type of a method. The IDL definition does not indicate what kind of object the `ICollection` or `Iterator` references. The Query Service must know this information in order to process the query and so it necessary to indicate this information using a cast function in the query statement itself.

For example, using the `dept` interface from the previous section, the query over a home collection to find all departments that contain an employee whose name starts with 'D' would be:

```
select d from deptHome d where d.emps..name like 'D%';
```

To do this same query over a reference collection (whose name is `deptRC`) of department objects would be:

```
select d from deptRC d where CAST(d.emps AS set (
    ref (employee)))..name like 'D%';
```

The previous query is the same as the following:

```
select d from deptRC d where d.emps(
    %Collection<:employee*>%)..name like 'D%';
```

The notation `CAST(d.emps AS set (ref(employee)))` is the OOSQL cast that specifies the object type of `emps`.

The following queries that count the number of employees in each department are the same.

```
select CAST(e.deptPtr AS ref(deptClass))->deptno,count(*) from
    empHome e group by CAST(e.deptPtr AS ref(deptClass))->deptno;
select e.deptPtr(%deptClass%)->deptno,count(*) from empHome e
    group by e.deptPtr(%deptClass%)->deptno;
```

The internal processing for a query over a reference collection is different from a home collection. A query over a reference collection is processed by iterating over the reference collection and activating each object (if it is not already activated) and evaluating the query.

When creating the reference collection, make sure you use the `createCollectionFor()` method and pass in the string equal to the IR name of the objects the collection will contain. For example:

```
createCollectionFor("IDL:policy:1.0");
```

You can find out what this IR name is by loading the IR and then doing an `irdump` with a parameter of your interface name.

If you do not use the `createCollectionFor()` then you must supply the interface name in the query statement `FROM` clause as in the example:

```
select r.OID from MyReferenceCollection.acct r where r.Name = 'Bob';
```

In this example `acct` is the interface name of the objects in the collection.

Data type mapping between DB2 and CORBA

Many of the DB2 data types have obvious counterparts in CORBA such as DB2 integer mapping to `CORBA::long` and DB2 `char(n)` and `varchar` mapping to `CORBA::string`.

Be aware that `char(n)` and `varchar` will map to null terminated strings in CORBA. Binary string data can be handled by using either `VARCHAR FOR BIT DATA` or `CHAR FOR BIT DATA` in the table definition mapped to `ByteString` data type in IDL.

There are four data types in DB2 that do not have obvious counterparts in CORBA. They are DB2 Date, Time, Timestamp and Decimal. Date, Time and Timestamp should be mapping to `CORBA::String`. There are helper classes (`ICBCDate`, `ICBCTime` and `ICBCTimestamp`) provided in Component Broker if you need to do date and time manipulations of these strings. Decimal can be mapped to `CORBA::double` or `CORBA::String`. Use string when you need exact precision. Double byte strings can be stored using `GRAPHIC` or `VARGRAPHIC` data type in the table definition and `Wstring` in the IDL.

Query evaluators

A query evaluator is the engine behind the Query Service. The query evaluator parses the query expression you supply, locates the inferred collections, and evaluates the collections for the set of objects that satisfy the query predicate. The query evaluator supports the `QueryEvaluator` interface as defined in the standard CORBA services Query Service specification.

The `evaluate()` operation, as specified by OMG in the `CosQuery::QueryEvaluator` interface returns an `any`. CORBA leaves it to query implementers to specify the structure of this `any` for the various types of results that can be produced from a query. Depending on the conditions you specify in the query expression you could potentially need to get back either an iterator (a collection of objects whose type matches the objects in the collection you are evaluating), or a data array. However, the Component Broker implementation of the `CosQuery::QueryEvaluator::evaluate()` operation

returns only an `IManagedCollections::Iterator`. You need to test the results returned to ensure they match your expectations.

Component Broker has extended the `CosQuery::QueryEvaluator` interface to introduce variations of the `evaluate()` method that return more specific result set types. If you want to be more specific about the type of result set you get, you can use the `evaluate_to_iterator()` operation which passes back an `IManagedCollections::Iterator`, or you can use the `evaluate_to_data_array()` operation which passes back an `IExtendedQuery::DataArrayIterator`.

In general, a query evaluator could support any number of query languages. You would be expected to specify the language that you intend to use, and accept that the query evaluator you choose may or may not be implemented to support that language. The Component Broker query evaluator only supports one language--object-oriented Structured Query Language (OOSQL). OOSQL is a rich language, following in the tradition of SQL, with extensions that are specific to object-oriented programming.

As a programming convenience queriable collections (namely, home collections, view collections and reference collections) support operations `evaluate()`, `extendedEvaluate()` and `extendedEvaluateToArray()`. `ExtendedEvaluate` and `extendedEvaluateToArray` are similar to `evaluate_to_iterator()` and `evaluate_to_data_array()` operations on the query evaluator object. `Evaluate` is a simplified form of `evaluate_to_iterator`. By using these methods, you do not have to locate the default query evaluator object for a server.

Default query evaluator

Component Broker automatically creates an instance of a query evaluator in every Component Broker server, and binds this in the system name space at the following location:

```
/host/resources/servers/<server-name>/query-evaluators/default
```

Plug your server name in for `<server-name>`. If your program is executing on a server, you can get the name of your local server from the `CBSeriesGlobal::serverName` static member function. This member function can only be invoked within a server process. In a client process, you must know the name of the server containing the query evaluator you want to use.

Obtain a query evaluator

The following procedure demonstrates how to obtain the default query evaluator from the system name space for a well-known server. This procedure can only be completed if you know the name of the server on which the query evaluator exists. The query evaluator you obtain should be on the same server as the collections that you will be querying.

1. Determine which server you want to use.
You need to know the name of the server that contains the query evaluator that you want to use. This could be the server on which you're already executing, or it could be a remote server. In the former case, you can simply get the local server name from the `CBSeriesGlobal::serverName` static member function.
2. Resolve the default query evaluator from the system name space.
Use the Naming Service operations to resolve the default query evaluator in the named server from the system name space.

The following example obtains the default query evaluator from the local server. This example can only be used in a server process—for instance, in a business object implementation.

```
// Declare an intermediate object ref, intermediate naming contexts
// and the targeted query evaluator

CORBA::Object_var intermediateObject;
IExtendedNaming::NamingContext_var serversNC;
IExtendedNaming::NamingContext_var localServerNC;
IExtendedQuery::QueryEvaluator_var defaultQE;

// Resolve to the server's naming context in the Host name space
serversNC = CBSeriesGlobal::nameService()->resolve_with_string(
    "/host/resources/servers");

// Resolve to the local server
localServerNC = serversNC->resolve_with_string(
    CBSeriesGlobal::serverName());

// Resolve and narrow to the default query evaluator
intermediateObject = localServerNC->resolve_with_string(
    "query-evaluators/default");
defaultQE = IExtendedQuery::QueryEvaluator::_narrow(intermediateObject);
```

Topology of query evaluators and collections

The FROM clause of the query statement contains the name of one or more collections. The Query Service operates only on collections that have been named in the system name space or named in a parameter list which is a list of Name/Value pairs supplied on the query method.

The query evaluator looks up the collection specified in the FROM clause in the parameter list. If the name is not found in the parameter list (or there is no parameter list), and the name is a fully-qualified name, the query evaluator will use the Naming Service to find the collection. (The name is considered

fully-qualified if it contains one or more forward slash “/” characters.) When the name is not a fully-qualified name, the query evaluator will search the Naming Service namespace for the collection in the following locations in the order:

```

/host/resources/servers/<localservername>/collections
/host/resources/collections
/workgroup/resources/collections
cell/resources/collections

```

If the collection name cannot be located a query exception is raised. Collections names are not necessarily unique across the namespace. For example it is possible for /host/resources/collections/test and /workgroup/resources/collections/test to both exist but refer to different collections. The query

```
select t.OID from test t;
```

will use the collection /host/resources/collections/test because that name will be found first in the search.

Homes are normally automatically bound in the collection’s name context in the server on which the home exists as well as the host workgroup and cell contexts for the server when the home is created. You can specify whether a home is to be bound, and its collection name in the Application DDL for that home using Object Builder. If you create Views and Reference Collections, it is up to you to bind that collection in the system name space.

As previously mentioned, you can supply the collection in the parameter list, provided you give it the same name in the parameter list as you specified in the FROM clause in the query statement, as shown in the following example:

Parameter List	Query Statement
“myCollection”, IManagedCollections::IReferenceCollection_var myCollection	“select p from myCollection p where p.number > 10”

Form a query

This procedure demonstrates how you can form a query on a collection.

1. Locate either the collection or the query evaluator object. Queryable collections contain three operations which can be used to submit a query: evaluate(), extendedEvaluate() and extendedEvaluateToArray(). If using the query evaluator object the operations are called evaluate_to_iterator() and evaluate_to_data_array(). You can query a home by narrowing to its IManagedAdvancedClient::IQueryableIterableHome interface.

2. Determine the type of result you want to receive. The `evaluate`, `extendedEvaluate` and `evaluate_to_iterator()` operations returns an iterator over a collection of object references. The `extendedEvaluateToDataArray` and `evaluate_to_data_array()` operations returns an iterator over a collection of data array rows.
3. Decide if you want any initial values back from the iterator. All `evaluate` methods that you may use will return an iterator: either an iterator over a reference collection, or an iterator over a collection of data array rows. Normally you will iterate over these collections either one or several elements at a time. You have the opportunity at the time you initiate the query to ask that an initial set of elements be returned in a sequence outside of the iterator. This is a convenience mechanism that is equivalent to invoking the query, getting back the iterator, and requesting the first n elements in a separate request. The `evaluate()` operation does not support the mechanism to request an initial set of elements.
4. Issue the query request with the query statement, any accompanying parameter list, and an indication of how many initial elements to return from the resulting iterator.

The operations `evaluate()`, `extendedEvaluate()` and `extendedEvaluateToDataArray()` have been implemented to locate their server's query evaluator object and delegate the query request.

If you use the `evaluate()` operation, note that the input parameter contains only the predicate part of the query statement. The collection will form its own `SELECT` and `FROM` clause and append the predicate that you supply. The query statement will be in the form `select x.0ID from thisCollection x where` and append the predicate you supply. The correlation ID will be the character "x". The collection will create a parameter list that defines `thisCollection` as pointing to itself. For example, if you issue the operation

```
evaluate(" name='ToTo' ");
```

The equivalent query will be

```
select x.0ID from thisCollection x where name='ToTo';
```

When you use the `extendedEvaluate()` or `extendedEvaluateToDataArray()` methods on a queryable collection, the `extendedEvaluate` methods will always add the name `thisCollection` to any name/value parameter list supplied on the method call. If there is no parameter list supplied on the call, then the `extendedEvaluate()` operation will create one. The value associated with the name `thisCollection` is a pointer to the collection itself.

This means that the query

```
select e.0ID from thisCollection e where e.name='John' ;
```

if used with the `extendedEvaluate()` method on the `empHome` collection, would be equivalent to

```
select e.OID from empHome e where e.name='John' ;
```

Using the keyword *thisCollection* is better for the following reasons: it can improve performance because referring to a collection by using a parameter list instead of by home name saves a call to the Naming Service to locate the home collection and it makes writing the application easier because the programmer may not know the actual home name for a collection.

Queries that result in an object collection

When you specify your query expression, you indicate in the `SELECT` clause the type of the results you expect to get back. The result can be an object type as defined by a managed object in IDL, or some combination of one or more data types.

```
select e.OID from empHome e;
```

In the preceding statement, the result is the type of object that is collected by the `empHome` collection, presumably `Employee`. This example returns a collection of objects. This has the benefit of allowing you to perform other operations supported by `Employee` objects on any of the objects returned from these queries. You can direct the query evaluator to return a collection of objects (literally, an iterator to a collection of objects) using the `evaluate()`, `extendedEvaluate()` or `evaluate_to_iterator()` operations.

The `extendedEvaluate()` operation returns an `IManagedCollections::Iterator` object, and a sequence of zero or more initial entries from the iterator. The `Iterator` is an object that represents a collection of references to objects, and can return one or more object references, that is, entries in the reference collection. If you use the `next()` operation on the `Iterator`, it will return the next object reference in the collection. If you use the `nextS()` operation, you can specify how many entries you want returned, and these will be returned as a sequence of references. You can request the `extendedEvaluate()` operation to return an initial set of entries from the `Iterator`. This is equivalent to returning the iterator, and then requesting `nextS` to get that same initial set.

Queries that result in a data array

There are times when you want a query to result in an array of data values instead of a set of objects that would normally encapsulate that data. This could be the case, for example, when you want to present the resulting data in a scrolling list on the end user interface.

The following statement returns an array of data values:

```
select empid, name from empHome e where e.deptPtr->deptno=11;
```

In the preceding example, the returned array of data values contains the employee number (*empid*) and the name for each employee contained in the empHome collection that is assigned to department 11. To return a data array you must use the `extendedEvaluateToDataArray()` or the `evaluate_to_data_array()` operation.

You can also use the data array operation to return object references or a combination of object references and data values as in the query

```
select e.OID, empid, name from empHome e where e.deptPtr->deptno=11;
```

The data array operation returns `DataArrayIterator` object, and a sequence of zero or more initial entries from the iterator. The `DataArrayIterator` is an object that represents the data array collection and can return one or more data array rows, that is, entries in the data array collection. If you use the `next()` operation on the `DataArrayIterator`, it returns the next `DataArray` row (a sequence of any types) in the collection. If you use the `nextS()` operation, you can specify how many entries you want returned, and these are returned as a sequence of *N* rows.

Using the data array operation, you can request that it return an initial set of entries from the `DataArrayIterator`. This is equivalent to returning the iterator, and then requesting `nextS` to get that same initial set.

Queries over unnamed collections

To run a query over a collection that is not registered with the Naming Service, use a parameter list.

Parameter Lists

A Name/Value pair list that consists of strings and references to collection objects (homes, views or reference collection). In the `from` clause of the query you use the name from the Name/Value pair, and on the `evaluate` call, you pass the Name/Value pair list.

Coding an `extendedEvaluate()` method call

For simple queries you can use the `evaluate()` method on collections. However if you want to accomplish the following:

- joins,
- projections with multiple elements or with elements other than object references,
- complex queries with aggregation and ordering

then you need to use the `extendedEvaluate()` or `extendedEvaluateToDataArray()` methods. The extended forms of `evaluate` also allow to you return an initial set of result elements which can help performance of your application.

Alternatively you can also obtain a reference to the query evaluator system object and invoke `evaluate_to_iterator()` or `evaluate_to_data_array()` methods.

Next let's look at what a data array is and how to use it in a program.

Data arrays

A data array is a CORBA Sequence of CORBA::Any data types. Each Any contains a typecode and either a primitive value, a pointer to a CORBA struct or a reference to a CORBA Object.

The following query contains three elements in the data array. The first element is *empid* and is type long. The second element is *name* and is type string, and the last element is an object reference to an employee object.

```
select e.empid, e.name, e.OID from empHome e;
```

Sometimes the data type of the attribute returned in the data array is different from the data type defined in the IDL definition of the interface.

Table 15. Comparisons between CORBA attribute data types and data types returned

CORBA Attribute Datatype in IDL	Datatype Returned in Data Array
long (signed or unsigned)	long
short (signed or unsigned)	short or long
double	double
string	string
object reference	object reference
float	float or double
octet	long
enum	long
boolean	short
char	string of length 1
IManagedCollections::Iterator	IManagedCollections::Iterator

Remember that nonprimitive attributes of types CANNOT be used in query statements.

- Union
- Sequence (except for sequence of octet)
- Array
- Any

A `DataArrayList` is used to return the initial set of result elements and also to return multiple elements on the `nextS()` or `nextN()` method of a data array

iterator. A `DataArrayList` is a sequence of `DataArrays`. You may think of it as a two-dimensional array of `CORBA::Any` data types.

DataArrays and SQL NULL values

How are SQL NULL values handled in a CORBA programming environment where the programming language primitive types do not support the concept of a NULL value?

In the case of an object reference query, query is returning references to objects and it is up to the object to decide how to handle null values. If your application calls the `name()` method on the employee object, and the value of `name` is NULL in the database, the `name()` method can either return an empty string or throw an exception. The point is that what `name()` does is determined by the object implementation. In the case of a data array query returning a primitive value type, if the value is NULL in the database, then the typecode of the `Any` in the `DataArray` will be set to `tk_null`.

Arguments of extendedEvaluate methods

The input parameters to the `extendedEvaluate` operations are:

1. The query statement itself as a string. Remember that the syntax for OOSQL requires the query statement to terminate in a semicolon character.
2. The parameter list consisting of Name/Value pairs for collections. In C++ the parameter list is optional. In Java you must pass at least a zero length parameter list.
3. The third parameter is a placeholder for future use.
4. The fourth parameter is an unsigned long which is the number of initial result elements to return.

Suppose that the result collection has 20 elements and the `evaluate()` method requested 10 elements. The first 10 elements would be returned from the `evaluate` call and the remaining 10 would be retrieved using the iterator. If the `evaluate()` method requests 30 for the initial result, since the result set only has 20 elements, the 20 elements would be returned in the `evaluate()` method call. An iterator would still be returned but that iterator would be positioned at the end of the result set.

The output parameters are:

1. The initial set of result elements.
2. An iterator to the remaining elements.

The exact types of the output parameters differ between `extendedEvaluate()` and `extendedEvaluateToArray()`. The example below shows the details.

DataArray iterator

The DataArrayIterator interface has operations which can be used to determine the number of columns in the data array in addition to the type, attribute name and class name for each column. These methods names are

unsigned long	get_number_of_fields()
string	get_field_name(in unsigned long position)
CORBA::TCKind	get_field_type(in unsigned long position)
string	get_field_class_name(in unsigned long position)

A performance tip

Do not forget to make use of the nextS() interface on the iterator to retrieve the multiple elements in one call. By using a blocking factor you can reduce the number of trips across the network and significantly improve your application's performance.

Handling exceptions

The query evaluator methods can throw three different exceptions when things go wrong. The exception types are IExQueryInvalid, IExQueryProcessingError, IExQueryTypeInvalid. All of these types contain an error number (errorNo) and message text (why). The substitution tokens that went into making up the message text are also available in the *argList* attribute.

Additional details on the cause of the error are found in the activity log of the application server.

C++ example

Following is a complete example of a C++ client program. If you are coding a method in that is part of C++ BusinessObject implementation , you can use a similar coding technique.

The program has five parts.

- Initialization
- Perform the query
- Process the query result
- Catch exceptions
- Cleanup

Initialization consists of getting a reference to the collection and starting a transaction. The query result has two parts, processing the initial set of result

elements returned from the evaluate() method, and processing the remainder of the result set from the iterator object.

Cleanup is very important to insure you do not have memory leaks on the server or the client. The remove() operation must be called for iterators, member lists must be deleted and other object references must be correctly released.

```
// include the proper header files in your C++ application
//
#include <IManagedAdvancedClient.hh>
#include <CBSeriesGlobal.hh>
#include <IExtendedLifeCycle.hh>
#include <CosTransactions.hh>
#include <IExtendedQuery.hh>
#include <IQueryLocalObjectImpl.hh>
#include <IQueryManagedClient.hh>

#include "empdep.hh"           // this is for your application

main() {

    CORBA::Object_var          obj;
    CosTransactions::Current_ptr currentTransaction;
    ICollectionsBase::IIterator_var iter;
    ICollectionsBase::MemberList* memberList=NULL;
    IManagedClient::IManageable_var tup;

    // step 2. initialization and get a factory finder
    //

    try {

        CBSeriesGlobal::Initialize(); // only needed for a client application
        obj = CBSeriesGlobal::nameService() ->

        resolve_with_string(
            "/host/resources/factory-finders/host-scope-widened");
        IExtendedLifeCycle::FactoryFinder_var myFinder =
            IExtendedLifeCycle::FactoryFinder::_narrow(obj);

        // find the employee home
        //
        obj = myFinder->find_factory_from_string("employee.object interface");
        IManagedAdvancedClient::IQueryableIterableHome_var empHome =
            IManagedAdvancedClient::IQueryableIterableHome::_narrow(obj);

        // begin a transaction
        //
        obj =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
        currentTransaction = CosTransactions::Current::_narrow(obj);
        currentTransaction->set_timeout( 50 );
    }
}
```

```

currentTransaction->begin();

// step 3. invoke extendedEvaluate method within a transaction context
//

iter = empHome->extendedEvaluate(
    "select e.OID from thisCollection e where e.deptno>20 ";
    NULL,          // parameter list not used
    NULL,          // reserved
    5,             // return first 5 elements
    memberList);  // sequence containing first 5 elements

// step 4. process result elements in member list
//
for (int i=0; i<length(); i++) {
    tup=(*memberList)[i];
    employee_var emp=employee::_narrow(tup);
    // . . process object
    cout << "employee object with id=" << emp->id() << endl;
}
delete memberList;

// step 5. process remaining elements from iterator
//

while( (tup=iter->next()) != NULL )
{
    employee_var emp =employee::_narrow(tup);
    // . . process employee object
    cout << "employee object with id=" << emp->id() << endl;
}

// step 6. catch and process any exceptions
//
}
catch (IExtendedQuery::IExQueryInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryProcessingError &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryTypeInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch(...) { }

// cleanup iterator
//
if (iter != NULL) iter->remove();

```

```

        currentTransaction->commit(1);

    return 0;
}

```

As an alternative to the previous code, instead of using the name “thisCollection” in the previous query, we will create and use a parameter list that refers to the employee home collection as “emp”.

```

#include <IManagedAdvancedClient.hh>
#include <CBSeriesGlobal.hh>
#include <IExtendedLifeCycle.hh>
#include <CosTransactions.hh>
#include <IExtendedQuery.hh>
#include <IQueryLocalObjectImpl.hh>
#include <IQueryManagedClient.hh>

#include "empdep.hh"           // this is for your application

main() {

    CORBA::Object_var          obj;
    CosTransactions::Current_ptr currentTransaction;
    ICollectionsBase::IIterator_var iter;
    ICollectionsBase::MemberList* memberList=NULL;
    IManagedClient::IManageable_var tup;

    // create a parameter list builder helper object
    //
    CosQuery::ParameterList      * pList;
    IExtendedQuery::ParameterListBuilder * pb ;

    // step 2. initialization and get a factory finder
    //

    try {

        CBSeriesGlobal::Initialize(); // only needed for a client application
        obj = CBSeriesGlobal::nameService() ->

        resolve_with_string(
            "/host/resources/factory-finders/host-scope-widened");
        IExtendedLifeCycle::FactoryFinder_var myFinder =
            IExtendedLifeCycle::FactoryFinder::_narrow(obj);

        // find the employee home
        //
        obj = myFinder->find_factory_from_string("employee.object interface");
        IManagedAdvancedClient::IQueryableIterableHome_var empHome =
            IManagedAdvancedClient::IQueryableIterableHome::_narrow(obj);

        // begin a transaction
        //

```

```

obj =
CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
currentTransaction = CosTransactions::Current::_narrow(obj);
currentTransaction->set_timeout( 50 );
currentTransaction->begin();

// step 3. invoke extendedEvaluate method using a parameter list
//         to pass reference to collection.
//
pb = IQueryLocalObjectImpl::ParameterListBuilder::_create();
pb->add_object_parm("emp", empHome);
pList=pb->get_parm_list();
iter = empHome->extendedEvaluate(
    "select e.OID from emp e where e.deptno>20;",
    *pList,
    NULL,
    0,
    memberList);

delete pb;
delete pList;

// step 4. process remaining elements from iterator
//
while( (tup=iter->next()) != NULL )
{
    employee_var emp =employee::_narrow(tup);
    // . . process employee object
    cout << "employee object with id=" << emp->id() << endl;
}

// step 5. catch and process any exceptions
//
}
catch (IExtendedQuery::IExQueryInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryProcessingError &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryTypeInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch(...) { }

// cleanup iterator
//
if (iter != NULL) iter->remove();

```

```

        currentTransaction->commit(1);

    return 0;
}

```

The above examples showed how to execute an object reference query. That is a query whose SELECT clause had only one element that was a object reference.

Now let's look at an example of a data array query. A data array query can have multiple elements in the SELECT clause and the elements can be primitive data types, structs, iterators or object references.

The iterator and initial list of result elements are a different type for a Data Array query than for a object reference query. In this example we use *daiter* and *daMemberList*.

```

CORBA::Object_var          obj;
IQueryManagedClient::DataArrayIterator_var daiter;
ICollectionsBase::DataArrayList *          daMemberList;

obj = empHome->extendedEvaluateToArray(
    "select e.empid, e.name, e.OID from thisCollection e "
    "where e.deptno>20";,
    NULL,          // parameter list
    NULL,          // reserved
    5,             // return first 5 elements
    daMemberList); // sequence containing first 5 elements
daiter= IQueryManagedClient::DataArrayIterator::_narrow(obj);

```

The *daMemberList* consists of a sequence of *DataArrays*. Since a *DataArray* is a sequence of *CORBA::Any*, the *memberList* is really a two dimensional array of *CORBA::Any*. Each row in the array consists of (empid, name, employee object reference). To get the values of the *i*th elements from the *memberList* you would code the following:

```

long empid;
CORBA::string      name_temp;
CORBA::string_var name;
IManagedClient::IManageable_ptr mo_temp;
employee_var emp;

// for each element in the memberList perform the following

for (int i=0; i< daMemberList->length(); i++) {
    if((*daMemberList)[i][0] >>= empid){} else empid=0;
    if((*daMemberList)[i][1] >>= name_temp) name=name_temp; else name="";
    if((*daMemberList)[i][2] >>= mo_temp)
emp=employee::_narrow(mo_temp);
        else emp =0;
}

```

If the extract operator fails, it does not throw an exception but returns FALSE and does not change the value of the operand. It is always good practice to test the return code from the extract and take appropriate action. The previous code assigns zero or an zero length string if the extract fails. One reason the extract would fail is if the database table contains SQL NULL values. OOSQL will return a SQL NULL value as an CORBA::Any with a typecode of tk_null.

A couple of important points about memory management in C++. For proper memory management when extracting strings, you should not extract into a *_var* variables because the extract operator will make a copy of the string but will not free any previous string value contained by the *_var*. So the previous code does an extract into a *char ** variable type, and then assigns the string value to a *string _var* variable.

When extracting an object reference from the member list, remember that the extract operator does not increment the reference count. So either extract into a *_ptr* variable or increment the reference count yourself.

Here is a complete example using the data array interface.

```
#include <IManagedAdvancedClient.hh>
#include <CBSeriesGlobal.hh>
#include <IExtendedLifeCycle.hh>
#include <CosTransactions.hh>
#include <IExtendedQuery.hh>
#include <IQueryLocalObjectImpl.hh>
#include <IQueryManagedClient.hh>

#include "empdep.hh"           // this is for your application

main() {

    CORBA::Object_var          obj;
    CosTransactions::Current_ptr    currentTransaction;
    IQueryManagedClient::DataArrayIterator_var    daiter;
    ICollectionsBase::DataArrayList    *    daMemberList;
    IExtendedQuery::DataArray *    tup;

    // step 2. initialization and get a factory finder
    //

    try {

        CBSeriesGlobal::Initialize();    // only needed for a client application
        obj = CBSeriesGlobal::nameService() ->

        resolve_with_string("/host/resources/factory-finders/host-scope-widened");
        IExtendedLifeCycle::FactoryFinder_var myFinder =
            IExtendedLifeCycle::FactoryFinder::_narrow(obj);

        // find the employee home
        //
```

```

obj = myFinder->find_factory_from_string("employee.object interface");
IManagedAdvancedClient::IQueryableIterableHome_var empHome =
    IManagedAdvancedClient::IQueryableIterableHome::_narrow(obj);

// begin a transaction
//
obj =
CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
currentTransaction = CosTransactions::Current::_narrow(obj);
currentTransaction->set_timeout( 50 );
currentTransaction->begin();

// step 3. invoke extendedEvaluateToDataArray method
//

obj = empHome->extendedEvaluateToDataArray(
    "select e.empid, e.name, e.OID from thisCollection e "
    "where e.deptno>20 ";
    NULL,          // parameter list
    NULL,          // reserved
    5,             // return first 5 elements
    daMemberList); // sequence containing first 5 elements
daiter= IQueryManagedClient::DataArrayIterator::_narrow(obj);

// step 4. process result elements in member list
//
for (int i=0; i< daMemberList->length(); i++) {
    long empid;
    CORBA::String name_temp;
    CORBA::String_var name;
    IManagedClient::IManageable_ptr mo_temp;
    employee_var emp;

    if ((*daMemberList)[i][0] >=> empid){} else empid=0;
    if ((*daMemberList)[i][1] >=> name_temp) name=name_temp;
        else name="";
    if ((*daMemberList)[i][2] >=> mo_temp)
emp=employee::_narrow(mo_temp);
        else emp =0;
    // . . process object
    cout << "ID=" << empid << " Name=" << name << endl;
}
delete daMemberList;

// step 5. process remaining elements from iterator
//

while ( daiter->next(tup)) {
    long empid;
    CORBA::String name_temp;
    CORBA::String_var name;
    IManagedClient::IManageable_ptr mo_temp;
    employee_var emp;

    if ((*tup)[0]>=> empid) {} else empid=0 ;

```

```

        if ((*tup)[1]>>= name_temp) name=name_temp; else name="";
        if ((*tup)[2]>>= mo_temp) emp=employee::_narrow(mo_temp); else
emp=0;
        delete tup;
        // . . process employee object
        cout << "employee object with id=" << emp->id() << endl;
    }

// step 6. catch and process any exceptions
//
}
catch (IExtendedQuery::IExQueryInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryProcessingError &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryTypeInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch(...) { }

// cleanup iterator
//
if (daiter != NULL) daiter->remove();

currentTransaction->commit(1);

return 0;
}

```

Using the query evaluator object directly

You can also write a program that uses the query evaluator directly. You must first use the Naming Service to get a reference to a query evaluator object. Each server has one instance of the query evaluator. Your application must either know or have a way of finding out the name of the server. For details, see “Query evaluators” on page 323.

The query evaluator methods take an additional parameter of language type. In Component Broker the only language supported is OOSQL so the language type is ignored.

For correct memory management, be aware that the memberList and the iterator are defined as output parameters and you should not use *_var* variables for output parameters. Therefore memberList and iterator are declared as *_ptr* types.

Here is a program that uses the query evaluator object

```
#include <IManagedAdvancedClient.hh>
#include <CBSeriesGlobal.hh>
#include <IExtendedLifecycle.hh>
#include <CosTransactions.hh>
#include <IExtendedQuery.hh>
#include <IQueryManagedClient.hh>
#include <IQueryLocalObjectImpl.hh>

#include "Policy.hh"

main () {

CORBA::Object_var      obj;
CosTransactions::Current_ptr  currentTransaction;

// step 2 start a transaction and get the query evaluator object

try {

CBSeriesGlobal::Initialize();
obj = CBSeriesGlobal::orb()->resolve_initial_references(
    "CurrentTransaction" );
currentTransaction = CosTransactions::Current::_narrow( obj );
obj = CBSeriesGlobal::nameService() ->resolve_with_string(
    "host/resources/servers/MyServer/query-evaluators/default");
IExtendedQuery::QueryEvaluator_var qe =
    IExtendedQuery::QueryEvaluator::_narrow(obj);

currentTransaction->set_timeout( 40 );
ICollectionsBase::IIterator_ptr queryIt_temp;
ICollectionsBase::IIterator_var queryIt;
IExtendedQuery::MemberList* ml;

    currentTransaction->begin();

// issue the query

qe->evaluate_to_iterator(
    "select e from policyDefaultTransDB2Home e "
    "  where e.amount > 0; ",
    0,    // query language. Only OOSQL is supported.
    0,    // no parameter list used in this example
    0,    // not used
    0,    // no initial list requested in this example
    ml,
    queryIt_temp);

queryIt=queryIt_temp;
```

```

// iterator over the result

IManagedClient::IManageable_var tup;
while( (tup=queryIt->next()) != NULL )
{
    Policy_var p =Policy::_narrow(tup);
    cout << "Policy no= " << p->policyNo() << " amount " <<
        p->amount() &< " premium " << p->premium()<< endl;
}
// normal cleanup
queryIt->remove();

IExtendedQuery::DataArrayList      * daml_temp;
IExtendedQuery::DataArrayIterator  * dait_temp;
IExtendedQuery::DataArrayIterator_var  dait;
IExtendedQuery::DataArray          * datup;

qe->evaluate_to_data_array(
    "select p.policyNo, p.premium, p.amount, p "
    " from policyHome p where p.policyNo in (10,11,12); ",
    0, // query language
    0, // no parameter list used in this example
    0, // not used
    0, // initial list of 3 elements
    daml_temp,
    dait_temp);

dait=dait_temp;

while ( dait->next(datup)) {
    long policyNo;
    double amount, premium;

    if (((*datup)[0]) >= policyNo) {} else policyNo=0;
    if (((*datup)[1]) >= premium) {} else premium=0;
    if (((*datup)[2]) >= amount) {} else amount=0;
    delete datup;
    // . . process policy
}

IQueryManagedClient::DataArrayIterator_var it =
    IQueryManagedClient::DataArrayIterator::_narrow(dait);
it->remove();

}

// exception processing
catch (IExtendedQuery::IExQueryInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
}
catch (IExtendedQuery::IExQueryProcessingError &ex)
{

```

```

        cout << "query error number=" << ex.errorNo << endl
            << "query error message=" << ex.why << endl;
    }
    catch (IExtendedQuery::IExQueryTypeInvalid &ex)
    {
        cout << "query error number=" << ex.errorNo << endl
            << "query error message=" << ex.why << endl;
    }
    catch (...) { cout << "Unknown exception occurred" << endl; }

    currentTransaction->commit(1);
    return 0;
}

```

Java client and Java BO example

Here is the same program but coded in Java in the form of a client application program. If you were coding a Java BO, you would following the same coding technique. Memory management is a lot easier in Java but remember you still have to code a remove on the query iterator.

```

import java.net.*;
import java.io.FileInputStream;
import java.io.InputStream;
import org.omg.CORBA.ORB;
import org.omg.CosTransactions.*;
import com.ibm.IExtendedNaming.*;
import com.ibm.IExtendedLifeCycle.*;
import com.ibm.IExtendedQuery.*;
import com.ibm.IManagedAdvancedClient.*;
import com.ibm.CBCUtil.CBSeriesGlobal;

//          application classes
import empClass;
import empClassKey;
import empClassCopy;
import empClassHelper;

public class SempdepQueryApp
{
    //-----
    // Instance variables for the test case:
    //-----

    static String host = "";
    static String port = "";

    org.omg.CORBA.Object obj;
    org.omg.CORBA.ORB orb;
    org.omg.CORBA.Object orbCurrent;

    // Object reference to transaction object
    org.omg.CosTransactions.Current currentTransaction = null;

    // Object reference to the factory finder object

```

```

FactoryFinder factoryFinder = null;

// Object reference to the queryableIterableHome for employee objects
com.ibm.IManagedAdvancedClient.IQueryableIterableHome empQueryableHome =
null;

// Object reference to a employee object
empClass emp = null;

//=====
// Constructor
//=====
public SempdepQueryApp () { }

//=====
// Main
//=====
public static void main( String args[])
{

    // Process Arguments -----
    String usage = "Usage: java -noclassgc SempdepQueryApp
<hostname.domain.company.com> <port> \n \t where port is optional
(default: 900)";

    switch (args.length) {
        case 0:
            System.out.println(usage); System.exit(1); break;
        case 1:
            host = args[0];
            port = "900";

            break;
        case 2:
            host = args[0];
            port = args[1];
            break;
        default:
            System.out.println(usage);
            System.exit(1);
    }

    System.out.println("Begin empdep extended query apis application ...");
    System.out.println("-----");
    SempdepQueryApp empdepta = new SempdepQueryApp();
    empdepta.start(args);
    System.out.println("-----");
    System.out.println("SUCCESS: empdep extended query apis application
ending ...");
}

void start(String args[])
{
    System.out.println("1) About to call CBSeriesGlobal.Initialize passing
host and port");
}

```

```

        System.out.println("\t ORBInitialHost = " + host);
        System.out.println("\t ORBInitialPort = " + port);

        testmain7();
        testmain8();
        testmain9();
    } // End of start();

//-----
// testmain7 -- example of extendedEvaluate
//-----

    void testmain7()
    {
        CBSeriesGlobal.Initialize(host, port);
        orb = CBSeriesGlobal.orb();
        try
        {
            orbCurrent = orb.resolve_initial_references("CurrentTransaction");
        }
        catch (org.omg.CORBA.ORBPackage.InvalidName e)
        {
            System.out.println("The name passed in the argument is invalid.
Exception was: " + e + "\n");
        }
        //-----
        // Next, resolve to the factory finder.
        //-----
        try
        {
            obj =
CBSeriesGlobal.nameService().resolve_with_string
                ("/host/resources/factory-finders/host-scope-widened");
            factoryFinder = FactoryFinderHelper.narrow(obj);
        }
        catch (Exception e)
        {
            System.out.println("testmain7>ERROR: resolve_to_factory_finder()
failed, Exception: " + e);
            System.exit(1);
        }
        //-----
        // Use this to resolve straight to the homes via find_factory().
        //-----
        try
        {
            obj = factoryFinder.find_factory_from_string("empClass.object
interface");
            empQueryableHome =
com.ibm.IManagedAdvancedClient.IQueryableIterableHomeHelper.narrow(obj);
        }
        catch (Exception e)
        {
            System.out.println("testmain7>ERROR:
resolve_to_emp_home_via_find_factory() failed, Exception: " + e);
        }
    }

```

```

        System.exit(1);
    }
    boolean commit=true;
    try
    {
        currentTransaction =
org.omg.CosTransactions.CurrentHelper.narrow(orbCurrent);
        currentTransaction.begin();
        com.ibm.ICollectionsBase.IIterator queryIt;
        com.ibm.ICollectionsBase.MemberListHolder aggregate_members =
            new com.ibm.ICollectionsBase.MemberListHolder();
        org.omg.CosQueryCollection.NVPair[] collection_names=
            new org.omg.CosQueryCollection.NVPair[0];
        org.omg.CosQueryCollection.NVPair[] params=
            new org.omg.CosQueryCollection.NVPair[0];
        int how_many=3;
        String buff = "select e.OID from thisCollection e where e.deptno>1;";
        queryIt = empQueryableHome.extendedEvaluate( buff,
                                                    collection_names,
                                                    params,
                                                    how_many,
                                                    aggregate_members);

        int i ;
        for (i=0; i < aggregate_members.value.length; i++)
        {
            emp = empClassHelper.narrow(aggregate_members.value[i]);
            System.out.println("  employee object with id = " + emp.no() );
        }
        com.ibm.IManagedClient.IManageable tup ;
        while ( (tup= queryIt.next()) !=
(com.ibm.IManagedClient.IManageable)null )
        {
            emp = empClassHelper.narrow(tup);
            System.out.println("  employee object with id = " + emp.no() );
        }
        queryIt.remove();
        currentTransaction.commit(commit);
    }
    catch (com.ibm.ICollectionsBase.IQueryInvalid e)
    {
        System.out.println("\t testmain7>query error number= " + e.errorNo +
" error msg=" + e.why);
    }
    catch (com.ibm.ICollectionsBase.IQueryProcessingError e)
    {
        System.out.println("\t testmain7>query error number= " + e.errorNo +
" error msg=" + e.why);
    }
    catch (com.ibm.ICollectionsBase.IQueryTypeInvalid e)
    {
        System.out.println("\t testmain7>query error number= " + e.errorNo +
" error msg=" + e.why);
    }
    catch (Exception e)
    {

```

```

        System.out.println("testmain7>ERROR: run_query failed, Exception: "
+ e);
    }

    } //end of testmain7

//-----
//---- testmain8 example of extendedEvaluate with a parameter list
//-----
void testmain8()
{
    boolean commit=true;
    try
    {
        currentTransaction.begin();
        com.ibm.ICollectionsBase.IIterator queryIt ;
        com.ibm.ICollectionsBase.MemberListHolder aggregate_members =
            new com.ibm.ICollectionsBase.MemberListHolder();
        org.omg.CosQueryCollection.NVPair[] collection_names=
            new org.omg.CosQueryCollection.NVPair[0];
        com.ibm.IExtendedQuery.ParameterListBuilder pb =
            com.ibm.IQueryLocalObjectImpl.
                ParameterListBuilderHelper._create();

        pb.add_object_parm("empClassMOHome", empQueryableHome);
        org.omg.CosQueryCollection.NVPair[] collection_names =
pb.get_parm_list();
        int how_many=3;
        String buff =
            "select e.OID from thisCollection e where e.deptno>1;";
        queryIt = empQueryableHome.extendedEvaluate( buff,
                                                    collection_names,
                                                    params,
                                                    how_many,
                                                    aggregate_members);

        int i = 0;
        for (i=0; i < aggregate_members.value.length; i++)
        {
            emp = empClassHelper.narrow(aggregate_members.value[i]);
            System.out.println("  employee object with id = " + emp.no() );
        }
        com.ibm.IManagedClient.IManageable tup ;
        while ( (tup= queryIt.next()) !=
(com.ibm.IManagedClient.IManageable)null )
        {
            emp = empClassHelper.narrow(tup.value);
            System.out.println("  employee object with id = " + emp.no() );
        }
        queryIt.remove();
        currentTransaction.commit(commit);
    }
    catch (com.ibm.ICollectionsBase.IQueryInvalid e)
    {

```

```

        System.out.println("\t testmain8>query error number= " +
e.errorNo + " error msg=" + e.why);
    }
    catch (com.ibm.ICollectionsBase.IQueryProcessingError e)
    {
        System.out.println("\t testmain8>query error number= " +
e.errorNo + " error msg=" + e.why);
    }
    catch (com.ibm.ICollectionsBase.IQueryTypeInvalid e)
    {
        System.out.println("\t testmain8>query error number= " + e.errorNo +
" error msg=" + e.why);
    }
    catch (Exception e)
    {
        System.out.println("testmain8>ERROR: run_query failed, Exception: "
+ e);
        // System.exit(1);
    }
} //end of testmain8

//-----
//---- testmain9 example of extendedEvaluateToArray
//-----

void testmain9()
{
    boolean commit=true;
    try
    {
        currentTransaction.begin();
        org.omg.CORBA.Object obj ;
        com.ibm.ICollectionsBase.DataArrayListHolder aggregate_members =
            new com.ibm.ICollectionsBase.DataArrayListHolder();
        org.omg.CosQueryCollection.NVPair[] collection_names=
            new org.omg.CosQueryCollection.NVPair[0];
        org.omg.CosQueryCollection.NVPair[] params=
            new org.omg.CosQueryCollection.NVPair[0];
        int how_many=3;
        String buff =
            "select e.no,e.name,e.OID from thisCollection e where
e.deptno>1;";

        obj = empQueryableHome.extendedEvaluateToArray( buff,
            collection_names,
            params,
            how_many,
            aggregate_members);

        com.ibm.IQueryManagedClient.DataArrayIterator queryIt;
        queryIt =
com.ibm.IQueryManagedClient.DataArrayIteratorHelper.narrow(obj);

        int i = 0;
        long idx;

```



```

String namex;
com.ibm.IManagedClient.IManageableHolder moObjx;
for (i=0; i < aggregate_members.value.length; i++)
{
    idx = aggregate_members.value[i][0].extract_long();
    namex = aggregate_members.value[i][1].extract_string();
    emp = empClassHelper.extract(aggregate_members.value[i][2]);
    System.out.println("    empid = " + idx + "\n" );
    System.out.println("    empname = " + namex + "\n" );
    System.out.println(
        "        employee object with id = " + emp.no() );
}

com.ibm.IExtendedQuery.DataArrayHolder tup =
    new com.ibm.IExtendedQuery.DataArrayHolder();
while ( queryIt.next(tup) )
{
    idx = tup.value[0].extract_long();
    namex = tup.value[1].extract_string();
    emp = empClassHelper.extract(tup.value[2]);
    System.out.println("    empid = " + idx + "\n" );
    System.out.println("    empname = " + namex + "\n" );
    System.out.println("    employee object with id = " + emp.no() );
}
queryIt.remove();
currentTransaction.commit(commit);
}
catch (com.ibm.ICollectionsBase.IQueryInvalid e)
{
    System.out.println("\t testmain9>query error number= " +
e.errorNo + " error msg=" + e.why);
}
catch (com.ibm.ICollectionsBase.IQueryProcessingError e)
{
    System.out.println("\t testmain9>query error number= " +
e.errorNo + " error msg=" + e.why);
}
catch (com.ibm.ICollectionsBase.IQueryTypeInvalid e)
{
    System.out.println("\t testmain9>query error number= " +
e.errorNo + " error msg=" + e.why);
}
catch (Exception e)
{
    System.out.println(
"testmain9>ERROR: run_query failed, Exception: " + e);
}

} //end of testmain9
} // class SempdeptApp

```

Memory management

Running a query builds a result collection in the application server. An iterator over this result collection is returned to the client. The memory used by this result collection is released only when the `remove()` method is run on the iterator. Remember to run the `remove()` method on an iterator in both your mainline and your exception code paths. If you do not do this, your server will run short of memory after you run many queries.

Usage of Naming Service by query

Collection names appearing in the FROM clause of queries are resolved using the Naming Service. To save the cost of repeated calls to the Naming Service, the Query Service internally caches collection names. If the binding of a name in the Naming Service changes, the change will not be seen by the Query Service until the Component Broker server is restarted.

A parameter list can be passed as an argument to a query evaluator to be used in conjunction with a query. When resolving collection names appearing in the FROM clause of a query, the system first looks in the parameter list before going to the name space. This can result in a significant performance optimization and it is most commonly used for collections that are not identified in the name space. Collection names taken from the parameter list are considered volatile; in contrast with names taken from the name space that are considered stable. The system still caches metadata associated with volatile names, but performs a minimal verification upon the usage of a name in every query to determine if the collection references objects of the same type. If so, the metadata is kept in the internal catalog, otherwise, the metadata is discarded and reacquired. The verification is performed if a name space collection name appears as a parameter list name in another query. Similarly, the verification is also performed if a parameter list name appears as a name space collection name in another query.

Limit on number of query iterators per transaction

The Query Service cannot have more than 64 open SQL cursors per transaction. SQL cursors are also used by the Cache Service. Therefore you may be limited to 64 or less active query iterators in a single transactions.

Limit on query statement size

Query Service uses an internal 8K buffer to parse the query statement as well as the internally generated query execution plan. If the query statement, the mapping metadata or the query executive plan cannot be contained in this buffer then an exception is thrown.

Query Service tips

The following tips will help you in using the Query Service. Some tips cite specific locations where further information can be found.

- Conditions required for queries
- Deferred updates and query statement processing
- DB2 LOBs and DB2 data types are not supported
- Reset the timeout to be greater than the default or to zero
- A query over persistent objects must be implemented within the scope of a transaction
- Use parameter lists instead of named collections
- Support for object relationships is limited to 1 to 1 and 1 to many relationships
- Collections must reside in the same server for navigation between collections
- Use the foreign key pattern in Object Builder for better performance

Conditions required for queries You can query Home Collections, Views and Reference Collections provided you meet the following conditions:

1. The interface of objects to be stored in collections is defined as “queryable” in Object Builder.
2. The managed object should be configured using Object Builder so that the home name is specified as `BOIMRegOfQIHomes`.

If the home is a specialized home, then make sure that the `BusinessObject` inherits from the interface

```
IManagedAdvancedServer::ISpecializedQueryableIterableHome
```

and in addition the `ManagedObject` inherits from the interface

```
IManagedAdvancedServer::  
    ISpecializedQueryableIterableHomeManagedObject.
```

3. The elements of the Reference Collection are defined as queryable. If the object interface is not queryable, you can still create Reference Collections containing these objects but you cannot query the collection.
4. The Reference Collection should be created with the `createCollectionFor()` operation or specify the interface name in the query statement `FROM` clause.

DB2 tables have the capability of using indexes and DB2 has a search engine. Having DB2 do the search is preferable to doing searches over large reference collections where there is no indexing capability.

Design your application to make use of home collections. Avoid the use of query over reference collections when query performance is important.

Deferred updates and query statement processing When there are deferred updates, the query may not take the deferred updates into account when processing the query statement. For example:

```
aMO->name("NewName");
it = aHome->evaluate(" name='NewName'");
// the object aMO will likely not be returned in the iterator result set
// because the search "name='NewName'" was performed against
// values in the database.
```

To make sure that the query search sees the current values, issue a commit() if possible before doing the query.

```
aMO->name("NewName");
currentTransaction->commit(1 );
it = aHome->evaluate(" name='NewName'");
// the object aMO will now be returned by the query search.
```

DB2 LOBs and DB2 data types are not supported. Extended data types such as DB2 LOBs and user-defined DB2 data types are not supported.

Reset the timeout to be greater than the default or to zero. The default ORB request timeout value of 180 seconds may be insufficient when executing queries from a client. Reset the timeout to either a higher value or to zero to wait indefinitely.

A query over persistent objects must be implemented within the scope of a transaction. The iterator returned from query must be used to retrieve the result set before ending the transaction. The iterator becomes invalid at end of the transaction. Refer to the following example.

```
currentTransaction->begin();
IManagedIterator_var it = myHome->evaluate (" amount > 10");
currentTransaction->commit();
currentTransaction->begin();
while (aMO= it->next())
{ . . . // do something with aMO }
// unpredictable behavior because you are using
// an iterator outside the transaction scope in which
// it was created.
```

This may affect the design of ManagedObjects whose methods return query iterators if the ManagedObject is configured into an atomic container. The iterator will become invalid because of the implicit commit done by the

atomic container at request termination. See the “More on Iterators” section in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

Use parameter lists instead of named collections. The use of parameter lists to pass object references to collections to query may perform better than having the Query Service retrieve the reference from the Naming Service. See “Coding an extendedEvaluate() method call” on page 329 for an example of how to use a parameter list.

Support for object relationships is limited to 1 to 1 and 1 to many relationships. Relationships with outer join are not supported.

Collections must reside in the same server for navigation between collections A navigation is specified by path expressions. Path expressions allow traversal. The path expression currently supports navigation between collections that reside in the same server. However, the collections can be mapped to different databases.

Use the foreign key pattern in Object Builder for better performance of object relationships. When defining object relationships and object attributes that are object references, use the foreign key pattern in Object Builder. Queries expressions that use the object references can be pushed down to the datastore resulting in better query performance.

```
interface person {
    attribute read only long id;
    attribute string name;
}

interface account {
    attribute read only long acct_id;
    attribute person acct_owner;
}
```

If acct_owner is stored as a stringified object reference or handle, then the query

```
select a from accountHome a where a.acct_owner..name like 'Bob%';
```

will not be pushed down to the datastore resulting in the more data being read from the database and the query performed in object space. If acct_owner is stored as a relational foreign key in the account table then this query can be pushed down and performed as a relational join between the account and person tables. Only the qualifying rows will be read by the Component Broker application server.

Chapter 12. Query Service for OS/390 and Solaris

▶ 390

▶ SOLARIS

The following chapter applies to OS/390 Component Broker and Sun Solaris.

The Query Service enables you to query for a set of objects that satisfy a set of conditions that you specify. Performing a query using the Query Service is conceptually very similar to performing query on a relational database. It differs in that the Query Service query is performed on a collection of objects rather than a collection of records, and the predicate is formed on the set of attributes and method return values for the object rather than on columns in the tables.

You will often find that you are dealing with very large collections in your business application. The collections that you deal with may have hundreds of thousands, millions, or even billions of object instances. Iterating through the entire collection of objects, looking for the one or few that you want to work with, can be enormously expensive, in terms of system resources. You can use the Query Service to preselect the set of objects that you want to work with. The Query Service will produce a subset of the original collection that satisfies the conditions that you set. If you only want to work with insurance policies that have coverage of more than a million dollars, you can form a query to return only those Policy objects that satisfy that condition, and then iterate on just those few.

The Query Service introduces a new query language: object-oriented SQL (OOSQL). OOSQL is an extension to the SQL language with additional constructs for operating on objects instead of tuples. OOSQL is described in detail in the Component Broker reference section.

Queries are actually performed by a query evaluator. The query evaluator understands the OOSQL grammar and how to apply it to collections of objects to form the requested result. In some cases, the query evaluator is able to push queries all the way down into the underlying datastore for a collection of objects. In this case, the query can be performed in the datastore and can yield significant performance improvements.

There are constraints on where a query evaluator resides relative to the collections that it is evaluating. The query evaluator you use to initiate a query must reside in the same server as the collections on which the query will be performed. If your program is running in a client process, or if your business object is in one server process and you are querying collections that reside in another server process, you will have to obtain the query evaluator that resides in that process. To accomplish this you must be able to find the

name of the server on which those collections reside. This process is described in Get the Server Name of a Query Evaluator.

Certain kinds of collections, termed queryable collections, are able to support query operations directly. If you are performing simple queries on only one collection, you can issue the query request directly on the collection. The collection, in turn, will locate an appropriate query evaluator and use that to form the results that it hands back to you.

Depending on the needs of your application, queries have two different kinds of results. If you perform a query on a single collection, or multiple collections of the same type of object, then you probably want to return a result set containing that same type of object. For instance, if you are performing a query on a collection of Policy objects, then you probably want a result set of Policy objects.

However, if you are performing a query over multiple collections of different types, or even if you are performing a query on a single collection but only want to return a subset of the state produced by those objects, also referred to as a projection, then you probably want to return an array of data records.

The Query Service locates collections referenced in the FROM clause of any query statement by looking up the referenced collection name in the system name space. Thus, collections must be named in the system name space before they can be used in a query statement.

The data returned is a collection of references to objects of one type. The objects in the returned collection can be the exact same objects that were queried in one of the input collections, or a subset of these objects. In this case, no new schema results when the data is returned. Alternatively, to support the notions of projection, join, union, and function, the objects in the returned collection can be of a new dynamic and transient type.

Object-Oriented Structured Query Language

The language for the Component Broker Query Service is object-oriented Structured Query Language (OOSQL). OOSQL is a Query Service over objects where the syntax of the query is expressed in standard or extended forms of the Structured Query Language (SQL). A small number of extensions of the SQL SELECT syntax are available to exploit the object model. The primary extension is that of path expression, that generalizes the notion of navigation through attributes and methods of a Component Broker CORBA Interface in IDL that are CORBA structures (for example, the struct construct in CORBA IDL), references to objects or collections of objects. In accordance with the object model, the database is perceived as a set of collections of objects, and

relationships are represented by collection and reference attributes of the IDL. Results are retrieved by specifying a result collection that can be derived from one or more collections. The result returned is a collection of references to objects of some type. The objects in the returned collection can be the exact same objects that were queried in one of the collections, or a subset of these objects. Alternatively, to support projection, join, union, and function, the objects in the returned collection can be the instances of a new type.

Differences between OOSQL and SQL

This section contains a brief review of SQL showing how OOSQL differs from SQL. For more information on SQL see the SQL Reference Manual and Application Programming Guide in the DB2 product information. To trace and display information about the SQL queries that are executed as a result of OOSQL queries from an application server, see the “Trace SQL Queries” section in the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*.

SQL is a Structured Query Language designed for use with relational databases. Use the following employee and department tables, you can perform queries to find specific data.

Table 16. Employee

empid	name	deptno
12	'Dave'	42
14	'Andrew'	42
16	'Liz'	44
18	'Amy'	44
20	'Don'	44

Table 17. Department

deptno	name	mgrid
42	'Sales'	16
44	'Dev'	20

You can find all employees in department 42:

```
select empid,name from employee where deptno=42
```

You can find all employees whose manager has the ID of 16. This query, however, requires a join of both tables.

```
select e.empid, e.name from employee e, dept d
where d.deptno=e.deptno and d.mgrid=16
```

You can find all employees that are not managers:

```
select empid, name from employee where empid not in
(select mgrid from dept)
```

You can find the number of employees in each department:

```
select deptno, count(*) from employee group by deptno
```

OOSQL is an extension of SQL. Instead of tables used in SQL, data takes the form of collections of objects with attributes and methods.

empHome is a home collection of employee objects with the interface:

```
interface employee{
    attribute readonly long empid;
    attribute string name;
    attribute dept deptPtr;
}
```

deptHome is a home collection of dept objects with the interface:

```
interface dept {
    attribute readonly long deptno;
    attribute string name;
    attribute employee mgr;
    IManagedCollection::IIterator emps();
}
```

Unfortunately IDL does not tell you what kind of objects the method emps() returns. Assume that these are employee objects.

OOSQL queries equivalent to the queries above would be:

```
select e.empid,e.name from empHome e where
e.deptPtr..deptno=42;
```

This query returns the values of empid and name for employee objects in department 42. It is called a data array query.

Some important points to remember:

- OOSQL queries always end with a semicolon.
- The FROM clause names the collection. (Later you will see how to query an unnamed collection.) The name of the collection is the name that the home uses in the DCE namespace. Be aware that home collections have two names, a home name and a factory finder name. The FROM clause always uses the home name, never the factory finder name.
- Correlation identifiers (the “e” in the query above) are always required in OOSQL. In SQL correlation ids are not always required.
- The 2 periods (..) is a dereference operator in OOSQL. This is the same idea as the -> operator in C++. The dereference operator can be used with data types that are object references.

If you want to return object references instead of attribute values the query would be:

```
select ref e from empHome e where e.deptPtr..deptno=42;
```

This is called a reference query.

A correlation name preceded by the keyword REF returns pointers to objects in the collection associated with the correlation name. If the correlation name is not one of the members in the collection associated with the correlation name, the REF correlation name has the same semantics as correlation name. This feature was introduced to preserve compatibility with SQL in which column names can be unqualified. For example, to select pointers to employee objects with the name of 'Bob' from the empHome collection:

```
select ref e from empHome e where name='Bob';
```

If the collection empHome has no member attribute "e", then the above query is the same as the following query:

```
select e from empHome e where name='Bob';
```

You can return attribute values and object references:

```
select e.empid, e.name, ref e from empHome e
where e.deptPtr..deptno=42;
```

This is considered another type of data array query.

To find all the dept objects where the deptno is between 10 and 100 the query would be:

```
select ref x from deptHome x where x.deptno between 10 and 100;
```

The query to find all non-manager employee objects would be:

```
select ref e from empHome e where e.empid not in
(select d.mgr..empid from deptHome d );
```

A count of employees in each department would be performed as follows:

```
select e.deptPtr..deptno, count(*) from empHome e
group by e.deptPtr..deptno;
```

Another similar query is:

```
select d.deptno, count(d.emps) from deptHome d;
```

You can perform string searches using the SQL LIKE operator:

```
select ref e from empHome e where e.name like 'Bob%';
```

"%" is the wild card character in SQL. SQL strings are delimited by single quotes where strings in C++ are delimited by double quotes.

Note: String searches are case sensitive in SQL.

If the dept interface also includes methods, I can also include methods in my queries.

```
interface dept {
    attribute readonly long deptno;
    attribute string name;
    attribute employee mgr;
    double compute_overtime();
    long compute_vacation(in long year);
};
```

Find all dept objects where the overtime is greater than 10 hours:

```
select ref d from deptHome d where d.compute_overtime() > 10;
```

Find the deptno, name and vacation days of dept objects where vacation in 1996 was less than 50 days.

```
select d.deptno, d.dname, d.compute_vacation(1996)
from deptHome d where d.compute_vacation(1996) < 50;
```

The key points of OOSQL queries is that they:

- Are similar to SQL queries.
- Use collection names in the FROM clause (SQL queries use table names).
- Can return object references as well as attribute values (SQL queries can only return column values).
- Have a dereference operator (..) that can be used to follow object references.
- Can do joins, subselects, ordering and summarize data just like SQL.
- Can use object attributes and methods in the select and where clause. Only methods that return a value and have either no parameters or only input parameters can be use in a query statement.

Methods

OOSQL supports invocation of CORBA IDL methods in queries. (Methods are also referred to as member functions.) In an OOSQL query, a method name is followed by the method arguments within parentheses. For example, $m(a_1, a_2, \dots, a_n)$ is a method with name m and arguments a_1, a_2, \dots, a_n . Following the C++ convention, methods with no arguments are followed by empty parentheses: $m()$.

Component Broker implements a dynamic run-time environment for method selection and invocation.

If a method argument is a null value, then the value returned by the method is also null. A programming error in the implementation of the method can cause OOSQL to fail.

Methods must be defined as having zero arguments or input only arguments and must not be defined with a return type of void.

The implementation of methods appearing in queries has limitations. Method arguments in OOSQL statements are checked for type correctness. Where possible, when method execution results in exceptions, a method failure message is generated and the query is terminated. In some cases, a programming error in a user's method might cause the query engine to halt (infinite loop in method) or terminate abnormally.

The following table presents the conversion that is performed when an argument of a given type is passed to a method that is defined with a parameter of the same or different type.

The table shows for example, if you have the idl interface

```
myInterface {
    attribute short s1;
    attribute string s2;
    long methoda(in long input);
}
```

the query statement

```
select e.methoda(e.s1) from myHome e;
```

is valid and the Query Service will convert *s1* from short to long when calling the method. However the query statement

```
select e.methoda(e.s2) from myHome e;
```

is not valid because the string type attribute *s2* cannot be passed to *methoda*.

argument/ parameter type	pointer	short	long	float	double	string	vargraphic	other
pointer	NC	E	E	E	E	E	E	E
short	E	NC	C	C	C	E	E	E
long	E	C*	NC	C	C	E	E	E
float	E	C*	C*	NC	C	E	E	E
ICBCdecimal	E	C*	C*	C*	C*	E	E	E
double	E	C*	C*	C*	NC	E	E	E
string	E	E	E	E	E	NC	C	E
vargraphic	E	E	E	E	E	E	NC	E
other	E	E	E	E	E	E	E	E

argument/ parameter type	pointer	short	long	float	double	string	vargraphic	other
C	Conversion. The argument type is converted to the parameter type.							
C*	Conversion with the possibility of an overflow, an underflow, or a loss of precision due to a type conversion.							
E	Error. The argument type used in a method in a query is not applicable to the parameter type of the method.							
NC	No Conversion. The argument type and the parameter type are the same.							
Note: There is no pass by value capability for method arguments in queries, as indicated by the <i>other</i> type. Arguments that are of complex type must be passed by pointer to be used in methods in queries. Character string pointers are considered string types by query, not as pointer types.								

Inheritance

OOSQL supports interface inheritance as in the following example. Suppose the manager interface inherits from employee.

```
interface manager : employee {
    attribute dept manages_deptPtr;
    attribute long executiveLevel;
}
```

A query statement over manager can select inherited attributes just like noninherited attributes. (No special syntax is required.)

```
select m.no, m.name, m.executiveLevel from managerHome m;
```

Navigation

A navigation is specified by path expressions. Path expressions allow traversal through references, embedded structures, and collections to reach embedded members. The “..” characters are used to express traversal through embedded members. Path expressions can appear anywhere a member can. A path expression is *q.m1..m2..mn* where *q* is a correlation name defined for collection *C*, and *m1* is a member of the element type of *C*, and *m2* is a member of the type of *m1* and so on. A member of *mi* can be a member or a method. A path expression evaluates to the value of the leaf of the expression.

Through embedded structures

Embedded Structures Members can be defined in terms of structures. Navigation allows traversing into the embedded members of structure definitions.

Example:

```

struct addressStruct {
    string street;
    string city;
    string state;
    string country;
    string zip;
}

interface employee {
    attribute long empid;
    attribute addressStruct address;
}

```

You can write the query statement

```

select e.address, e.empid, e.address..city from empHome e
where e.address..zip='95120' order by e.address..city;

```

returns the address struct, employee ID and city for employees in postal code 95120 sorted by city.

Through references

Reference members can also participate in navigational expressions. A reference that has a zero value is treated as a null reference. Only references to objects, structures, and collections can be traversed (that is, appear as other than leaf nodes in path expressions). However, a member of any type can appear as a leaf node of a path expression. Its interpretation is dependent upon its type. If a reference member points to a character string, then OOSQL will dereference the reference to return the character string value. In all other cases (for example, integer, double precision), OOSQL cannot dereference the reference to obtain the value and if such a data item appears in a SELECT statement, the reference to the value is returned. The application can then dereference the reference to retrieve the value. Uninitialized or invalid references can cause OOSQL to terminate abnormally if these are part of a path expression that is traversed.

Through collections

Navigation or traversal through collection members creates implicit correlation names over each embedded collection in the path expression.

Example: The query selects the numbers and names of employees in department one. The set of employees in each department is modeled with the member *emps* defined as a collection. The traversal through *emps* given by *d.emps..empid* defines an implicit correlation *q* over *d.emps*, and the semantics of *empid* is *q.empid*.

```

select d.emps..empid, d.emps..name from deptHome d where d.deptno=1;

```

Collections

Collections are used in place of tables in the OOSQL SELECT syntax. Collections are made up of objects of some type. Objects are used in place of rows and the members of objects are used in place of columns. The term members is used to mean both attributes and methods of a Component Broker CORBA interface. There is no inherent order of objects within a collection. Every object must have one or more members, but the collection can be empty or the number of objects in the collection can be zero. Some types of collections include:

queryable collection

A collection that may appear in the FROM clause of the OOSQL queries. The creation of these collections is system dependent.

result collection

A set of objects that OOSQL selects or generates from one or more queryable collections.

Query optimizations

In general, the semantics of any query requires that the Query Service invoke the methods or attributes specified in the query expression, perform any mathematical operations on the resulting values as specified in the query expression, and determine whether the final result satisfies the specified predicate for each object in the collection. Due to the inherent encapsulation semantics of object-oriented programming, this requires that the Query Service activate each object instance in the collection and perform the evaluation one object at a time.

However, if the collection(s) you are querying is collected by a Home, and that Home is configured to store the state of its objects in a relational database, specifically DB2, then the Query Service can optimize its search. The query evaluator is smart enough to detect that the collection(s) it is operating on may be adapted to a DB2 data store. When this is the case, and when there is a straightforward mapping between the attributes specified in the query, and the underlying columns of the tuple in which the managed objects collected by the Home are stored, then the query evaluator will transform the OOSQL expression into a standard SQL expression and “push down” the query to examine the underlying datastore.

Even if only part of the OOSQL expression can be mapped in this way, the resulting push down will often yield an intermediate result set that is a fraction of the original collection size. Thus, the remaining query can iterate over the intermediate subset and significantly reduce the overhead that would be incurred if the evaluation had to iterate over every object in the collection.

DBMS pushdown rules

The query pushdown determines what parts of the OOSQL query are passed down to the underlying relational database management systems (DBMS) where the data resides. The parts of the OOSQL query that are not pushed down are evaluated in the memory. This section lists which of the OOSQL query constructs are pushed down and which are not for the DB2/390, DB2 Universal Database (UDB) and Oracle database management systems.

Simple expressions: All DBMS types: The expressions containing numeric, date, time, and timestamp functions, arithmetic operators and comparison operators appearing in predicates are pushed down to the DBMS. Comparison operators among {<, ≤, >, ≥} are not pushed down for varchar and vargraphic types.

Joins: All DBMS types: Multiple tables from the same database can be pushed down in a single query if the table are related by an equi-join predicate.

Aggregates, group by clause, having clause: DB2/390, Universal Database (UDB): Aggregates, group by and having can be pushed down provided that all other clauses in the query can be pushed down.

Oracle: Aggregates, group by and having can be pushed down provided that all other clauses in the query can be pushed down and that only columns appear in the projection list.

Distinct: All DBMS types: Pushdown is disabled.

Order by: All DBMS types: Pushdown is disabled.

Union: All DBMS types: Pushdown is disabled.

Subqueries: All DBMS types: If the tables participating in a subquery are from the same database as the tables participating in the outer query and the body of the subquery can be pushed down, the following rules apply: Exists (existential) subqueries are pushed down. Subqueries that are basic predicates, and ANY (existential) and ALL (universal) subqueries are pushed down if the comparison operator is between {=, ≠} or if the comparison operator is among {<, ≤, >, ≥} and the operands of the operator are numeric, date, time, timestamp (that is, arguments of type varchar, vargraphic disable pushdown).

Projection (query without aggregates, group by and having): All DBMS types: Projection clauses in subqueries can be pushed down. The projection clause of the outer query is computed in the memory.

Projection (query with aggregates, group by and having): DB2/390.UBD:
The projection list can be pushed down (see aggregate function list below for details on when pushdown occurs).

Oracle: Pushdown is disabled if anything other than columns appear in the projection list.

Scalar functions:

char

Universal Database (UDB):

The CHAR function can be pushed down for arguments of type varchar, smallint, integer, decimal, date, time, timestamp. Pushdown is not applied for arguments of type 4 byte float, double, duration due to formatting differences.

DB2/390:

Same as Universal Database (UDB) except that varchar arguments cannot be pushed down.

Oracle:

The CHAR function is pushed down using the Oracle to_char function for arguments of type smallint, integer, and decimal. Note that these numeric types map to the Oracle number type with a certain precision and scale.

integer

Universal Database (UDB):

The INTEGER function can be pushed down for varchar and numeric arguments.

DB2/390:

Same as Universal Database (UDB) except that varchar arguments cannot be pushed down.

Oracle:

Pushdown disabled.

digits, float, decimal, year, month, day, hour, minute, second, date, time, timestamp, microsecond

DB2/390, Universal Database (UDB):

Pushdown enabled.

Oracle:

Pushdown disabled.

smallint, double

Universal Database (UDB):

Pushdown enabled.

DB2/390, Oracle:
Pushdown disabled.

Aggregate functions:

count, sum, avg

DB2/390, Universal Database (UDB):
Pushdown enabled.

Oracle:
Pushdown is enabled if aggregates are not in the projection list (e.g., in a having clause or in a subquery).

min, max

All DBMS types:
Pushdown is enabled for numeric, date, time, timestamp arguments, and disabled for varchar, vargraphic arguments.

Miscellaneous rules:

Methods in queries

All DBMS types:
Query terms that are methods are not pushed down.

Queries with reference collections

All DBMS types:
Query terms that are reference collection attributes are not pushed down.

Queries with in-memory object building

All DBMS types:
Query terms that are reference collection attributes are not pushed down.

Path expressions over home collections

All DBMS types:
Path expression over home collections can be pushed down as join expressions.

The cast operator

The cast operator sets the type of a member or method in the body of a query. Thus casting provides type information to OOSQL when it is not available from the schema. However, the casting can be used to override the type information in the schema.

The cast operator follows the member to be altered and specifies the type name between(`%` and `%`).

Casting is a fragile operation, since casting a member to the wrong type may result in producing incorrect answer to the query or can cause OOSQL to fail. See Query over reference collections for an example of casting.

Query over reference collections

The syntax of a query over a reference collection is the same as query over a home collection.

There is one important difference when querying an object whose IDL definition contains an `ICollection` or `Iterator` type as an attribute or return type of a method. The IDL definition does not indicate what kind of object the `ICollection` or `Iterator` references. The Query Service must know this information in order to process the query and so it necessary to indicate this information using a cast function in the query statement itself.

For example, using the `dept` interface from the previous section, the query over a home collection to find all departments that contain an employee whose name starts with 'D' would be:

```
select d from deptHome d where d.emps..name like 'D%';
```

To do this same query over a reference collection (whose name is `deptRC`) of department objects would be:

```
select d from deptRC d where d.emps(%Collection<::employee*>%)  
..name like 'D%';
```

The notation (`%Collection<::interfaceName*>%`) is the OOSQL cast function that specifies the object type of *emps*.

The internal processing for a query over a reference collection is different from a home collection. A query over a reference collection is processed by iterating over the reference collection and activating each object (if it is not already activated) and evaluating the query.

When creating the reference collection, make sure you use the `createCollectionFor()` method and pass in the string equal to the IR name of the objects the collection will contain. For example:

```
createCollectionFor("IDL:policy:1.0");
```

You can find out what this IR name is by loading the IR and then doing an `irdump` with a parameter of your interface name.

If you do not use `createCollectionFor()`, when you query the reference collection, the Query Service won't know what kinds of objects are contained in the collection and cannot execute the query. An exception is thrown.

DB2 tables have the capability of using indexes and DB2 has a search engine. Having DB2 do the search is preferable to doing searches over large reference collections where there is no indexing capability.

Design your application to make use of home collections. Avoid the use of query over reference collections when query performance is important.

Data type mapping between DB2 and CORBA

Many of the DB2 data types have obvious counterparts in CORBA such as DB2 integer mapping to `CORBA::long` and DB2 `char(n)` and `varchar` mapping to `CORBA::string`.

Be aware that `char(n)` and `varchar` will map to null terminated strings in CORBA. Binary string data can be handled by used either `VARCHAR FOR BIT DATA` or `CHAR FOR BIT DATA` in the table definition mapped to `ByteString` datatype in idl.

There are four data types in DB2 that do not have obvious counterparts in CORBA. They are DB2 Date, Time, Timestamp and Decimal. Date, Time and Timestamp should be mapping to `CORBA::String`. There are helper classes (`ICBCDate`, `ICBCTime` and `ICBCTimestamp`) provided in Component Broker if you need to do date and time manipulations of these strings. Decimal can be mapped to `CORBA::double` or `CORBA::String`. Use string when you need exact precision. Double byte strings can be stored using `GRAPHIC` or `VARGRAPHIC` datatype in the table definition and `Wstring` in the idl.

Query evaluators

A query evaluator is the engine behind the Query Service. The query evaluator parses the query expression you supply, locates the inferred collections, and evaluates the collections for the set of objects that satisfy the query predicate. The query evaluator supports the `QueryEvaluator` interface as defined in the standard CORBA services Query Service specification.

The evaluate operation, as specified by OMG in the `CosQuery::QueryEvaluator` interface returns an any. CORBA leaves it to query implementers to specify the structure of this any for the various types of results that can be produced from a query. Depending on the conditions you specify in the query expression you could potentially need to get back either an iterator (a collection of objects whose type matches the objects in the collection you are evaluating), or a data array. However, the Component

Broker implementation of the `CosQuery::QueryEvaluator::evaluate()` operation returns only an `IManagedCollections::Iterator`. You need to test the results returned to ensure they match your expectations.

Component Broker has extended the `CosQuery::QueryEvaluator` interface to introduce variations of the `evaluate` method that return more specific result set types. If you want to be more specific about the type of result set you get, you can use the `evaluate_to_iterator()` operation which passes back an `IManagedCollections::Iterator`, or you can use the `evaluate_to_data_array()` operation which passes back an `IExtendedQuery::DataArrayIterator`.

In general, a query evaluator could support any number of query languages. You would be expected to specify the language that you intend to use, and accept that the query evaluator you choose may or may not be implemented to support that language. The Component Broker query evaluator only supports one language--object-oriented Structured Query Language (OOSQL). OOSQL is a rich language, following in the tradition of SQL, with extensions that are specific to object-oriented programming.

Default query evaluator

Component Broker automatically creates an instance of a query evaluator in every Component Broker server, and binds this in the system name space at the following location:

```
/host/resources/servers/<server-name>/query-evaluators/default
```

Plug your server name in for `<server-name>`. If your program is executing on a server, you can get the name of your local server from the `CBSeriesGlobal::serverName` static member function. This member function can only be invoked within a server process. In a client process, you must know the name of the server containing the query evaluator you want to use.

Obtain a query evaluator

The following procedure demonstrates how to obtain the default query evaluator from the system name space for a well-known server. This procedure can only be completed if you know the name of the server on which the query evaluator exists. The query evaluator you obtain should be on the same server as the collections that you will be querying.

1. Determine which server you want to use.

You need to know the name of the server that contains the query evaluator that you want to use. This could be the server on which you're already executing, or it could be a remote server. In the former case, you can simply get the local server name from the `CBSeriesGlobal::serverName` static member function.

2. Resolve the default query evaluator from the system name space.

Use the Naming Service operations to resolve the default query evaluator in the named server from the system name space.

The following example obtains the default query evaluator from the local server. This example can only be used in a server process—for instance, in a business object implementation.

```
// Declare an intermediate object ref, intermediate naming contexts
// and the targeted query evaluator

CORBA::Object_var intermediateObject;
IExtendedNaming::NamingContext_var serversNC;
IExtendedNaming::NamingContext_var localServerNC;
IExtendedQuery::QueryEvaluator_var defaultQE;

// Resolve to the server's naming context in the Host name space

serversNC = CBSeriesGlobal::nameService()->resolve_with_string(
    "/host/resources/servers");

// Resolve to the local server

localServerNC = serversNC->resolve_with_string(
    CBSeriesGlobal::serverName());

// Resolve and narrow to the default query evaluator

intermediateObject = localServerNC->resolve_with_string(
    "query-evaluators/default");
defaultQE = IExtendedQuery::QueryEvaluator::_narrow(intermediateObject);
```

Get the server name of a query evaluator

The query evaluator you use to initiate a query must reside in the same server as the collections on which the query will be performed. If your program is running in a client process, or if your business object is in one server process and the query that it wants to perform is on collections that reside in another server process, then you will have to obtain the query evaluator that resides in that process. To accomplish this you must be able to find the name of the server on which those collections reside.

Unfortunately, Component Broker does not provide any automated mechanisms for determining the name of a remote server. However, there are a number of techniques you can employ.

First, if the collections you need to query are on the same server as your business object, then you can get the name of the local server from the `CBSeriesGlobal::serverName` static member function. Even if the collections are not in the same server, it is good to note that the `CBSeriesGlobal::serverName` static function will always return the name of its local server.

There are at least five strategies for finding the server name for a query evaluator. These include:

- Ask the User
- Ask an Administrator
- Know your Collections and you'll know their Server
- Create an Anchor
- Specialize your Collections

You may be able to derive your own approach, perhaps as a variation of one of these.

Ask the User

In some cases, the query statement will actually be formed by an end user and provided to you through a user interface. In this case, you may be able to depend on the user knowing the layout of their distributed system topology, and can supply the appropriate server name along with the query statement.

Ask an Administrator

Even if the topology of the system is foreign to your business end users, the system administrator who configured the system should know something about where different resources are located. As part of your application design, you may be able to require that the administrator identify on which servers specific collections exist, and to supply that information to your application through a configuration file. Your application can then read this file during its initial load and use that during its runtime.

If your application uses a number of different collections, this configuration file may need to contain a table of collection to server assignments. By indexing on the collection name, you can look up the corresponding server and use that to find the appropriate query evaluator for the collection you are querying.

Know your Collections and you will know their Server

If your application uses a number of different collections, this configuration file may need to contain a table of collection to server assignments. By indexing on the collection name, you can look up the corresponding server and use that to find the appropriate query evaluator for the collection you are querying.

```
/host/resources/servers/<server-name>/collections
```

where <server-name> is the name of the server on which they exist. Likewise, the corresponding query evaluator that can operate on those collections is named in,

```
/host/resources/servers/<server-name>/query-evaluators/default
```


If you know the fully-qualified name of your collections, then you can deduce from their path the server in which they reside. From that, you can form a path to the corresponding query evaluator.

Note that if the collections reside on a server in a remote host, you may have to use a fully qualified path to navigate to that host. For instance,

```
././hosts/<host-name>/resources/servers/<server-name>/collections
```

or

```
/workgroup/hosts/<host-name>/resources/servers/  
<server-name>/collections
```

depending on whether the host is in your local workgroup or elsewhere in the cell. Notice that in both cases you need to know the name of the target host.

If you only know the host name or that it is on the local host, you can do a brute-force search through all of the servers within the

```
/host/resources/servers
```

naming context. In the worst case, you can do a brute-force search through all of the servers in all of the hosts in the

```
././hosts
```

naming context.

Create an Anchor

For each of the collections that you want to query, you can introduce your own Application Object (an anchor) with instructions to the application assembler to install an instance of each these in the same server as the collections that you're interested in. These anchor application objects can have a simple interface supporting a single operation that returns its own server name. The implementation simply uses the `CBSeriesGlobal::serverName` static function to deduce its own server name and return that from the operation. However, you would want each anchor to have a distinct primary interface name, perhaps all derived from the same base interface containing the single operation.

You could then build or hard code a mapping table that correlates collections with the primary interface name of the anchor that corresponds to that collection. Before initiating a query, you can look up the anchor's primary interface name corresponding to the intended collection, and use that with a factory finder to find the

Home for the anchor. From the anchor itself, invoke the simple operation that returns its server name, and then use that to form a name path to its query evaluator.

A variation of this is to create an individual instance of the same type of anchor in each relevant server, and as part of the creation process, bind these by their collection name in a naming context that you define. You could then look up the anchor by its corresponding collection name and then use the anchor to return its local server name as before. In this case, each anchor would not have to have its own unique primary interface, and you could avoid using the factory finder for locating it.

Specialize your Collections

Many of the queries you perform will be on queryable Homes. Homes are collections and normally are automatically bound in the collections naming context under the server where they belong. However, as Homes, they're also registered in the factory repository and can be found using a factory finder.

If you specialize your Home, you can introduce the same method that we suggested in the "Create an Anchor" strategy that returns its own local server name. In this way, the Home collection can act as its own anchor. You can find the Home using a factory finder, get its local server name, and use that to find a query evaluator that operate on that Home as a collection.

This strategy probably only works well for Home collections.

Topology of query evaluators and collections

The FROM clause of the query statement contains the name of one or more collections. The `evaluate_to_iterator()` and `evaluate_to_data_array()` operations operate only on collections that have been named in the system name space or named in a parameter list. Specifically, collections should be named under the following location in the Host name tree and server, as specified in `<server-name>`, where the collection actually exists:

```
/home/resources/servers/<server-name>/collections
```

The query evaluator looks up the collection specified in the FROM clause either in the parameter list or on the local Host name tree under the query evaluator's own server if the collection is not supplied in the parameter list. If the referenced collection is not supplied in the parameter list or has not been bound in this naming context, then the query cannot be performed. This has the following implications:

- If you do not supply the collection in the parameter list, collections must be uniquely named in the system name space, specifically within the collections naming context for any given server.
- Queries can only operate on collections that exist in the same server as the query evaluator that is operating on them.

Homes are normally automatically bound in the collection’s name context in the server on which the Home exists when the Home is created. You can specify whether a Home is to be bound, and its collection name in the Application DDL for that Home using Object Builder. If you create your own collections, other than Homes, it is up to you to bind that collection in the system name space before you can perform any queries on it.

As previously mentioned, you can supply the collection in the parameter list, provided you give it the same name in the parameter list as you specified in the FROM clause in the query statement, as shown in the following example:

Parameter List	Query Statement
<pre>“myCollection”, IManagedCollections::IReferenceCollection_var myCollection</pre>	<pre>“select p from myCollection p where p.number > 10”</pre>

Form a query

This procedure demonstrates how you can form a query on a collection. This procedure assumes you already know the name of the server containing the collection you will query (myServer in this example).

1. Determine the name of the server where your collections exist. You need to determine the name of the server on which your collections exist. The query evaluator you use must reside in the same server as the collection(s) you are querying. If you are using a local collection, then you can get the local server name using the `CBSeriesGlobal::serverName` static member function. Otherwise, get a remote server name as outlined under “Get the server name of a query evaluator” on page 371.
2. Obtain a corresponding query evaluator. Get the query evaluator that corresponds to the collections you will be querying. Otherwise, you will have to form a name path using the server name that you produced in step 1.
3. Determine the type of result you want to receive. The `evaluate_to_iterator()` operation returns an iterator over a collection of object references. The `evaluate_to_data_array()` operation returns an iterator over a collection of data array rows.
4. Decide if you want any initial values back from the iterator. Both evaluation methods that you may use will return an iterator: either an iterator over a reference collection, or an iterator over a collection of data

array rows. Normally you will iterate over these collections either one or several elements at a time. You have the opportunity at the time you initiate the query to ask that an initial set of elements be returned in a sequence outside of the iterator. This is a convenience mechanism that is equivalent to invoking the query, getting back the iterator, and requesting the first n elements in a separate request.

5. Issue the query request. Depending on the decision you made in step 3, invoke either the `evaluate_to_iterator()` or `evaluate_to_data_array()` operation on the query evaluator. In doing so, pass in the query statement, any accompanying parameter list, and an indication of how many initial elements to return from the resulting iterator.

Queries on queryable collections

Certain collections supplied by Component Broker, specifically Homes, can be queried directly. You can query a Home by narrowing to its `IManagedAdvancedClient::IQueryableIterableHome` interface, and invoking the `evaluate` method. This method has been implemented to locate an appropriate query evaluator, and reissue the request on it. If you use this approach, the queryable collection will form its own `SELECT` and `FROM` clause, and append the predicate that you supply in the query statement argument of the `evaluate` method. If you use this approach, the queryable collection will form its own `SELECT` and `FROM` clause in the form `SELECT REF x FROM thisCollection x WHERE` and append the predicate that you supply in the query statement argument of the `evaluate` method.

Note: The correlation ID will always be the letter `x`.

Queries that result in an object collection

When you specify your query expression, you indicate in the `SELECT` clause the type of the results you expect to get back. The result can be an object type as defined by a managed object in IDL, or some combination of one or more data types.

```
select ref e from empHome e;
```

In the preceding statement, the result is the type of object that is collected by the `empHome` collection, presumably `Employee`. This example returns a collection of objects. This has the benefit of allowing you to perform other operations supported by `Employee` objects on any of the objects returned from these queries. You can direct the query evaluator to return a collection of objects (literally, an iterator to a collection of objects) using the `evaluate_to_iterator()` operation.

The `evaluate_to_iterator()` operation returns an `IManagedCollections::IIterator` object, and a sequence of zero or more initial entries from the iterator. The

Iterator is an object that represents a collection of references to objects, and can return one or more object references, that is, entries in the reference collection. If you use the next operation on the Iterator, it will return the next object reference in the collection. If you use the nextS operation, you can specify how many entries you want returned, and these will be returned as a sequence of references. You can request the evaluate_to_iterator() operation to return an initial set of entries from the Iterator. This is equivalent to returning the iterator, and then requesting nextS to get that same initial set.

Queries that result in a data array

There are times when you want a query to result in an array of data values instead of a set of objects that would normally encapsulate that data. This could be the case, for example, when you want to present the resulting data in a scrolling list on the end user interface.

The following statement returns an array of data values:

```
select empid, name from empHome e where e.deptPtr..deptno=11;
```

In the preceding example, the returned array of data values contains the employee number (empid) and the name for each employee contained in the empHome collection that is assigned to department 11.

In the preceding example, the returned array of data values contains the employee number (empid) and the name for each employee contained in the empHome collection that is assigned to department 11. You can direct the query evaluator to return an array of data values using the evaluate_to_data_array() operation.

```
select ref e from empHome e where e.deptPtr..deptno=11;
```

You can also use this operation with any query statement that would normally return a collection of objects. In this case a reference to each resulting object is stored as a data field in the data array. You can then iterate through the data array, pick up the first field (and presumably the only field unless your SELECT clause specifies other data fields to return as well) of each row in the array, and use that as a reference to the object. At that point, once you narrow to the appropriate interface, you can perform any operation on the referenced object that it supports.

The evaluate_to_data_array() operation returns an IExtendedQuery::DataArrayIterator object, and a sequence of zero or more initial entries from the iterator. The DataArrayIterator is an object that represents the data array collection and can return one or more data array rows, that is, entries in the data array collection. If you use the next_one() operation on the DataArrayIterator, it returns the next DataArray row (a

sequence of any types) in the collection. If you use the nextS() operation, you can specify how many entries you want returned, and these are returned as a sequence of N rows.

Using the evaluate_to_data array() operation, you can request that it return an initial set of entries from the DataArrayIterator. This is equivalent to returning the iterator, and then requesting nextS to get that same initial set.

Queries over unnamed collections

In the queries we have seen so far, the from clause is the name of a collection. This name must be registered with the Naming Service. A home collection usually registers itself with the Naming Service with a name such as:

```
host/resources/servers/MyServer/collections/empHome
```

Only the string empHome is specified in the query statement. Most home collections are registered, but they do not have to be. Reference collections or view collection many times are not registered. To run a query over a collection that is not registered with the Naming Service, use a parameter list. A parameter list is a Name/Value pair list that consists of strings and references to collection objects (homes, views or reference collection). In the from clause of the query you use the name from the Name/Value pair, and on the evaluate_to_data_array or evaluate_to_iterator call, you pass the Name/Value pair list.

An example using the query evaluator interface

This section contains an example that shows:

- Why you would use the query evaluator object.
- How to get an object reference to the query evaluator.
- How to deal with data arrays returned from query evaluator.
- An example using the evaluate_to_iterator method.
- An example using the evaluate_to_data_array method.

The query evaluator is a system object on the application server that is a direct interface to the Query Service. The details of the query evaluator are described in IExtendedQuery.idl. ActiveX, Java and C++ clients all have bindings to the query evaluator. The query evaluator is used when you want to:

1. Run queries over reference collections (reference collections do not have an evaluate method).
2. Run complex queries where you want to join several home collections together.
3. Run a data array query that returns object attribute values. This might be more efficient than first finding object references and then using the references to get the values.

4. Run a data array query that does data summarization.

If RC is a reference collection of employee objects and you want to find employees with names starting with the letter D:

```
select ref e from RC e where e.name like 'D%';
```

You have to use the query evaluator interface because of reason 1 in the previous list. If you want to find the employees whose name is the same as any department name:

```
select ref e from empHome e, deptHome d where e.name = d.name ;
```

You use the query evaluator because of reason 2 in the previous list.

Find the deptname and object reference of all departments with deptid greater than 100:

```
select d.name, d from deptHome d where d.deptno > 100;
```

You use the query evaluator because of reason 3 in the previous list.

Find the number of departments whose dept number is greater than 100:

```
select count(*) from deptHome d where d.deptno > 100;
```

You use a query evaluator because of reason 4 in the previous list.

The query evaluator has these important methods:

evaluate_to_iterator()

This method is similar to evaluate() on the home collection and is used to return object reference queries such as `select ref e from empHome e;`

evaluate_to_data_array()

This interface takes a data array query and the output is an iterator over a collection of data arrays. A data array is a sequence of CORBA::Any. Each any contains an attribute value.

The following query contains 3 elements in the data array. The first element is empid and is of type long second element is name and is type string, and the last element is an object reference to an employee object.

```
select e.empid, e.name, ref e from empHome e;
```

Sometimes the data type of the attribute that comes back in a data array is different from the data type defined in the idl definition of the interface.

Corba Attribute Datatype in IDL	Datatype Returned in Data Array
long (signed or unsigned)	long

Corba Attribute Datatype in IDL	Datatype Returned in Data Array
short (signed or unsigned)	short or long
double	double
string	string
object reference	object reference
float	float or double
octet	long
enum	long
boolean	short
char	string (length 1)

Complex types such as struct, union, sequence, array and CORBA::Any can be used in query statements.

The query evaluator interface is very powerful but slightly more complex because you have to know how to deal with the idl SEQUENCE and the CORBA::Any data types. A data array is a CORBA sequence <any> structure and a memberList is a sequence of data arrays. You can think of a sequence as an array of elements.

To find the length of sequence X:

```
x.length()
```

To get the third element of X:

```
x[2]
```

An Any is a CORBA structure that is a self describing value. The Any stores both a typecode and a value.

To put something into an Any you use the operator <<=:

```
CORBA::Any y;
y <<= "a string value";
y <<= 52;
```

To get something from an Any you use the >>= operator. The >>= operator will first do a typecode check to make sure the receiving variable is of the correct type:

```
CORBA::string_var s;
if (y >>= s)
    cout << "If TRUE, then s contains a copy of
        the string value" << endl;
```



```

else
    cout << "if FALSE, the extract failed because
           the any did not contain a string" << endl;

```

When dealing with data arrays for the first time, be careful. Do not just extract from an any and not check the return code. You extract operator might fail and no exception will be thrown.

Look at the parameters to the method `evaluate_to_iterator` and `evaluate_to_data_array`. The input parameters are:

- The query statement itself as a string. Make sure you include the ending semicolon;
- The second parameter can be coded as NULL. This indicates the query language, but the default Query Service supports only OOSQL.
- The third parameter is a name/value pair list (see the next topic).
- The fourth parameter is NULL for now. This is reserved for future use.
- The fifth parameter is called `HOW_MANY`. It is the number of result elements to return in the `memberList`. This should be a value of zero or a positive number.

There output parameters are:

- `MemberList`. The first `n` result elements are returned as a CORBA sequence. The value of `n` is determined by `HOW_MANY`.
- `Iterator`. The iterator gives access to the remainder of the result collection.

Suppose that the result collection was 20 elements and `HOW_MANY` was set to 10. The first 10 elements would be returned in the `memberList`, and the remaining 10 could be retrieved using the iterator. There are two types of iterators. `IManagedCollection::IIterator` is returned from `evaluate_to_iterator`. This is the same iterator interface as used by Reference Collections. `Evaluate_to_data_array` returns a data array iterator. Its interface is defined in `IExtendedQuery.idl`. It is similar to `IIterator` with the major difference is the `IIterator` returns managed object references and the `Data Array Iterator` returns `Data Arrays` which are CORBA sequences of values from the query result.

Look at the IDL definition of `DataArrayIterator`. At run time the following methods can be used to determine the number of columns in the data array in addition to the type and attribute name for each column:

- `get_number_of_fields`
- `get_field_name`
- `get_field_type`
- `get_field_class_name`

By setting `how_many` to the expected size of the result set, you can reduce the number of trips across the orb to fetch the query result collection.

Do not forget to make use of the nextS() interface on the iterator to retrieve the remainder of the result collection in groups of N. By using n=10 and retrieving 100 objects, you can do it with 10 trips across the ORB instead of 100.

The query evaluator methods can throw 3 different exceptions when things go wrong. The exception types are IExQueryInvalid, IExQueryProcessingError, IExQueryTypeInvalid. All of these types contain an error number (errorNo), message text (why) and extended details (argList).

Additional details on the cause of the error are found in the activity log of the application server. Following is a complete example of using the evaluate_to_iterator method:

```
// step 1 have the following include files in your C++ client
//      program
#include <IManagedAdvancedClient.hh>
#include <IManagedCollections.hh>
#include <CBSeriesGlobal.hh>
#include <IExtendedLifeCycle.hh>
#include <CosTransactions.hh>
#include <IExtendedQuery.hh>
#include <IQueryManagedClient.hh>
#include <IQueryLocalObjectImpl.hh>
#include "Policy.hh"
CORBA::Current_ptr          orbCurrentPtr;
CosTransactions::Current_ptr currentTransaction;

// step 2 start a transaction and get the query evaluator object
CBSeriesGlobal::Initialize();
orbCurrentPtr = CBSeriesGlobal::orb()->get_current();
currentTransaction =
    CosTransactions::Current::_narrow( orbCurrentPtr );

// set transactions time out to 600 seconds.
// Default value of 30 seconds may not be long enough
// when doing some queries.

currentTransaction->set_timeout( 600 );
ICollectionsBase::IIterator_var _queryIt;
try {
    currentTransaction->begin();
    CORBA::Object_var o = CBSeriesGlobal::nameService() ->
        resolve_with_string("host/resources/servers/MyServer
        /query-evaluators/default");
    IExtendedQuery::QueryEvaluator_var _qe =
        IExtendedQuery::QueryEvaluator::_narrow(o);

// step 3 issue the query

    IExtendedQuery::MemberList* x;
    _qe->evaluate_to_iterator(
        "select e from policyDefaultTransDB2Home e
```

```

        where e.amount > 0; ",
    0,
    0,
    0,
    0,    // in this example how_many is set to zero
    x,    // how_many is zero, x will be an empty sequence
    _queryIt);

// step 4 iterator over the result
IManagedClient::IManageable_var tup;
while( (_queryIt->nextOne(tup)) != NULL )
{
    Policy_var p =Policy::_narrow(tup);
    cout << "Policy no= " << p->policyNo() << " amount " <<
        p->amount() << " premium " << p->premium()<< endl;
}
// step 5 normal cleanup
_queryIt->remove();
currentTransaction->commit(0);
}

// step 6 exception processing
catch (IExtendedQuery::IExQueryInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
    currentTransaction->rollback();
}
catch (IExtendedQuery::IExQueryProcessingError &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
    currentTransaction->rollback();
}
catch (IExtendedQuery::IExQueryTypeInvalid &ex)
{
    cout << "query error number=" << ex.errorNo << endl
        << "query error message=" << ex.why << endl;
    currentTransaction->rollback();
}
catch (...)
{
    if (_queryIt!=0) _queryIt->remove();
    currentTransaction->rollback();
}

```

Following is the same query except using a data array query to retrieve attributes amount and premium:

```

// step 1 and 2 same as above
// step 3 becomes
IExtendedQuery::DataArrayList* members;
_ge->evaluate_to_data_array(
"select p.policyNo, p.premium, p.amount, p
    from policyHome p where p.policyNo in(10,11,12); ",

```

```

        NULL,
        NULL,
        NULL,
        3,
        members,
        _queryDataIt);
long policyNo;
double amount, premium;
if ((*members)[0][0] >= policyNo) {}
    else cout << "error extracting policyNo" << endl;
if ((*members)[0][1] >= amount) {}
    else cout << "error exactlying amount" << endl;
if ((*members)[0][2] >= premium) {}
    else cout << "error extracting premium" << endl;
cout << policyNo << amount << premium << endl;
if ((*members)[1][0] >= policyNo) {}
    else cout << "error extracting policyNo" << endl;
if ((*members)[1][1] >= amount) {}
    else cout << "error exactlying amount" << endl;
if ((*members)[1][2] >= premium) {}
    else cout << "error extracting premium" << endl;
cout << policyNo << amount << premium << endl;
if ((*members)[2][0] >= policyNo) {}
    else cout << "error extracting policyNo" << endl;
if ((*members)[2][1] >= amount) {}
    else cout << "error exactlying amount" << endl;
if ((*members)[2][2] >= premium) {}
    else cout << "error extracting premium" << endl;
cout << policyNo << amount << premium << endl;
IQueryManagedClient::DataArrayIterator_var itptr =
IQueryManagedClient::DataArrayIterator::_narrow(_queryDataIt);
itptr->remove();

// exception processing not shown here.

```

Following is an example of a query using a parameter list:

```

IExtendedQuery::ParameterList* collection_names;
IExtendedQuery::ParameterListBuilder *pb =
    IQueryLocalObjectImpl::ParameterListBuilder::_create();
// add a name value pair for "RC"
// rcptr is a pointer to the collection.
pb->add_object_parm("RC", rcptr);
collection_names = pb->get_parm_list() ;
// the following line of code is needed in release 1.0, it is
// not needed in release 1.1 and above
collection_names->length(1); // set the length to 1
_qe->evaluate_to_iterator(
    "select a from RC a where a.policyNo > 3000 ; ",
    0,
    *collection_names,
    0,
    0,
    x,
    _queryIt);

```

Java clients and Java BO example

The following example is a program that uses the query evaluator and is written in Java. The program comments indicate the important points. One important difference is that C++ allows some of the query evaluator input parameters to be zero or null but in Java all parameters must have a value.

```
import java.net.*;
import java.io.FileInputStream;
import java.io.InputStream;

import org.omg.CORBA.ORB;
import com.ibm.IExtendedTransactions.*;
import com.ibm.IExtendedNaming.*;
import com.ibm.IExtendedLifeCycle.*;
import com.ibm.IExtendedQuery.*;
import com.ibm.IManagedCollections.*;
import com.ibm.IManagedClient.*;
import com.ibm.CBCUtil.CBSeriesGlobal;
class QuerySample {
    public static void main (String args[]) {

        org.omg.CORBA.Current orbCurrent;
        org.omg.CosTransactions.Current currentTransaction ;
        org.omg.CORBA.Object obj;

        try {
            // initialize using hostname and default port number and
            // get currentTransaction

            CBSeriesGlobal.Initialize("wisneski.stl.ibm.com","900");
            orbCurrent = CBSeriesGlobal.orb().get_current(
                "CosTransactions::Current");
            currentTransaction =
                org.omg.CosTransactions.CurrentHelper.narrow(orbCurrent);

            // locate query evaluator object for server "testsrv"

            obj=CBSeriesGlobal.nameService().resolve_with_string(
                "host/resources/servers/testsrv/query-evaluators/default");
            com.ibm.IExtendedQuery.QueryEvaluator qe;
            qe = com.ibm.IExtendedQuery.QueryEvaluatorHelper.narrow(obj);

            // start a transaction scope

            currentTransaction.begin();

            // allocate parameters for query evaluator call

            com.ibm.ICollectionsBase.IIteratorHolder it =
                new com.ibm.ICollectionsBase.IIteratorHolder();

            com.ibm.IExtendedQuery.MemberListHolder members =
                new com.ibm.IExtendedQuery.MemberListHolder();
```

```

org.omg.CORBA.InterfaceDef ql_type=null;

org.omg.CosQueryCollection.NVPair[] collection_names;

// build a parameter list and convert to sequence of Name,Value pairs

com.ibm.IExtendedQuery.ParameterListBuilder pb =
com.ibm.IQueryLocalObjectImpl.ParameterListBuilderHelper._create();
org.omg.CosQueryCollection.NVPair[] params =
    new org.omg.CosQueryCollection.NVPair[0];

com.ibm.IManagedClient.IHome aHome = null;

// code goes here to find pointer to a Home Collection object.

pb.add_object_parm("aHome", aHome);
collection_names = pb.get_parm_list();

// invoke the query service returning set of object references
qe.evaluate_to_iterator(
    "select x from mcarMOHome x;", // the query statement
    ql_type,
    collection_names,           // parameter list of collection names
    params,                     // empty parameter list
    3,                          // return first 3 results in members
    members,                    // iterator for remainder of result set
    it);

com.ibm.IManagedClient.IManageableHolder mo = new
    com.ibm.IManagedClient.IManageableHolder();

// retrieve the list of managed objects from the members

    int n;
    int max = members.value.length;
    for (n=0; n<max; n++)

    {
        mo.value = members.value[n];
        System.out.println(" object reference from sequence " + n );
    }
// retrieve the remainder of the query result set
boolean more;
for (more = it.value.nextOne(mo);
    more;
    more = it.value.nextOne(mo))
    {
        System.out.println(" object ref from iterator" + n );
        n++;
    }
// you must remember to cleanup the query iterator
it.remove();
// invoke query service and return a data array
com.ibm.IExtendedQuery.DataArrayIteratorHolder da_it =
    new com.ibm.IExtendedQuery.DataArrayIteratorHolder();

```

```

com.ibm.IExtendedQuery.DataArrayListHolder da_members =
    new com.ibm.IExtendedQuery.DataArrayListHolder();

collection_names = new org.omg.CosQueryCollection.NVPair[0];

qe.evaluate_to_data_array(
    "select x.id, x.model from mcarMOHome x ";
    ql_type,
    collection_names,
    params,
    10,           // first 10 results returned in da_members
    da_members,
    da_it);      // remainder of results set returned in da_it

// retrieve the list of tuples from the da_members sequence

com.ibm.IExtendedQuery.DataArrayHolder da =
    new com.ibm.IExtendedQuery.DataArrayHolder();
// number of columns in the data array
int n_elements = da_it.value.get_number_of_fields();
{
    max = da_members.value.length; // number of rows in
                                   // the da_members array

    for (n=0; n<max; n++)
    {
        // for each row print out the column value
        da.value = da_members.value[n];
        int i;
        for (i=0; i < n_elements; i++)

        {
            // for each column print out the value of the
            // CORBA::Any based on its typecode
            // the following code shows how to retrieve values
            // from a CORBA::Any variable in Java.
            org.omg.CORBA.TCKind tc = da.value[i].type().kind();
            if (tc == org.omg.CORBA.TCKind.tk_long)
                System.out.println(da.value[i].extract_long());
            else if (tc == org.omg.CORBA.TCKind.tk_short)
                System.out.println(da.value[i].extract_short());
            else if (tc == org.omg.CORBA.TCKind.tk_double)
                System.out.println(da.value[i].extract_double());
            else if (tc == org.omg.CORBA.TCKind.tk_string)
                System.out.println(da.value[i].extract_string());
            else if (tc == org.omg.CORBA.TCKind.tk_null)
                System.out.println(" null ");
            else if (tc == org.omg.CORBA.TCKind.tk_objref)
                System.out.println(" obj ref returned ");
            else
                System.out.println(" unknown type returned ");
        }
    }
}

```

```

// retrieve the list of tuples from the iterator
for (more = da_it.value.next_one(da);
     more;
     more = da_it.value.next_one(da))
{
    int i;
    for (i=0; i < n_elements; i++)
    {
        // for each column print out the value of the
        // CORBA::Any based on its typecode
org.omg.CORBA.TCKind tc = da.value[i].type().kind();
        if (tc == org.omg.CORBA.TCKind.tk_long)
            System.out.println(da.value[i].extract_long());
        else if (tc == org.omg.CORBA.TCKind.tk_short)
            System.out.println(da.value[i].extract_short());
        else if (tc == org.omg.CORBA.TCKind.tk_double)
            System.out.println(da.value[i].extract_double());
        else if (tc == org.omg.CORBA.TCKind.tk_string)
            System.out.println(da.value[i].extract_string());
        else if (tc == org.omg.CORBA.TCKind.tk_null)
            System.out.println(" null ");
        else if (tc == org.omg.CORBA.TCKind.tk_objref)
            System.out.println(" obj ref returned " );
        else
            System.out.println(" unknown type returned " );
    }
}

// you must remember to clean up the iterator

// first narrow the iterator to a managed client iterator
com.ibm.IQueryManagedClient.DataArrayIterator damc_it ;
damc_it =
    com.ibm.IQueryManagedClient.DataArrayIteratorHelper.narrow
        (da_it.value);
damc_it.remove();
currentTransaction.commit(true);
}
catch (org.omg.CosTransactions.SubtransactionsUnavailable e)
{
    System.out.println("transcations not available excpetion " + e );
}
catch (Exception e)
{
    System.out.println(" system exception " + e);
}
}
}

```

Memory management

When you run a query using the `evaluate()`, `evaluate_to_iterator()` or `evaluate_to_data_array()` method, query builds a result collection in the

application server. An iterator over this result collection is returned to the client. The memory used by this result collection is released only when the `remove()` method is run on the iterator. Remember to run the `remove()` method on an iterator in both your mainline and your exception code paths. If you do not do this, your server will run short of memory after you run many queries.

Usage of Naming Service by query

Collection names appearing in the FROM clause of queries are resolved using the Naming Service. To save the cost of repeated calls to the Naming Service, the Query Service internally caches collection names. If the binding of a name in the Naming Service changes, the change will not be seen by the Query Service until the Component Broker server is restarted.

A parameter list can be passed as an argument to a query evaluator to be used in conjunction with a query. When resolving collection names appearing in the FROM clause of a query, the system first looks in the parameter list before going to the name space. This can result in a significant performance optimization and it is most commonly used for collections that are not identified in the name space. Collection names taken from the parameter list are considered volatile; in contrast with names taken from the name space that are considered stable. The system still caches metadata associated with volatile names, but performs a minimal verification upon the usage of a name in every query to determine if the collection references objects of the same type. If so, the metadata is kept in the internal catalog, otherwise, the metadata is discarded and reacquired. The verification is performed if a name space collection name appears as a parameter list name in another query. Similarly, the verification is also performed if a parameter list name appears as a name space collection name in another query.

Limit on number of query iterators per transaction

The Query Service cannot have more than 64 open SQL cursors per transaction. SQL cursors are also used by the Cache Service. Therefore you may be limited to 64 or less active query iterators in a single transactions.

Query Service tips

The following tips will help you in using the Query Service. Some tips cite specific locations where further information can be found.

- Conditions required for queries on page
- Deferred updates and query statement processing on page
- DB2 LOBs and DB2 data types are not supported on page

- Reset the timeout to be greater than the default or to zero on page
- A query over persistent objects must be executed within the scope of a transaction on page
- Use parameter lists instead of Named collections on page
- Support for object relationships is limited to 1 to 1 and 1 to Many relationships on page
- Use the query evaluator of a Component Broker server to query collections local to that server on page
- Use the foreign key pattern in Object Builder for better performance on page

Conditions required for queries You can query Home Collections, Views and Reference Collections provided you meet the following conditions:

1. The interface of objects to be stored in collections is defined as “queryable” in Object Builder.
2. The home inherits from IBOIMIQueryableHome.
3. For view collections, the underlying home collection is queryable.
4. The elements of the Reference Collection are defined as queryable. If the object interface is not queryable, you can still create Reference Collections containing these objects but you cannot query the collection.
5. The Reference Collection should be created with the createCollectionFor() operation.

If you do not use the createCollectionFor() then you must supply the interface name in the query statement FROM clause as in the example:

```
select r from MyReferenceCollection.acct r where r.Name = 'Bob';
```

In this example *acct* is the interface name of the objects in the collection.

Deferred updates and query statement processing When there are deferred updates, the query may not take the deferred updates into account when processing the query statement. For example:

```
aMO->name("NewName");
it = aHome->evaluate(" name='NewName'");
// the object aMO will likely not be returned in the iterator result set
// because the search "name='NewName'" was performed against
// values in the database.
```

To make sure that the query search sees the current values, issue a commit() if possible before doing the query.

```
aMO->name("NewName");
currentTransaction->commit(1 );
it = aHome->evaluate(" name='NewName'");
// the object aMO will now be returned by the query search.
```

DB2 LOBs and DB2 data types are not supported. Extended data types such as DB2 LOBs and user-defined DB2 data types are not supported.

Reset the timeout to be greater than the default or to zero. The default ORB request timeout value of 180 seconds may be insufficient when executing queries from a client. Reset the timeout to either a higher value or to zero to wait indefinitely.

A query over persistent objects must be executed within the scope of a transaction. The iterator returned from query must be used to retrieve the result set before ending the transaction. The iterator becomes invalid at end of the transaction. Refer to the following example.

```
currentTransaction->begin();
IManagedIterator_var it = myHome->evaluate (" amount > 10");
currentTransaction->commit();
currentTransaction->begin();
while (aMO= it->next())
{ . . . // do something with aMO }
// unpredictable behavior because you are using
// an iterator outside the transaction scope in which
// it was created.
```

This may effect the design ManagedObjects whose methods return query iterators if the ManagedObject is configured into an atomic container. The iterator will become invalid because of the implicit commit done by the atomic container at request termination. See the “More on Iterators” section in the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

Use parameter lists instead of Named collections. The use of parameter lists to pass object references to collections to query may perform better than having the Query Service retrieve the reference from the Naming Service. See “An example using the query evaluator interface” on page 378 for an example of how to use a parameter list.

Support for object relationships is limited to 1 to 1 and 1 to Many relationships. Relationships with outer join are not supported.

Use the query evaluator of a Component Broker server to query collections local to that server. Each Component Broker server has a query evaluator and only that query evaluator can be used to query collections that are local to the server. A query evaluator in server A cannot be used to query collections that reside in server B. Nevertheless, queries can span multiple back end datastores provided that the Component Broker server has connections to each datastore. So a single query statement can join collections C1 and C2 provide that C1 and C2 are defined in same server as the query evaluator even if C1 is mapped to database D1 and C2 is mapped to database D2.

Use the foreign key pattern in Object Builder for better performance of object relationships. Use Object Builder object relationship foreign key pattern for better performance of object relationships. When defining object relationships and object attributes that are object references, use the foreign key pattern in Object Builder. Queries expressions that use the object references can be pushed down to the datastore resulting in better query performance.

```
interface person {
    attribute read only long id;
    attribute string name;
}

interface account {
    attribute read only long acct_id;
    attribute person acct_owner;
}
```

If `acct_owner` is stored as a stringified object reference or handle, then the query

```
select a from accountHome a where a.acct_owner.name like 'Bob%';
```

will not be pushed down to the datastore resulting in the more data being read from the database and the query performed in object space. If `acct_owner` is stored as a relational foreign key in the account table then this query can be pushed down and performed as a relational join between the account and person tables. Only the qualifying rows will be read by the Component Broker application server.

Chapter 13. Cache Service



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

The objective of the Cache Service is to provide better performance and concurrency for applications. The Cache Service does this by following these heuristics:

- Keeping read-only data (or mostly read-only data) resident in memory. If data does not change very often it should not be necessary to have to reread the data from the database on every transaction. The **refresh interval** cache option can be used to control how often data is reread from the database.
- Data whose application must have the most current value must be read from the database for every transaction and the data must be locked in the database to guarantee that the data does not change.
- The administrator can control the isolation level of locking used by the database manager through use of the **lock confidence** and **access** cache options.
- The user can specify whether to defer updates until commit. If a transaction updates two attributes on an object instance, using deferred updates will result in only one SQL update. If the updates are done immediately when the attribute value changes, then two SQL updates will be done. The advantage of deferring update is reduced SQL calls and better performance. The advantage of not deferring updates is that the application gets immediate feedback (with an exception) if the update violates some database constraint. If you defer updates until commit and a database constraint is violated, the only feedback that the client application receives is that the transaction commit failed and the transaction is rolled back. The client application will have to repeat the entire transaction. Even so, it is usually recommended to defer updates.

There are four configuration attributes for the Cache Service. Each managed object type (interface name) can have its own configuration. The attributes are:

- Lock confidence (pessimistic or optimistic)
- Refresh interval (specified in number of seconds to retain data in memory)
- Defer update (yes or no)
- Access (read, write, upgrade; applicable to DB2 only)

Pessimistic caching holds a lock on the database record and data is read for every transaction. Optimistic caching does not hold a lock on the database record after reading the record. When using pessimistic locking, if access is “read” or “upgrade”, a share lock is maintained on the database record to prevent concurrent updates. If access is “write”, an exclusive lock is maintained on the database record to serialize all access. “Access=write” is useful when reading and updating a sequential counter to serialize concurrent transactions that attempt to read and the update the counter. “Refresh interval” does not apply when the pessimistic cache option is used.

If data that is being cached optimistically is updated, there is a possibility that the record in the database may have been updated by another transaction or may have been deleted. When optimistically cached data is written back to the database, a read and compare is done to make sure that the data record still exists in the database and the data values have not changed. This prevents any “lost updates” from occurring. Only the attributes that are actually updated are compared. The read-and-compare operation does not apply to long varchar columns in a table. When updating a long varchar column you should also update some other column (such as date of last update) to guarantee data integrity.

When using optimistic caching, there is potential for a large amount of virtual memory to be used in caching data. A limit on the size of the cache can be specified. The cache uses an LRU algorithm to manage the cache to this specified size. Access does not apply for optimistic caching.

Table 18 on page 395 and Table 19 on page 395 contain summaries of the valid cache options. You set these options using the System Manager User Interface. If no Profile with the name “iDefault” exists or no profile class exists for a particular managed object, then the default profile class consists of the following:

- Lock confidence: pessimistic
- Defer update: yes
- Access: read

For more information, see *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*.

Table 18. Optimistic cache options

Defer Updates	Refresh Interval	Access	Comments
yes	=0	not used	Database locks are released after reading the data. This minimizes database lock resource usage. Updates are deferred. Because the refresh interval is zero, data is purged from memory at the end of the transaction. Every transaction rereads data from the database so data is current. This is the recommended option for long running transactions. It prevents the long running transactions from causing concurrency problems at the database server.
yes	>0	not used	Same as above except data stays in memory at the end of the transaction. The refresh interval specifies how many seconds data stays in memory until a refresh. This is the recommended option for read-only data like TAX tables, ZIP CODE tables and product descriptions.
no	=0	not used	Similar to above except updates are not deferred. An SQL update is performed whenever an attribute changes value. If the transaction needs immediate feedback on any database integrity violations, use this option.
no	>0	not used	Similar to above except data stays in memory and can be reused by other transactions. Updates are not deferred. The refresh interval specifies the number of seconds for data to stay in memory.

Table 19. Pessimistic cache options

Defer Updates	Refresh Interval	Access	Comments
yes	not used	read or upgrade	Data is locked (share locks) in the database during the transaction. Data is purged from cache at commit. Updates are deferred until commit. This is the recommended option for data that changes often or if the application must have current data. Examples might be inventory quantities or bank balances.
no	not used	read or upgrade	Same as previous except updates are not deferred. An SQL update is performed whenever as object attribute changes value.

Table 19. Pessimistic cache options (continued)

Defer Updates	Refresh Interval	Access	Comments
yes	not used	write	Data is locked exclusively in the database during the transaction. Data is purged from cache at commit. Updates are deferred until commit. This is the recommended option for data that must be serialized for both reading and writing and there is a high probability of contention among multiple transactions such as reading and updating a sequential number generator.
no	not used	write	Same as previous except updates are not deferred. An SQL update is performed whenever an object attribute changes value.

For information on how to use Object Builder using the Cache Service, see *WebSphere Application Server Enterprise Edition Component Broker Application Development Tools Guide*.

For information on how to configure the Component Broker application server to use the Cache Service or how to configure your database to use the Cache Service see *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*.

Cache Service and DB2 locking considerations

When reading records from a DB2 database, the Cache Service uses CS database locks when optimistic locking is specified and RS locks for pessimistic locking.

Cache Service limits

The Cache Service can have a maximum of 64 SQL cursors per transaction. One cursor is used for each table accessed by the Cache Service and one cursor is used for each active query iterator. During a transaction, accessing tables and query iterators whose combined number exceeds 64 will cause an exception.

Cache Service and Oracle locking

When reading records from an Oracle database, the Cache Service obtains an exclusive lock when “lock confidence=pessimistic”. When “lock confidence=optimistic” no lock is obtained on the data record. For data retrieved using an OOSQL query, the record in the database is never locked. The setting of the “access” configuration attribute has no effect on the type of locking used for an Oracle database.

Chapter 14. Object Request Broker

The Object Request Broker (ORB) enables objects to transparently make requests (invoke methods) and receive responses from local or remote objects. The ORB supports remote method calls for distributed C++ and Java applications.

The following topics describe the Object Request Broker:

- “Remote method invocation”
- “Dynamic invocation interface (DII)” on page 403

Remote method invocation

This section includes the following topics:

- “Conversion of objects to string form”
- “Code-set conversion for remote method invocations” on page 400

Conversion of objects to string form

Both proxy objects and pointers to local objects are kinds of “object reference”. An object reference contains information that is used to identify a target object. For example, a pointer to a local object contains the physical address of the object; a proxy object contains information to locate the target server and the target object within that server.

Sometimes, it is useful to convert object references to a string form (for example, to save references in a file system or to exchange object references with other application processes). This technique can be used, for example, to allow a client process to obtain a proxy to an object residing in a server process, for one client process to “give” a proxy object to another client process, or for a server process to record an object reference in a form that is meaningful beyond the lifetime of the process. Object references to and from string form are used when object references are passed between clients and servers using remote method invocations.

The ORB class defines a method for converting object references (both local object pointers and proxy objects) to an external form. This external form is a string that can be used by any process to identify the target object. The ORB class also supports the translation of these strings back into the original local

objects or equivalent proxies. The ORB methods for converting between object references and their string representations are specified by the following IDL:

```
string object_to_string (in Object obj);  
Object string_to_object (in string str);
```

- When a string refers to a remote object (an object not residing in the same address space as the calling process), the `string_to_object` method always returns a new proxy object for that string. The returned proxy is not the same as the proxy passed to `object_to_string`, and repeated invocations of `string_to_object` each return different proxy objects. These duplicate proxies can be destroyed using the `CORBA::release` function.
- When a string refers to a local object (an object residing in the same address space as the caller), the `string_to_object` method returns a pointer to the local object. This is true even if the string was originally created by another process calling `object_to_string`, passing in a proxy to the object. If the local object to which the string refers no longer exists, and is not a persistent object that can be reactivated by the instance manager, an exception is raised.
- When `object_to_string` is invoked on a local object within a client-only process (a process that has not called `impl_is_ready` on the BOA object), the resulting string has validity only as long as the object exists within that process, and only within that process.
- When `object_to_string` is invoked on a local object within a server process (one that has called `impl_is_ready` on the BOA object), the resulting string can be distributed to other processes, which can then call `string_to_object` with the string to generate a proxy to the original object.

As with other forms of object references, the lifetime of a string reference to an object in a server depends on the implementation of the server. If the server's instance manager supports persistent objects, and the object is persistent, then the reference is valid even after the server process terminates. (The next time the reference is used by a client, the location-service daemon will restart it, and the server can reactivate the referenced object.)

The string form of an object reference (the result of calling `object_to_string`) should be considered opaque to application programmers. The only assumption that can be made about such a string is that it can be passed to `string_to_object` to locate the original object.

Two different strings can refer to the same object. Generally, it is not safe for an application to use the strings as unique object identifiers.

Code-set conversion for remote method invocations

Configuration settings are available that allow applications to request that the ORB perform code-set conversion for all character and string data transmitted

over the network in remote method invocations. This applies to method parameters and return results that have been declared as IDL type char, string, or constructed types composed of char or string types. The ORB performs code-set conversion using the XPG4 libraries and locales, which must be available and properly configured on both communicating systems. The code-set conversion is performed according to the OMG "IDL Type Extensions" specification. Because the OMG specification is based on OSF code set numbers, rather than XPG4 code set names, the configuration settings to enable code-set conversion must be set using OSF code set numbers, rather than XPG4 code set names.

The following configuration settings support code-set conversion. These settings can be updated in the appropriate client or server image using the systems management tools.

In client and server images:

translation enabled

When set to 1 (or any non-zero, non-NULL value), code-set conversion is enabled. This setting has no effect unless set to 1 for both the client and the server. Because the code-set conversion support requires the client and server to exchange IIOB 1.1 messages, the server should not be configured to generate IIOB 1.0 messages, otherwise this setting has no effect. (The default is for servers to generate IIOB 1.1 messages.) The default for this setting is zero.

use ISO-Latin1

When set to 1 (or any non-zero, non-NULL value), indicates that the process should transmit all character and string data in the ISO-Latin1 codeset. This setting provides a more limited form of code-set conversion than the above, and is provided to aid interoperability with other vendors' ORBs. This setting is meaningful when the above setting is zero or when communicating with a remote process that communicates using IIOB 1.0 messages. Both client and server should use the same setting. The default setting is zero.

native char code set

This setting is the OSF number of the native codeset used by the process for single-byte char and string data. This setting is optional (if unset or zero, it is calculated automatically using the XPG4 `nl_langinfo()` function.) If present, either this setting or the "native wchar code set" setting must match the process's actual native code set, as determined by the XPG4 `nl_langinfo()` function.

native wchar code set

The OSF number of the native codeset used by the application for wchar and wstring data. This setting is needed only if both the application and the ORB support the IDL types wchar and wstring. If

present, either this setting or the "native code set" setting must match the process's actual native code set, as determined by the XPG4 `nl_langinfo()` function.

In server images only:

char code sets

These are the codesets that a server can translate to/from its native code set, for single-byte char and string data, specified as a space-delimited list of OSF codeset numbers. (The list can be empty.) This list of code sets gets advertised in the object references that the server exports; clients then choose either the server's native code set or one of the code sets in this list as the transmission code set.

wchar code sets

These are the codesets that a server process can translate to/from its native code set, for wchar and wstring data, specified as a space-delimited list of OSF codeset numbers. (The list can be empty.) This setting is needed only if both the server application and the ORB support the IDL types wchar and wstring.

When a server is configured to perform code-set conversion, the object references that it exports contain information about the server's native code sets and the additional code sets that the server supports (for both char and wchar data), based on the configuration settings above. When a client that is configured to perform code-set conversion invokes a method on such an object reference, the client selects two "transmission code sets", the code sets in which char and wchar character data (in method parameters and return results) will be transmitted between the client and the server that exported that object reference. This selection is based on the client's native code set and the code set(s) advertised in the object reference. If the client and server are using the same native code sets, then no conversion is performed. Otherwise, a transmission code set is selected using the following priorities:

1. The server's native code set.
2. The client's native code set (if supported by the server).
3. Some other code set the server supports to which the client can translate.
4. Unicode (UTF-8 for char data, UTF-16 for wchar data).

The client then translates all character data into the transmission code set before sending it to the server. Messages from the client to the server contain information about which transmission code sets are being used, so that the server can translate the incoming character data into the server's native code set and translate outgoing character data into the transmission code set. If the client's message to the server does not specify that character data has been converted, then the server does not perform this conversion either. In this

case, character data is transmitted in ISO-Latin1 (if designated by the "use ISO-Latin1" configuration setting) or is transmitted in the sending process's native code set.

To give the application an opportunity to change the native code set (for instance, using the XPG4 `set_locale()` interface), initialization of the ORB's code-set conversion facility is not performed until the application calls `CORBA::ORB_init()`. Hence, an application that requires code-set conversion must establish its native code set prior to calling `CORBA::ORB_init()`, and must call `CORBA::ORB_init()` prior to making remote method invocations.

If any errors are encountered during code-set conversion, the ORB will throw a `DATA_CONVERSION` error. This can occur, for example, if the configuration settings are incorrect, if an incorrect IIOP 1.1 message is received from the remote process, or if the conversion is not possible (because the client and server are using incompatible character sets, or because the client or server does not have the necessary XPG4 code-set converters available).

Dynamic invocation interface (DII)

The dynamic invocation interface (DII) allows client applications to dynamically build and invoke requests on objects, even if the operation to be invoked is not known at compile time or the client bindings for the operation have not been compiled and linked into the client application. In addition, the DII allows a client application to make synchronous, *deferred-synchronous*, or oneway invocations. (A deferred-synchronous request is one in which the sending of the request and the receiving of the response are done separately, allowing the client application to perform other processing in the interim.) Multiple deferred-synchronous and oneway requests can be sent simultaneously, allowing batch processing.

When using DII, exceptions raised by the remote implementation are thrown to the client and are returned in Environment objects, because the client may not be prepared to catch user-defined exceptions for dynamic requests. If an exception has been raised, as indicated by the Environment object associated with the Request, then the inout/out parameter values and the return values will not be meaningful, just as with static requests.

Building a DII request

Before invoking a DII request, the application must first construct a `CORBA::Request` object to represent the request. The `CORBA::Request` object can be created using the `CORBA::Object::_create_request` method or the `CORBA::Object::_request` method. The application invokes one of these methods on the proxy object that is the target of the DII request.

After constructing a `CORBA::Request`, the application should call `CORBA::Request::set_return_type` to update the `CORBA::NamedValue` contained in the `Request` with the appropriate operation return type.

Using `CORBA::Object::_create_request`

When using one of the `CORBA::Object::_create_request` methods, the application supplies the operation name, as well as the `CORBA::NVList` representing the method parameters (previously constructed using `CORBA::ORB::create_list` or `CORBA::ORB::create_operation_list`) and a `CORBA::NamedValue` used to contain the return result (previously constructed using `CORBA::ORB::create_named_value`).

Using `CORBA::Object::_request`

When using the `CORBA::Object::_request` method to construct the `Request`, the application supplies only the operation name; the parameters for the request and the return type must be added afterward using methods on `CORBA::Request`. When using this technique, the application is not required to explicitly construct a `CORBA::NVList` to represent the operation parameters or a `CORBA::NamedValue` to contain the return result.

The `CORBA::Object::_create_request` method is overloaded to provide two signatures. The only difference between them is that one allows the application to supply a `CORBA::ContextList` object and a `CORBA::ExceptionList` object. These objects specify a list of `Context` strings that must be sent with the operation, if any, and a list of the user-defined exceptions that can be thrown by the operation, if any. The application may want to provide the `ContextList` and `ExceptionList` objects to the `CORBA::Object::_create_request` method so that the ORB does not need to perform any potentially expensive interface repository lookups when the request is later invoked.

Constructing `NamedValue` objects

`CORBA::NamedValue` objects are used in `NVLists` to represent operation parameters for DII requests. `CORBA::NamedValue` objects are also used as placeholders for return results of methods invoked using the DII.

A `CORBA::NamedValue` has attributes for an optional parameter name, a `CORBA::Any` (containing a `CORBA::TypeCode` representing the parameter type and a `void*` value pointer), and flags to indicate the parameter mode (`CORBA::ARG_IN`, `CORBA::ARG_INOUT`, or `CORBA::ARG_OUT`). When a `CORBA::NamedValue` represents a method return value, the name and flags are unused.

The `CORBA::Any` stored in a `CORBA::NamedValue` is created automatically by the `CORBA::NamedValue` constructor; it can then be manipulated using the usual `CORBA::Any` interface.

The `CORBA::NamedValue` objects contained in a `CORBA::NVList` are created and initialized automatically when the `CORBA::NVList` is created, alleviating the programmer from the need to explicitly create the `CORBA::NamedValue` objects to represent the operation parameters. See “Constructing `NVList` objects” on page 404 for more information. The `CORBA::NamedValue` objects contained in an `NVList` can be accessed using methods of `CORBA::NVList`.

Depending on how the application creates the `CORBA::Request` object for a DII request (see Building a Request), the application may or may not need to explicitly create the `CORBA::NamedValue` object to contain the return result of the request. If necessary, the application can use the `CORBA::ORB::create_named_value()` method to create a new `CORBA::NamedValue`. This method creates a `NamedValue` whose embedded `TypeCode` is `CORBA::_tc_null`; the application must call `CORBA::Request::set_return_type` (after creating a `Request` object to contain the `NamedValue`), to initialize the `NamedValue` for a specific IDL operation.

Constructing `NVList` objects

A `CORBA::NVList` object contains an ordered set of `CORBA::NamedValue` objects, representing the method parameters to be used for a DII request invocation. Depending on how the application creates the `CORBA::Request` object for a DII request (see “Building a DII request” on page 403), the application may or may not need to explicitly create the `CORBA::NVList` object.

`CORBA::NVList` objects can be created using either the `CORBA::ORB::create_list` or the `CORBA::ORB::create_operation_list` method. The difference between these two methods is that `CORBA::ORB::create_list` creates a generic `NVList` of a specified length, whose `CORBA::NamedValue` elements are uninitialized, whereas `CORBA::ORB::create_operation_list` creates an `NVList` whose `CORBA::NamedValue` elements are already initialized for a specific IDL operation (minus the parameter values). The latter approach requires, however, that the application supply a `CORBA::OperationDef` object that it has retrieved from the interface repository.

After creating an `NVList` using `CORBA::ORB::create_operation_list`, the application must initialize the parameter values in the `NVList`. This can be done by accessing the appropriate `NamedValue` in the `NVList`, accessing the `Any` in the `NamedValue`, then using standard `Any` operations to insert the appropriate value. The `NamedValue` objects in the `NVList` are ordered according to the order of parameter declarations in IDL.

As an alternative to creating a `CORBA::NVList` using `CORBA::ORB::create_operation_list`, an application can create an empty (length zero) `CORBA::NVList` using `CORBA::ORB::create_list`. The `CORBA::NVList` class provides several methods for adding new `CORBA::NamedValue` objects to an (initially empty) `NVList`. These methods vary in which of the `NamedValue`'s attributes are initialized (name, value, or flags) and in whether the `NamedValue` takes ownership of the memory used to initialize it.

The `CORBA::NVList` associated with a DII request can be accessed using the `CORBA::Request::arguments()` method. The `CORBA::NVList` class provides methods for accessing and deleting a particular `CORBA::NamedValue` element given its index in the `NVList`, and a method for returning the size of the `NVList`.

Constructing a DII request

To invoke a request using the dynamic invocation interface (DII), the client must explicitly construct a `CORBA::Request` object to represent the request, and must initiate the request by invoking a method on the `Request` object. A `Request` object embodies all the information needed to invoke the request, including the proxy object on which it is to be invoked, the operation name, the operation parameters, and a place to store the return result. The operation parameters are represented in the `Request` by a `CORBA::NVList` object, that is an ordered set of `CORBA::NamedValue` object. Each `NamedValue` object represents the name and mode of a parameter, and a `CORBA::Any` object to hold its `CORBA::TypeCode` and value. The return result of the operation is stored in the `Request` object using another `NamedValue` object.

Initiating a DII request

After a `CORBA::Request` object is constructed using the `CORBA::Object::_request` or `CORBA::Object::_create_request` method, the application can issue a DII request in one of the following ways:

- Invoke the request synchronously, using `CORBA::Request::invoke()`.
- Send the request oneway (with no response expected), using `CORBA::Request::send_oneway()`.
- Send the request, with the response to be requested at a later time, using `CORBA::Request::send_deferred()`.
- Send the request oneway as part of a batch of requests, using `CORBA::ORB::send_multiple_requests_oneway()`.
- Send the request as part of a batch of requests, with the response to be requested at a later time, using `CORBA::ORB::send_multiple_requests_deferred()`.

When a request is invoked synchronously (using `CORBA::Request::invoke`), the results of the invocation are available immediately after calling `invoke()`. When a request is sent using either `CORBA::Request::send_deferred` or `CORBA::ORB::send_multiple_requests_deferred`, the application must explicitly call for a response, using `CORBA::Request::get_response` or `CORBA::ORB::get_next_response`. The response to a Request sent using `CORBA::Request::send` can be retrieved using either `CORBA::Request::get_response` or `CORBA::ORB::get_next_response`, and similarly with Requests sent using `CORBA::ORB::send_multiple_requests_deferred`. There is no guarantee that results will be returned in the same order that deferred requests were sent.

If the response is not yet available when the application calls `CORBA::Request::get_response` or `CORBA::ORB::get_next_response`, the application blocks until the response is available. Applications can avoid blocking by first calling `CORBA::Request::poll_response` or `CORBA::ORB::poll_next_response` to determine whether the response is available.

After a response to a DII request is retrieved, the application can access the inout/out parameter values and the return value, or any exception that was thrown, by examining the Request object. The inout and out parameter values can be accessed using the `NVList` associated with the Request, returned by `CORBA::Request::arguments()`. The Any containing the return value can be accessed using either the `CORBA::Request::result()` or `CORBA::Request::return_value()` methods.

If an exception was thrown by the remote request, it is stored in a `CORBA::Environment` object, which can be accessed using the `CORBA::Request::env()` method. When using DII, exceptions are not thrown to the client, but are returned in Environment objects, because the client may not be prepared to catch user-defined exceptions for dynamic requests. If an exception was thrown, as indicated by the Environment object associated with the Request, then the inout/out parameter values and the return values are not be meaningful.

Sample DII requests

This sample application uses the DII to invoke a request with the following IDL signature:

```
interface testObject
{string testMethod (in long input_value, out float out_value);
};
```

In the following example, the application is written with knowledge of the name and signature of the method to be invoked using DII. In general, however, an application might discover the name or signature at run time

(using the interface repository or application input), and would create and examine the TypeCodes used to represent parameter and return types.

```
try {  
  
    // Get the OperationDef that describes testMethod:  
  
    CORBA::ORB_var myorb = CORBA::ORB_init (argc, argv, "DSOM");  
    CORBA::Object_var generic_IR =  
    myorb->resolve_initial_references ("InterfaceRepository");  
    CORBA::Repository_var my_IR = CORBA::Repository::_narrow (generic_IR);  
    CORBA::Contained_var generic_opdef = my_IR->lookup (  
        "testObject::testMethod");  
    CORBA::OperationDef_var my_opdef = CORBA::OperationDef::_narrow(  
        generic_opdef);  
  
    // Create the NVList and NamedValue for the request:  
  
    CORBA::NVList_ptr params = NULL;  
    myorb->create_operation_list (my_opdef, params);  
    CORBA::NamedValue_ptr result = NULL;  
    myorb->create_named_value (result);  
  
    // Create the Request object:  
  
    CORBA::Object_var my_proxy = /* get a proxy somehow */  
    CORBA::Request_ptr my_request;  
    my_proxy->_create_request (  
        NULL, "testMethod", params, result, my_request, 0);  
    *(my_request->arguments()->item(0)->value()) <<=  
        (CORBA::Long) 12345;  
    my_request->set_return_type (CORBA::_tc_string);  
  
    // Invoke the request and get the return value and out parameter value:  
  
    my_request->invoke();  
    CORBA::Exception_ptr my_exception = my_request->env()->exception();  
    if (!my_exception) { // no exception occurred  
        CORBA::String_var return_string;  
        CORBA::Float out_float;  
        my_request->return_value() >>= return_string;  
        *(my_request->arguments()->item(1)->value()) >>= out_float;  
        // Do something application-specific with return_string and out_float.  
    }  
    else { // an exception occurred  
        // Do something application-specific with my_exception.  
        // Use my_exception->id() to determine what kind of exception it is.  
    }  
    CORBA::release (my_request);  
  
}
```

```

catch (CORBA::SystemException) {
    // appropriate exception handling
}

```

The following code example is a variation of the previous one, in which the Request object is created and initialized differently (using a technique that does not require the application to access the interface repository), and in which the DII request is a deferred synchronous request (rather than an asynchronous request).

```

try {

    // Create the Request object:
    CORBA::Object_var my_proxy = /* get a proxy somehow */
    CORBA::Request_ptr my_request = my_proxy->_request ("testMethod");
    my_request->add_in_arg() <<= (CORBA::Long) 12345; // sets type and value
    CORBA::Float out_float;
    my_request->add_out_arg() <<= out_float; // sets type
    my_request->set_return_type (CORBA::_tc_string);

    my_request->send_deferred();
    // ... Do some application-specific processing for a while ....

    // Do "my_request->poll_response();" to see if the Request is ready,
    // to avoid blocking on the next instruction:

    my_request->get_response();

    // Get the return value and out parameter value:
    CORBA::Exception_ptr my_exception = my_request->env()->exception();
    if (!my_exception) { // no exception occurred
        CORBA::String_var return_string;
        my_request->return_value() >>= return_string;
        *(my_request->arguments()->item(1)->value()) >>= out_float;
        // Do something application-specific with return_string and out_float.
    }

    else { // an exception occurred
        // Do something application-specific with my_exception.
        // Use my_exception->id() to determine what kind of exception it is.
    }

    CORBA::release (my_request);

}

catch (CORBA::SystemException) {
    // appropriate exception handling
}

```

Dynamic skeleton interface (DSI)

The dynamic skeleton interface (DSI) provides a way for a server application to service requests on an object implementation for which it does not have server-side bindings. DSI can be thought of as the server-side equivalent to DII, although the use of DII on the client side and the use of DSI on the server side are independent. Just as a server application is unaware whether a client is using DII or client bindings, a client application is unaware whether a server is using DSI or server bindings.

To support DSI, a target object must support the `CORBA::BOA::DynamicImplementation` interface. The `DynamicImplementation` interface provides a single pure-virtual method, `invoke`. This method is called by the ORB/BOA to dispatch methods on the target object without requiring that the server be statically compiled and linked with the server-side bindings for the object. Implementations derived from `CORBA::BOA::DynamicImplementation` must provide an implementation of the `invoke` method, as described in “Enabling an Object for DSI Dispatching”.

The `CORBA::BOA::DynamicImplementation::invoke` method, implemented by application objects wishing to support DSI dispatching, takes as input a `CORBA::ServerRequest` object. The `CORBA::ServerRequest` object is similar to the client-side `CORBA::Request` object used for DII; it represents a dynamic request, and has methods for accessing the operation name, the parameters in the form of an `NVList`, etc. The `CORBA::ServerRequest` object is used by an application’s `DynamicImplementation::invoke` implementation, to get information about a request to be dispatched dynamically, and to inform the ORB/BOA of the results.

- The `CORBA::ServerRequest::params` method is called by the application to ask the ORB to store the in/inout parameter values for the request in the given `NVList`. (An `NVList` is passed as an inout parameter to `ServerRequest params`; the application is responsible for initializing it with the appropriate `TypeCodes` and flags for the request so that the ORB knows how to demarshal the parameters.)
- The `CORBA::ServerRequest::result` method is invoked by the application’s `DynamicImplementation::invoke` method, to inform the ORB/BOA of the result of a dynamically invoked request, so that it can marshal the result and any output values.
- The `CORBA::ServerRequest::exception` method is invoked by the application’s `DynamicImplementation::invoke` method, to inform the ORB that an exception was thrown by a method that was dynamically invoked, so that it can marshal the exception.

Figure 9 on page 410 shows the flow of control that occurs when a method is dispatched using DSI. The shaded box shows actions that must be performed by the application; the unshaded boxes show actions performed by the ORB/BOA. When a request comes into the server, the BOA locates the target object and passes the request to the object's skeleton. When the target object is a CORBA::BOA::DynamicImplementation, the object's skeleton is (automatically) a dynamic skeleton (rather than a static skeleton provided by server-side bindings). The dynamic skeleton creates a CORBA::ServerRequest object to represent the request, and calls the invoke method on the target object, passing the CORBA::ServerRequest.

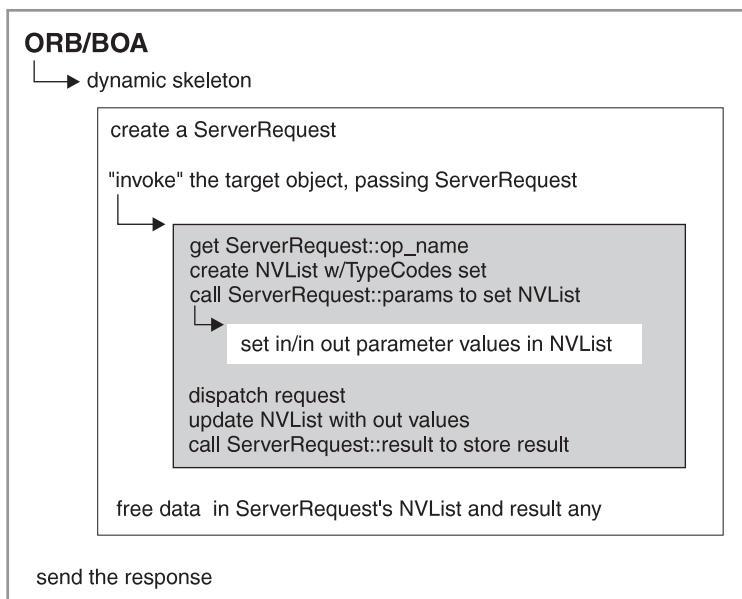


Figure 9. Dispatching a method using DSI

The target object's implementation of the invoke method must use the input CORBA::ServerRequest object to discover the operation name and the parameter values for the request. After the target object dispatches the method, it must update the ServerRequest with output and return values. The ORB/BOA and the dynamic skeleton are responsible for deleting the CORBA::ServerRequest object and the method parameters and return values, just as with static calls.

To support DSI, an application object must support the CORBA::BOA::DynamicImplementation interface. The DynamicImplementation interface provides a single pure-virtual method, invoke. This method is called by the ORB/BOA to dispatch methods on the target object without requiring that the server be statically compiled and

linked with the server-side bindings for the object. Implementations derived from `CORBA::BOA::DynamicImplementation` must provide an implementation of the `invoke` method.

The following actions are typically performed within an application's implementation of the `CORBA::BOA::DynamicImplementation::invoke` method. Every method should be supported.

1. Get the operation name to be dispatched from the input `CORBA::ServerRequest` object.
2. Create an appropriate `CORBA::NVList`, containing `CORBA::TypeCode` objects (but not values) for the signature of the method to be dispatched.
3. Call the `params` method of the `ServerRequest` object, passing in the new `CORBA::NVList`. In response, the `CORBA::ServerRequest` (with the help of the ORB), fills in the `in` and `inout` parameter values in the `CORBA::NVList`.
4. Implement the requested operation, using the `in` and `inout` parameter values now found in the `CORBA::NVList`, catching all exceptions.
5. If no exception occurred, store the resulting output parameter values in the same `CORBA::NVList`.
6. If no exception occurred, and the return type of the dispatched method is not `void`, call the `result` method on the `CORBA::ServerRequest` object to store the result.
7. If an exception was thrown by the dispatched method, call the `exception` method on the `CORBA::ServerRequest` to record the exception.
8. Return control to the ORB. The ORB/BOA then sends a response to the waiting client.

Chapter 15. Non-IBM ORB usage

The IBM Component Broker Java Client comes prepackaged with a Java ORB developed by IBM. It also includes client-local access to IBM implementations of security and transactions, plus pre-generated client stub bindings for naming, lifecycle, events, notification, query, externalization, identity, and properties. Component Broker also generates Java client stubs that a client can use to access Component Broker managed objects. All these client stubs work with the Component Broker Java ORB.

If you have a client environment that does not use Component Broker security and transactions, and you are willing to use non-IBM emitters to generate the client stubs for CORBA services (for example, lifecycle and events) and for Component Broker managed objects, then you can use a non-IBM client ORB to access Component Broker managed objects. The remaining parts of the section describe, by way of a simple example, how to do this.

The example

The remaining parts of this section use a concrete example to describe how a non-IBM client ORB can be used to access a Component Broker managed object. The example is very simple. It uses the existing Component Broker Policy sample managed object and its associated PolicyHome. The example merely uses the PolicyHome to create a new instance of a Policy object and call a method on the object. The idl for this object can be found in Policy.idl:

```
#ifndef _Policy_idl
#define _Policy_idl

#include <IManagedClient.idl>
interface Policy : IManagedClient::IManageable
{
    attribute float amount;
    readonly attribute long policyNo;
    attribute float premium;
    void addBeneficiary( );
    void delBeneficiary( );
};
```

The idl for the PolicyHome is:

```
#ifndef _PolicyHome_idl
#define _PolicyHome_idl

interface Policy;
#include <IManagedClient.idl>
#include <Policy.idl>
```

```

interface PolicyHome : IManagedClient::IHome
{
    Policy create(in float premium,in float amount )
        raises (
            IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy defaultCreate( )
        raises (
            IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy createWithNumber(in long policyNo )
        raises (
            IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy findByPolicyNumber(in long policyNo )
        raises (
            IManagedClient::IInvalidKey, IManagedClient::INoObjectWKey);
};

```

These idl files describe existing Component Broker managed objects.

The example creates a new Policy Object. The Java source for the example is:

```

import java.net.URL;

import CosLifeCycle.GenericFactory;

public class PolicyTest
{
    public static void main (String[] args)
    {
        try
        {
            URL
            url = new URL ("http://xxx.xxx.com/bootstrap/NameServer.ior");
            Bootstrap
            bs = new Bootstrap (url, null);
            GenericFactory
            gf = bs.factory ("Policy", "object interface");
            PolicyHome
            ph = PolicyHomeHelper.narrow (gf);

            org.omg.CORBA.Object obj = ph.create (
                (float)100.00, (float) 10000.00);
            Policy p = PolicyHelper.narrow (obj);
            System.out.println ("Policy number = " + p.policyNo ());
        }
        catch (Exception e)
        {
            e.printStackTrace ();
        }
    }
}

```

The Bootstrap object created above is described in Bootstrapping. The remaining parts of this chapter describe the steps required to get the

PolicyTest example to talk to the existing Component Broker PolicyHome managed object to create a new Policy object.

Bootstrapping

IBM Java clients use the methods of CBSeriesGlobal to get attached to the Component Broker name server. Since CBSeriesGlobal is not available to non-IBM ORBs we need another way to get hooked into the Component Broker name server.

We first write and run a Component Broker program to get a reference to the Component Broker Naming Service, stringify the reference, and write it out to a file. This file must then be made available to the non-IBM client as a URL. In the preceding code for the PolicyTest, we assumed the URL is “http://xxx.xxx.com/bootstrap/NameServer.ior”.

The following Java program illustrates one way to generate the stringified IOR. It uses the IBM Java ORB’s implementation of resolve_initial_references to get a reference to the Component Broker Naming Service’s root context. This Java program is compiled in the Component Broker environment, using the Component Broker Java ORB.

Important: Be very careful about the CLASSPATH. You must be sure that only the IBM Java ORB classes are in the CLASSPATH when compiling this class.

```
import java.applet.Applet;

import java.io.FileWriter;
import java.io.PrintWriter;

import java.util.Properties;

import org.omg.CORBA.ORB;

// WriteRootNCIOR primes the naming service for access from a
// non-Component Broker CORBA client. It does this by getting a reference
// to the root naming context and creating a file containing the Naming
// Context's IOR (NameService.IOR). This file must be placed in a
// well-known location accessible as a URL file, the root WWW directory.
// Note that this program uses IBM's ORB. It uses the CB bootstrap
// mechanism.

// To invoke this program: java WriteRootNCIOR <bootstrapHost>
// Alternately, in a html file:
// <applet>
// Note: This assumes that the bootstrap port is 900.

// This program is an application, but it is also an applet, so
// it COULD be run through a browser.
```

```

public class WriteRootNCIOR extends Applet
{
    public WriteRootNCIOR ()
    {
        this.args = null;
    }

    public WriteRootNCIOR (String[] args)
    {
        this.args = args;
    } // ctor

    public void init ()
    {
        try
        {
            Properties props = new Properties ();
            props.put ("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
            props.put ("com.ibm.CORBA.BootstrapPort", "900");
            ORB orb = runningAsAnApplet ? ORB.init (
                this, props) : ORB.init (args, props);

            org.omg.CORBA.Object obj = orb.resolve_initial_references (
                "NameService");

            java.io.FileOutputStream fw = new java.io.FileOutputStream (
                "NameService.ior");
            PrintWriter pw = new PrintWriter (fw);
            pw.print (orb.object_to_string (obj));
            pw.close ();
        }
        catch (Throwable t)
        {
            System.out.println (
                "WriteRootNCIOR failed to create the file: NameService.ior");
            t.printStackTrace ();
        }
    } // init

    private String[] args;
    private boolean  runningAsAnApplet = true;

    public static void main (String[] args)
    {
        WriteRootNCIOR writer = new WriteRootNCIOR (args);
        writer.runningAsAnApplet = false;
        writer.init ();
    }
}

```

Once the stringified IOR has been written to a file, we need a way in the non-IBM client to use the IOR to find the Component Broker's root naming

context and then find the factory for our Policy Object. The following Java class illustrates one way to do is. This class is used in PolicyTest class. See The example for further information.

```
import java.io.DataInputStream;
import java.io.InputStream;

import java.net.URL;

import org.omg.CORBA.ORB;

import CosNaming.NameComponent;
import CosNaming.NamingContext;
import CosNaming.NamingContextHelper;

import CosLifeCycle.FactoryFinder;
import CosLifeCycle.FactoryFinderHelper;
import CosLifeCycle.GenericFactory;
import CosLifeCycle.GenericFactoryHelper;

// Bootstrap allows a non-CB CORBA Java client to access a CB Server.
// It assumes a URL containing the IOR for the Root Naming Context
// resides in a well known place.

// Make sure this file is compiled with the non-CB .class files

public class Bootstrap
{
    public Bootstrap (URL bootstrapURL, String[] orbArgs)
    {
        try
        {
            // Get ORB
            orb = ORB.init (orbArgs, null);

            // Get the IOR string for the Root Naming Context
            DataInputStream input = new DataInputStream (
                (InputStream)bootstrapURL.getContent ());
            byte[] bytes = new byte[1024];
            int bytesRead = input.read (bytes);
            String stringifiedIOR = new String (bytes, 0, bytesRead);

            // Use the Root Naming Context IOR string to instantiate a Root
            // Naming Context
            org.omg.CORBA.Object obj = orb.string_to_object (stringifiedIOR);
            rootNC = NamingContextHelper.narrow (obj);
        }
        catch (Exception e)
        {
            System.out.println ("Resolving the root naming context FAILED.");
            e.printStackTrace ();
        }
    }
} // ctor
```

```

public NamingContext rootNamingContext ()
{
    return rootNC;
} // rootNamingContext

public GenericFactory factory (String id, String kind)
{
    try
    {
        // Resolve to the factory finder
        NameComponent[] name = new NameComponent [4];
        name[0] = new NameComponent ("host", "");
        name[1] = new NameComponent ("resources", "");
        name[2] = new NameComponent ("factory-finders", "");
        name[3] = new NameComponent ("host-scope", "");
        org.omg.CORBA.Object obj = rootNC.resolve (name);
        FactoryFinder ff = FactoryFinderHelper.narrow (obj);

        // Resolve to the factory
        name = new NameComponent [1];
        name[0] = new NameComponent (id, kind);
        org.omg.CORBA.Object[] objs = ff.find_factories (name);
        return GenericFactoryHelper.narrow (objs[0]);
    }
    catch (Exception e)
    {
        System.out.println (
            "Getting the factory finder for
            <" + id + ", " + kind + "> FAILED.");
        e.printStackTrace ();
        return null;
    }
} // factoryFinder

private String      bootstrapHost;
private ORB         orb;
private NamingContext rootNC;
}

```

Creating the client bindings

Now we need to create the client side Java stubs for all the CORBA and Component Broker objects used in the example. `BootStrap.java` uses the `CosNaming` and `CosLifeCycle` modules, so the first step is to run these through a non-IBM idl-to-Java compiler. Since we are using these interfaces to talk to Component Broker implementations, the safest thing to do is to copy the Component Broker versions of the idl files to the client rather than using versions of the files that come with the non-IBM ORB.

The non-IBM Java ORB has a command to convert CORBA idl files to Java `.java` files. Use this command to create `.java` files for the interfaces in the `CosNaming` and `CosLifeCycle` modules.

Policy.idl uses Component Broker's IManagable interface in the IManagedClient module. (See Policy.idl example in The example). IManagedClient::IManagable, in turn uses interfaces in CORBA's CosStream, CosLifeCycle, CosNaming, and CosObjectIdentity modules. In addition, PolicyTest uses PolicyHome. So we must copy all the idl files for all these modules to the client and run them through the non-IBM ORB's IDL-to-Java compiler.

The example being used in this section is very simple. More complicated examples may use interfaces with more dependencies. The general procedure is to copy over from the server the idl files for the interface you are using, and also copy the idl for all the CORBA and Component Broker interfaces that are used by your interface, and further up the chain until all the needed idl is available. Then produce the Java client stubs for these interfaces.

The final step is to compile all client-side .java files. This includes PolicyTest.java and Bootstrap.java as well as all the .java files that are client stubs for the idl interfaces.

Important: Be very careful with the CLASSPATH. When compiling these files, be sure that only the non-IBM ORB classes are in the CLASSPATH.

Running the example

Now we are ready to try it out. First start up the Policy server. See the *WebSphere Application Server Enterprise Edition Component Broker Planning, Performance and Installation Guide* for a description of how to start a server for the Policy object. Then, at the client, run the following:

```
java PolicyTest
```

Additional tips for non-IBM ORB usage

Specialized homes

PolicyHome is a specialized home, therefore a Component Broker key helper is not needed. Nonspecialized homes require the use of key helper classes, and it is cumbersome to make these classes available to a client using a non-IBM ORB. So it is best to not attempt to use non-IBM ORBs to access managed objects that do not live in specialized homes.

CORBA IIOP

The Component Broker server ORBs use, by default, IIOP level 1.1. If you find that the non-IBM ORB that you are using needs to use IIOP level 1.0, you

must tell Component Broker server ORBs to use IIOP 1.0 instead of their default. If the server processes see the environment settings SOMDGETENV=1 and GENERATE_IIOP10_OBJREFS=1, then they will use IIOP 1.0.

▶ WIN

You can set these values into your user environment before you start the System Manager User Interface.

▶ AIX

You can export these values from the shell where you start the System Manager User Interface.

The ORB daemon (somdd), the name server, and the application server must all be restarted after the environment variables are set.

Trimming client-side dependencies on Component Broker interfaces

The example in this section used the existing Policy sample managed object. The Policy interface uses the `IManagedClient::IManagable` interface, and `IManagable`, in turn, uses interfaces in other CORBA modules, like `CosStream` and `CosLifeCycle`.

If you have the opportunity to influence how the Component Broker managed object is designed, you can remove the dependency on `IManagable` and thus not require that client bindings be created for it and its parents. In the Policy example you could create a Policy interface with no parent. Then make a new interface, say, `ManagedPolicy`, that inherits from `Policy` and `IManagable`. The `ManagedPolicy` is the Component Broker managed object. You can do the same for the Policy specialized home. Create a `PolicyHome` with no parent. Then create a `ManagedPolicyHome` that inherits from `PolicyHome` and `IHome`.

Now you can create client stub bindings for the `Policy` and `PolicyHome` interfaces, as described above, but you do not have to drag over `idl` for `IManagedClient` and its parents.

Chapter 16. Interlanguage object model

Interlanguage object model (IOM) is the language interoperability technology that enables objects written in C++ and objects written in Java to interact cooperatively within a single process. Although the Object Request Broker (ORB) is written in C++, you might find, as a developer, that you prefer to implement some of your distributed objects in other languages such as Java, Smalltalk, or Object Oriented COBOL. IOM is the component that makes local interactions between objects written in different languages simple and efficient.

As a developer, you provide descriptive information about object interfaces to IOM by using the IDLC Command development tool. See *WebSphere Application Server Enterprise Edition Component Broker Programming Reference* for information about the IDLC command development tool.

IOM and Component Broker

Although the Component Broker Server and the Component Broker ORB are written entirely in C++, some customers will either need or prefer to implement some of their distributed objects in other languages, such as Java. IOM provides language interoperability technology to permit objects written in C++ and objects written in Java to interact cooperatively within a single process. The IOM language interoperability technology makes local interactions between objects written in different languages simple and efficient enough to be practical.

The most common strategy employed by ORB vendors to support multiple languages is to offer multiple products – one for each targeted implementation language. From a vendor perspective this strategy works reasonably well when the number of target languages is small, but as the number of languages expands, the effort of maintaining multiple ORB implementations grows proportionally. From a customer perspective, having server products segregated by programming language means that if any language integration issues ever do need to be addressed (whether dealing with legacy products, or new application prototypes), how to accomplish this is an exercise left to the customer. Using an ORB purely as a mechanism for crossing language domains will, in the best case, necessarily result in process switching for each request and response, a solution that has unattractive performance characteristics. Component Broker's IOM technology is designed to avoid these difficulties and provide a cross-language capability that is both efficient and conceptually simple.

The IOM object model provides object interoperability across language domains. The IOM object model is not a programming system; you cannot use it to implement an object. In order to implement an object, you must use an object-oriented programming language. Instead, the IOM object model allows client programs to use objects by invoking the operations they provide without knowing the precise details about how (or in what language) an object has been implemented. With the IOM object model, the form in which a request to invoke an operation on an object is expressed, depends on the requesting program's language (not the language used to implement the object). This means that both object providers and object users have the flexibility to work in a language that is most suited to the problem they are solving.

It is not a coincidence that the object model described here is remarkably similar to the OMG CORBA model for distributed objects. Knowing the essential role that the CORBA standard plays in the Component Broker product, the IOM object model is closely aligned with CORBA.

Not every aspect of a C++ or Java object can be accessed using a mechanism like the IOM object model. If every feature of every supported object language were to be accessible to a user, the most natural way to accomplish this would be to have the client programmer use the same language that the object itself was written in, but this, of course, is not multi-language programming. Consequently, not every object in the world can be fully accessed through the IOM object model. To qualify for full access, objects must be implemented to exploit the capabilities that the IOM object model supports.

However, the IOM object model is very generic. The utility of the IOM object model arises from the fact that for many objects, the specific syntax used to exercise their capabilities is more of an incidental, rather than an essential, property of the object. The essence of the object tends to be more the functionality it offers on the data it holds than a particular artifact of its implementation. As a result nearly all but the most light-weight of objects are expressible as IOM object model objects.

The IOM object model places its emphasis on those aspects of an object that must be visible to its users: the interfaces it exports and the operations they contain. In the IOM object model, the names of the object's interfaces express the type of the object, and the operations they contain define its functionality, or behavior. Types are a way of categorizing the behavior of objects, so that objects with common capabilities can be processed interchangeably, a characteristic known as polymorphism. A new type can also be derived from one or more existing types in the following way:

- Providing a new interface name.
- Defining its specific behavior by introducing operations that express it.

- Indicating the name of one or more existing interfaces (referred to as ancestor or parent interfaces) whose operations it also supports.

Although objects usually contain data, granting users direct access to the data can be problematic. Sometimes the internal consistency of an object can be compromised, or its locality unnecessarily restricted by giving users direct access to its data. A more flexible technique is to introduce a specific set of operations (typically a get operation and sometimes an accompanying set operation) to access the data. Although there is a small performance penalty associated with executing an operation to access an object's data, it also means that the object's implementation has the means to oversee the integrity and consistency of the data, and avoids the unconstrained accesses that would prohibit distributing the object to some other location. Data items accessed through methods are called attributes. The IOM object model provides access to an object's data only in the form of attributes.

Not only is the data in a IOM object model object accessed indirectly through its methods, all IOM object model objects are themselves accessed indirectly, through an object reference. When an operation is invoked on an object, it is an object reference that designates the target object. Client programs obtain new object references as results or outputs from operations performed on object references they already have, or they use a special factory call to obtain an initial object reference. Object references are "use counted" and the underlying objects that they refer to remain accessible until their last object reference is released. IOM object model objects are never allocated or freed; instead their object references are duplicated whenever a new reference is needed, and released when a reference is no longer required. When the last reference is released, the object becomes inaccessible and is presumably discarded, but what actually happens is not visible to a client programmer.

Operations invoked on IOM object model objects may take a combination of zero or more input, in/out, or output arguments and may produce a distinguished value referred to as a result. Both arguments and results are described with a type system that includes scalars, object references, aggregate types, and constructed types. Operations that encounter error conditions may return an out-of-band result referred to as an exception. User-defined exceptions must be declared as a formal part of an object's interface and may contain arbitrary information, limited only by the expressive power of the type system.

IOM, Component Broker's interlanguage interoperability technology, provides an OMG CORBA programming model for brokering object requests within a process. Component Broker's language interoperability technology is also combined with a CORBA ORB to provide the programmer with a consistent programming model for objects, that allows considerable configuration flexibility in placing objects in different computing hosts. Additionally,

because the CORBA programming model is largely independent of the programming language in which it is implemented, Component Broker provides in-process capabilities for communicating across different implementation languages.

Defining IOM interfaces and implementations

IOM interfaces are defined by writing IDL; implementations are written following the guidelines established by the CORBA 2.0 specification.

IOM implementations are either remotable or local only. Remotable implementations are accessed using interfaces defined using standard CORBA IDL. They can be accessed from Java or C++ either remotely across the ORB or locally within a process.

Local only implementations are accessed using interfaces defined using standard CORBA IDL. They can be accessed from Java or C++ locally within a process. They are created using a special creation method that is part of the generated usage bindings for a particular interface that uniquely represents the implementation (the implementation interface). This interface name is related to the implementation name by the name mapping rule that interface `x` has implementation `x_Impl`.

Implementation interfaces for local only implementations are identified to the compiler and emitters by a compilation flag so that streamlined bindings can be emitted. This flag can be specified in the form of an IDL pragma or a command line switch. Mixing local only and remotable interfaces within a single IDL compilation is supported.

Communication between C++ and Java

C++ applications can communicate with Java objects and Java applications can communicate with C++ objects using Interlanguage Object Model (IOM). Additionally this Component Broker feature allows the communication between objects that are not Component Broker business objects. The primary use of IOM is to support Java business objects.

The following sections provide examples for the supported types of cross-language interaction.

- “Scenario: C++ client of a local Java object” on page 425
- “Scenario: Java client of a local C++ object (NT Only)” on page 430
- “Scenario: C++ client of a remote Java object” on page 434
- “Scenario: Java client of a remote C++ object” on page 440

Each example uses the simple “Hello World!” application. The object has a `getMessage()` method that returns the “Hello World!” string. The client calls the `getMessage()` method of the object and prints the result.

The examples assume:

- These examples reside in the following directory tree:

```
example
  HelloWorld
  CppJavaLocal
  JavaCppLocal
  CppJavaRemote
  JavaCppRemote
```

- The `TOPDIR` environment variable is set to point to the example directory.
- The Component Broker server is installed.
- The Windows NT command shell.

Scenario: C++ client of a local Java object

In this example the user of the object, the client, is written in C++ and the object itself is written in Java. The name prefix `CJL` indicates C++ client, Java implementation, Local (that is, in the same process).

The steps in this example are:

1. Create an IDL file that describes the interface
2. Use the `idlc` command to produce the Java implementation files
3. Write Java code to implement the `getMessage()` method
4. Compile the Java files
5. Use the `idlc` command to produce C++ client files
6. Write the C++ client main program
7. Compile and link the client
8. Run the application

Each step is described in greater detail in the following sections.

Create the IDL file

Create the IDL files. From the command line, navigate to the `...\HelloWorld\CppJavaLocal` directory by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal
```

Create a file named `CJL.idl` with the following contents:

```
module CJL {
    interface CJLMessages {
        string getMessage();
    };
};
```

Run the idlc command

Run the idlc command to create the implementation Java file by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal
idlc -e uj:sj CJL.idl
```

This creates the directory %TOPDIR%\HelloWorld\CppJavaLocal\CJL and creates the following files in that directory:

- CJLMessages.java
- CJLMessagesHelper.java
- CJLMessagesHolder.java
- CJLMessagesOperations.java
- _CJLMessagesImplBase.java
- _CJLMessagesStub.java

The idlc command creates the directory that contains these files and gives the directory the same name as the name of the IDL module that contains the interface, that is, CJL. This name is also used for the name of the Java package that contains the Java interface.

CJLMessagesHelper.java provides static methods for managing and interrogating the CJLMessages type; these methods are not used in this example and the CJLMessagesHelper class will be ignored.

CJLMessagesHolder.java aids in streaming objects; this file, also, is not used in this example and will be ignored.

Although the Helper and Holder classes are not used in this example, they are a required part of the CORBA specified Java/IDL mappings and would be used in more complicated examples. _CJLMessagesStub.java is used on the client side and can be ignored for this example.

The three files of interest here are CJLMessages.java, CJLMessagesOperations.java and _CJLMessagesImplBase.java. The file CJLMessages.java contains the Java interface definition:

```
package CJL;
public interface CJLMessages extends CJLMessagesOperations,
    org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
}
```

The file CJLMessagesOperations.java contains the Java interface definition that contains the definition of the method getMessage():

```
package CJL;
public interface CJLMessagesOperations {
    String getMessage();
}
```

The file `_CJLMessagesImplBase.java` contains the base class from which the Java implementation must derive:

```
public abstract class _CJLMessagesImplBase
extends org.omg.CORBA.portable.ObjectImpl
implements CJL.CJLMessages, org.omg.CORBA.portable.InvokeHandler {
    ...
}
```

Write the Java code to implement `getMessage`

Now that you have compiled the IDL file and seen the files this produced, the next step is to write the Java class that implements the `CJLMessages` interface.

From the command line, navigate to the `...\HelloWorld\CppJavaLocal` directory by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal\CJL
```

Create a file named `_CJLMessagesImpl.java` with the following contents:

```
package CJL;

public class _CJLMessagesImpl extends _CJLMessagesImplBase {
    public String getMessage()
    {
        String message = "Hello World! (from local Java implementation)";
        return message;
    }
}
```

There are a few things to note about the implementation:

- The name of the implementation class must be `_CJLMessagesImpl`. IOM assumes that if the IDL interface named `CJLMessages` has a Java implementation then the name of the implementation class will be `_CJLMessagesImpl`.
- Java expects the name of the file that holds the source code for a class to be the same as the name of the class, with the extension `.java` added. Therefore the new `_CJLMessagesImpl` class is in the file named `_CJLMessagesImpl.java`.
- Java expects the name of the package to match the name of the directory that holds the file. Thus, in this case the source for the package `CJL` is in the directory `CJL`.

Note: The source for the package includes both the files emitted from the IDL (as described in “Run the `idlc` command” on page 426) as well as the file created manually by the user as described in this section (Write the Java code to implement `getMessage`).

Compile the Java file

Now you are ready to use the `javac` command to compile the `.java` files by entering:

```
set CLASSPATH=%TOPDIR%\HelloWorld\CppJavaLocal;%CLASSPATH%
cd %TOPDIR%\HelloWorld\CppJavaLocal\CJL
javac *.java
```

This produces several new `.class` files in the `%TOPDIR%\HelloWorld\CppJavaLocal\CJL` directory; one for each of the previously created `.java` files. IOM uses the code in these `.class` files to provide access to our `CJLMessages` object from a C++ client.

This completes the steps needed to create the Java implementation.

Produce the C++ client file

Now you can use the `idlc` command to create the C++ files that the client program will use to access the Java object. The contents of these files are called “client-side bindings”, or, sometimes, just “bindings”. Type:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal
idlc -mlocalonly -ehh:uc CJL.idl
```

The “`-m localonly`” flag is used because, in this example, you know that the client and the implementation object are in the same process, or “local” to each other. The command creates the files `CJL.hh` and `CJL_C.cpp`. The file `CJL.hh` is the header file that declares the `CJLMessages` object. It contains (only the parts of interest to this example are shown, and the contents have been slightly reformatted):

```
class CJLMessages : virtual public CORBA::Object
{
public:
    static const char* CJLMessages_CN;
    static const char* CJLMessages_RID;
    CJLMessages () { }
    virtual CJLMessages () { }
    static CJLMessages_ptr _duplicate(CJLMessages_ptr obj);
    static CJLMessages_ptr _narrow (CORBA::Object_ptr);
    virtual void* _has_ancestor(const char* classname);

#ifdef _MSC_VER
    void* __CJL_CJLMessages__has__ancestor(const char* classname);
#endif // _MSC_VER
    static CJLMessages_ptr _nil ()
    {
        return (CJLMessages_ptr) ((void*)CORBA::Object::_nil());
    }
    virtual void *_SOMThis(const char *& ifname);
    static CJLMessages_ptr _weakProxy(CJLMessages_ptr obj);
    static CJLMessages_ptr _strongProxy(CJLMessages_ptr obj);
```



```

        /* static create method(s)*/
        static CJLMessages_ptr _create();
        virtual char * getMessage()=0;
}; // end nesting scope for interface class ::CJL::CJLMessages

```

For this simple example the only methods you use are `_create()`, to create a new instance of the object, and `getMessage()`, to get the “Hello World” string.

Write the C++ client main program

The next step is to create a C++ client application. There are no constraints on the name of the file that holds the application. This example uses the name `CJLClient.cpp`, and you will put it in the `%TOPDIR%\HelloWorld\CppJavaLocal` directory.

From the command line, navigate to the `%TOPDIR%\HelloWorld\CppJavaLocal` directory by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal
```

Create a file named `CJLClient.cpp` with the following contents:

```

#include <iostream.h>
#include "CJL.hh"

void main()
{
    CJL::CJLMessages_ptr p = CJL::CJLMessages::_create();
    CORBA::String_var message = p->getMessage();
    cout << message << endl;
}

```

Compile and link to the client

The next step is to compile and link the two C++ files, `CJLclient.cpp` and `CJL_c.cpp`. Type:

```

cd %TOPDIR%\HelloWorld\CppJavaLocal
icc /Ti+ /c /Ge+ /DSHASTIFIED CJL_C.cpp
icc /Ti+ /c /Ge+ /DSHASTIFIED CJLClient.cpp
ilink /debug /out:CJLClient.exe CJL_C.obj CJLClient.obj somsh.lib /
    somshcpi.lib somorori.lib

```

Run the application

Now you can run the application by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaLocal
CJLClient
```

The Hello World! (from local Java implementation) message is printed to the console.

Scenario: Java client of a local C++ object (NT Only)

In this example the user of the object, the client, is written in Java and the object itself written in C++. The name prefix JCL indicates Java client, C++ implementation, Local (that is, in the same process).

The basic procedure for this example is:

1. Create an IDL file that describes the interface
2. Use the idlc command to produce implementation C++ files
3. Write C++ code to implement the getMessage() method
4. Compile and link the C++ pieces into a DLL
5. Use the idlc command to produce a Java client Stub
6. Compile the Java client files
7. Write the Java client program and compile it
8. Run the application

Each step is described in greater detail in the following sections.

Create the IDL file

Create the IDL file. From the command line, navigate to the `...\HelloWorld\JavaCppLocal` directory by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppLocal
```

Use an editor to create a file named `JCL.idl` and put into it the following contents:

```
module JCL {
    interface JCLMessages{
        string getMessage();
    };
};
```

Produce the implementation-side C++ binding files

Use the `idlc` command to produce the implementation side C++ binding files by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppLocal
idlc -ehh:ih:uc:sc -mlocalonly -mdllname=JCL JCL.idl
```

This produces the files `JCL.hh`, `JCL_C.cpp`, `JCL.ih` and `JCL_S.cpp`. In this example you are building a C++ object implementation. The `JCL.hh` and `JCL_C.cpp` files are client side files; they must be emitted only because some of the implementation side files include them. `JCL_S.cpp` contains methods used internally by IOM to dispatch requests to the implementation object. The `JCL.ih` file contains the header for the implementation:

```

class JCL_JCLMessages_Impl : public virtual ::JCL::JCLMessages_Skeleton
{
    public: char*  getMessage ();
};

```

The idlc command parameter “-mdllname=JCL” causes the necessary import/export statements to be added to declarations and definitions.

Write the C++ code to implement getMessage

You need to provide an implementation of JCL_JCLMessages_Impl::getMessage(). To do this, run the idlc command to produce the implementation file by entering:

```

cd %TOPDIR%\HelloWorld\JavaCppLocal
idlc -eic -mlocalonly -mdllname=JCL JCL.idl

```

This causes the JCL_I.cpp file to be emitted. This file contains:

```

char* JCL_JCLMessages_Impl::getMessage () { }

```

Use an editor to add the actual implementation to the getMessage() method. So this section of the JCL_I.cpp file becomes:

```

char* JCL_JCLMessages_Impl::getMessage ()
{
    return CORBA::string_dup("Hello World! (From local C++ implementation)");
}

JCL::JCLMessages_ptr JCL::JCLMessages::_create ()
{
    return new JCL_JCLMessages_Impl();
}

```

Note: When the idlc command is told to produce an _I.cpp file (that is, when the “ic” emitter is specified with the -e or -s flag) the idlc command will attempt to produce a new copy of the file. Because this new file would wipe out any previous file that might contain manually introduced implementations, the idlc command will fail if there is a previously existing _I.cpp file. If a new version of the _I.cpp file is needed then you should rename the original, re-emit the new, and then manually re-integrate your implementations.

Compile and link the C++ piece to the DLL

The DLL file name will be JCL.dll. Type:

```

cd %TOPDIR%\HelloWorld\JavaCppLocal
icc /Ti+ /Ge- /c /DSOM_DLL_JCL /DSHASTIFIED JCL_S.cpp
icc /Ti+ /Ge- /c /DSOM_DLL_JCL /DSHASTIFIED JCL_I.cpp
ilib /GENI:JCL.lib JCL_S.obj JCL_I.obj
ilink /DLL /OUT:JCL.dll JCL_S.obj JCL_I.obj JCL.exp somsh.lib \
    somshcpi.lib somorori.lib

```

Produce the client stub

Use the `idlc` command to produce a Java client Stub. Build the Java proxy for the `JCLMessages` object by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppLocal
idlc -euj JCL.idl
```

This creates the directory `%TOPDIR%\HelloWorld\JavaCppLocal\JCL` and creates the following files in that directory:

- `JCLMessages.java`
- `JCLMessagesHelper.java`
- `JCLMessagesHolder.java`
- `JCLMessagesOperations.java`
- `_JCLMessagesStub.java`

The `idlc` command creates the directory that contains these files and gives the directory the same name as the name of the IDL module that contains the interface, that is, “JCL”. This name is also used for the name of the Java package that contains the Java interface.

`JCLMessagesHelper.java` provides static methods for managing and interrogating the `JCLMessages` type; these methods are not used in this example and the `JCLMessagesHelper` class will be ignored.

`JCLMessagesHolder.java` aids in streaming objects; this file, also, is not used in this example and will be ignored. Although the Helper and Holder classes are not used in this example, they are a required part of the CORBA specified Java/IDL mappings and would be used in more complicated examples.

The three files of interest here are `JCLMessages.java`, `JCLMessagesOperations` and `_JCLMessagesStub.java`. The file `JCLMessages.java` contains the Java interface definition:

```
package JCL;
public interface JCLMessages extends CJLMessagesOperations,
    org.omg.CORBA.Object , org.omg.CORBA.portable.IDLEntity {
}
```

The file `JCLMessagesOperations.java` contains the Java interface definition that contains the definition of the method `getMessage()`:

```
package JCL;
public interface CJLMessagesOperations {
    String getMessage();
}
```

The file `_JCLMessagesStub.java` contains the class `_JCLMessagesStub` that provides an implementation of this interface. The implementation does not, however, contain the actual code to produce the message string. Instead, it contains code to connect to the IOM run time. The IOM run time connects to

the C++ implementation that was created earlier, calls `JCL_JCLMessages_Impl::getMessage()`, and returns the result.

Compile the Java files

Now you are ready to use the `javac` command to compile the `.java` files by entering:

```
set CLASSPATH=%TOPDIR%\HelloWorld\JavaCppLocal;%CLASSPATH%
cd %TOPDIR%\HelloWorld\JavaCppLocal\JCL
javac *.java
```

Write and compile the Java client program

Create a main program, in Java, to create the object and call its `getMessage()` method. In the directory `%TOPDIR%\HelloWorld\JavaCppLocal`, use an editor to create a file named `JCLClient.java` and put into it the following contents:

```
import JCL.*;
class JCLClient {
    public static void main(String args[]) {
        System.loadLibrary("JCL");
        try
        {
            JCL.JCLMessages hw = JCL.JCLMessagesHelper._create();
            String m = hw.getMessage();
            System.out.println(m);
        }
        catch ( Exception e)
        {
            System.out.println(
                "JCL._JCLMessagesStub._create() threw an exception");
            System.out.println("\t" + e.toString());
            Runtime.getRuntime().exit(1);
        }
    }
}
```

The line `System.loadLibrary("JCL");` loads the DLL that was built to hold the C++ implementation. Notice the `_JCLMessageStub`'s `_create()` method is used, instead of a Java "new" to create the proxy.

Now compile the `JCLClient` by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppLocal
javac *.java
```

Run the application

Now you can run the application by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppLocal
java JCLClient
```

The Hello World! (From local C++ implementation) message is printed to the console.

Scenario: C++ client of a remote Java object

In this example the user of the object, the client, is written in C++ and the object itself, the implementation, is written in Java, and the client and implementation are in different processes. The Component Broker C++ CORBA Object Request Broker (ORB) is used to provide communication between the client process and the server process where the implementation lives.

The client is written in C++ and the ORB is also implemented in C++. Therefore Interlanguage Object Model (IOM) is not used in the client process.

In the server process there is a Java implementation interacting with a C++ ORB. When the C++ ORB receives a request from a remote client, the ORB knows how to dispatch the request to a C++ implementation.

```
Client --> network --> ORB --> C++ Implementation
```

If, however, the implementation is written in Java, the ORB does not know how to dispatch across the language boundary. To solve this problem, the idlc command can be told to emit a C++ “ORB adapter” that appears to the ORB to be a normal C++ implementation but that knows how to use IOM to dispatch across the language boundary to the real Java implementation.

```
Client --> network --> ORB --> ORB Adapter --> IOM -->Java Implementation
```

There is a limitation in IOM that prevents having a local and a remote implementation of the same interface. This example will have an abstract interface, `CJRAbstractMessage`, and a concrete interface, `CJRConcreteMessage`, that derives from `CJRAbstract`. The implementation for the example will be of `CJRConcreteMessage`. Other uses of the `CJRAbstractMessage` interface can derive their own concrete interface and use IOM to implement it.

Portions of the ORB interfaces are IOM-enabled; so that Java implementations can call their methods, but the BOA interface has not yet been made available to Java programs. Therefore, even though the example `CJRMessages` implementation can be completely written in Java, you must use C++ code to initialize the ORB, instantiate objects, and export IORs. This code resides in the `CJRServer.cpp`.

The basic procedure for this example is:

1. Create the IDL files
2. Run the `idlc` command to produce the Java implementation bindings
3. Write the Java implementation of the `getMessage()` method

4. Compile the Java pieces
5. Create the C++ ORB Adapter
6. Create the Java server code
7. Compile the server code
8. Create the client program
9. Compile and link the client program
10. Run the application

Each step is described in greater detail in the following sections.

Create the IDL files

Create the IDL files. From the command line, navigate to the `...\HelloWorld\CppJavaRemote` directory by entering:

```
cd %TOPDIR%\HelloWorld\CppJavaRemote
```

Create the file `CJRAbstract.idl` and put into it the following contents:

```
module CJRAbstract {
    interface CJRAbstractMessages {
        string getMessage();
    };
};
```

Create `CJRConcrete.idl` and put into it the following contents.

Note: The “`orbadapter`” pragma informs the `idlc` command that this is a Java server side implementation. The emitter emits special C++ bindings capable of dispatching method invocation through IOM run time to the Java implementation. Another way to create an ORB adapter is using the “`—m orbadapter`” modifier to the `idlc` command, when emitting C++ bindings for this IDL file

```
#include "CJRAbstract.idl"
module CJRConcrete {
    interface CJRConcreteMessages : CJRAbstract::CJRAbstractMessages { };
    #pragma meta CJRConcreteMessages orbadapter
};
```

Produce the Java implementation bindings

Run the `idlc` command to produce the Java Implementation bindings by entering.

```
cd %TOPDIR%\HelloWorld\CppJavaRemote
idlc -euj CJRAbstract.idl
```

You need only the usage bindings for the abstract interface. Type:

```
idlc -euj:sj CJRConcrete.idl
```

Write the Java implementation of getMessage()

The next step is to write the Java class that implements the CJLConcrete interface. Type:

```
cd %TOPDIR%\HelloWorld\CppJavaRemote\CJRConcrete
```

Use an editor to create a file named `_CJRConcreteMessagesImpl.java` and put into it the following contents:

```
package CJRConcrete;
class _CJRConcreteMessagesImpl extends _CJRConcreteMessagesImplBase {
    private final String message = "HelloWorld! (
        From remote Java implementation)";
    public String getMessage() {
        return message;
    }
}
```

Compile the Java pieces

Compile the Java pieces by entering:

```
set CLASSPATH=%TOPDIR%\HelloWorld\CppJavaRemote;%CLASSPATH%
cd %TOPDIR%\HelloWorld\CppJavaRemote\CJRConcrete
javac *.java
cd %TOPDIR%\HelloWorld\CppJavaRemote\CJRAbstract
javac *.java
```

Create the C++ ORB adapter

Create two different DLL files: `CJRAbstract.dll` that contains the client-side bindings of the abstract interface and `CJRConcrete.dll` for the concrete implementation. Type:

```
cd %TOPDIR%\HelloWorld\CppJavaRemote
idlc -ehh:uc -mdllname=CJRAbstract CJRAbstract.idl
idlc -ehh:uc:sc -mdllname=CJRConcrete CJRConcrete.idl
icc /Ti+ /c /Ge- /DSOM_DLL_CJRAbstract /DSHASTIFIED CJRAbstract_C.cpp
ilib /geni:CJRAbstract.lib CJRAbstract_C.obj
ilink /DLL /debug /out:CJRAbstract.dll CJRAbstract_C.obj \
    CJRAbstract.exp somshcpi.lib somsh.lib somorori.lib
icc /Ti+ /c /Ge- /DSOM_DLL_CJRConcrete /DSHASTIFIED CJRConcrete_S.cpp
ilib /geni:CJRConcrete.lib CJRConcrete_S.obj
ilink /DLL /debug /out:CJRConcrete.dll CJRConcrete_S.obj \
    CJRConcrete.exp somshcpi.lib somsh.lib somorori.lib CJRAbstract.lib
```

Create the Java server code

Now you can create the Java portion of the server process, that is, the Java code that interacts with the BOA interfaces. Type:

```
cd %TOPDIR%\HelloWorld\CppJavaRemote
```


Use an editor to create a file named CJRServer.java and put into it the following contents:

```
public class CJRServer {
    public static void main(String argv[]){
        try {
            /* Load the appropriate DLLs */
            System.loadLibrary("CJRConcrete");
            System.loadLibrary("somshcpi");
            System.loadLibrary("somshori");

            org.omg.CORBA.ImplementationDef imp =
                org.omg.CORBA.ImplementationDefHelper._create();
            imp.set_protocols("SOMD_TCPIP");
            org.omg.CORBA.ORB op;
            org.omg.CORBA.BOA bp;
            int stat;
            java.io.FileOutputStream fout =
                new java.io.FileOutputStream("OBJREF.OUT");
            op = org.omg.CORBA.ORB.init(argv, null);
            System.out.println("ORB_init has run");

            com.ibm.som.corba.rt.PseudoOrb pso =
                com.ibm.som.corba.rt.DelegateImpl.getPseudoOrb();
            bp = pso.BOA_init (
                new com.ibm.som.corba.rt.PseudoOrbPackage.string_seqHolder(argv),
                "DSOM_BOA");
            System.out.println("BOA_init has run");

            bp.impl_is_ready(imp, false);
            System.out.println("impl_is_ready has run");

            CJRConcrete.CJRConcreteMessages cobj =
                CJRConcrete.CJRConcreteMessagesHelper._create();
            /* This is special code for Java server needed to pin a
               remotable Java implementation so that it is callable by the
               C++ ORB. Use unpinRemotableProxy() below when the instance
               is no longer needed.
            */
            pso.pinRemotableProxy(cobj);

            String asd = op.object_to_string(cobj);
            System.out.println("Object Reference = ");
            System.out.println(asd);

            /* Write asd to file */
            int len = asd.length();
            byte [] bytes = new byte[len];
            for (int i=0; i< len; i++)
                bytes[i] = (byte)asd.charAt(i);
            fout.write(bytes);
            fout.close();

            System.out.println("Server Listening ...");
            stat = bp.execute_request_loop(0);
        }
    }
}
```

```

        bp.deactivate_impl(imp);

        /* Do not need cobj any more */
        pso.unpinRemotableProxy(cobj);
    }
    catch(Exception e) {
        System.out.println("caught exception" + e);
        e.printStackTrace();
    }
}
}
}

```

This example is similar to the provided example in “Scenario: Java client of a remote C++ object” on page 440; you initialize the ORB, create an instance of `CJRConcreteMessages`, write the string version of the IOR to a file, and start the BOA request loop. However, there are two major differences:

- You need to load the appropriate C++ libraries using `System.loadLibrary`. `CJRConcrete.dll` contains the C++ bindings for the ORB adaptor, while `somshori.dll` contains the interlanguage requirements for the C++ run time needed to communicate with the BOA from Java.
- You need to control the lifetime of the C++ ORB adaptor that corresponds to the Java implementation. Controlling the lifetime is done by pinning the C++ ORB adaptor in memory immediately after a remotable object is created. When we are sure that the object can be removed, you unpin its C++ ORB adaptor from memory.

The pinning and unpinning are necessary for all remotable Java objects from Java source code. They are unnecessary if the server is written in C++, because in C++, only a reference can be held (and therefore the Orb adaptor exists).

Compile and link the server

Compile and link the server by entering:

```

cd %TOPDIR%\HelloWorld\CppJavaRemote
javac CJRServer.java

```

Create the client program

You are now ready to work on the client program. First create the `CJRClient.java` file by entering:

```

cd %TOPDIR%\HelloWorld\CppJavaRemote

```

Use an editor to create a file named `CJRClient.cpp` and put into it the following contents:

```

#include "CJRAbstract.hh"
#include <fstream.h>
#include <orb.h>

```

```

#include <stdlib.h>
#include <stdio.h>

char * infile = "OBJREF.OUT";
int main(int argc, char * argv[])
{
    try {
        CORBA::ORB_ptr op;
        op = CORBA::ORB_init(argc, argv, "DSOM");

        // read stringified IOR from file
        ifstream fin(infile);
        char objref[512];
        memset(objref, 512, '\0');
        fin >> objref;

        // Convert the string into a proxy object
        CORBA::Object_var optr = op->string_to_object(objref);

        // Call method on remote object
        CJRAbstract::CJRAbstractMessages_var aobj =
            CJRAbstract::CJRAbstractMessages::_narrow(optr);
        CORBA::String_var message = aobj->getMessage();
        cout << message << endl;

        return 0;
    }
    catch (...)
    {
        cout << "\t an exception was thrown" << endl;
        return -1;
    }
};
}

```

Compile and link the client program

Compile and link the client by entering:

```

cd %TOPDIR%\HelloWorld\CppJavaRemote
icc /Ti+ /c /Ge+ /DSHASTIFIED CJRClient.cpp
ilink /debug /out:CJRClient.exe CJRClient.obj CJRAbstract.lib \
    somsh.lib somshcpi.lib somorori.lib

```

Run the application

Now you can run the application. First start the ORB daemon by entering:

```

cd %TOPDIR%\HelloWorld\CppJavaRemote
start somorbd

```

Wait for the daemon window to appear and display a “location server ready” message. Start the server by entering:

```

start CJRServer

```

Wait for the server window to appear and display a “server listening...” message. Run the application by entering:

```
java CJRClient
```

At this point the message HelloWorld! (From remote Java implementation) displays.

The ORB daemon and the server windows must be manually closed.

Scenario: Java client of a remote C++ object

In this example the user of the object, the client, is written in Java and the object itself is written in C++. The client and the implementation are in different processes. The Component Broker C++ CORBA Object Request Broker (ORB) is used to provide communication between the client and implementation in the server processes.

The implementation object is written in C++ and the server process’s ORB is also C++. Therefore IOM’s language interoperability is not needed in the server process.

Component Broker includes a client ORB written in Java, and one option would be to use this Java Client ORB, together with the Java client, in a homogeneous Java process. This option would not require IOM’s interprocess technology. This example assumes, however, that the client process contains a mixture of C++ and Java code and that the Component Broker C++ ORB is used in the client. The client side of this example explains how to build a Java client that uses the C++ ORB to communicate with a remote CORBA object.

The basic procedure for this example is:

1. Create an IDL file that describes the interface
2. Use the idlc command to produce a set of C++ implementation classes
3. Write C++ code to implement the getMessage() method
4. Write C++ code for the main server program
5. Compile and link the server program that hosts the implementation
6. Compile and link the DLL that contains the C++ portions of the client program
7. Write the Java client program and compile it
8. Run the application

Each step is described in greater detail in the following sections.

Create the IDL file

Create an IDL file that describes the interface. From the command line, navigate to the ...\`HelloWorld\JavaCppRemote` directory by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
```

Use an editor to create a file named `JCR.idl` and put into it the following contents:

```
module JCR {
    interface JCRMessages{
        string getMessage();
    };
};
```

Produce a set of C++ implementation classes

Use the `idlc` command to produce a set of C++ implementation classes by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
idlc -ehh:ih:uc:sc -mdllname=JCR JCR.idl
```

This produces the files `JCR.hh`, `JCR_C.cpp`, `JCR.ih` and `JCR_S.cpp`. In this example you are building a C++ implementation. The `JCR.hh` and `JCR_C.cpp` files are client side files; they must be emitted only because some of the implementation side files include them. `JCR_S.cpp` contains methods used internally by IOM to dispatch requests to the implementation object. The `JCR.ih` file contains the header for the implementation:

```
class JCR_JCRMessages_Impl : public virtual ::JCR::JCRMessages_Skeleton {
public:
    char* getMessage ();
};
```

The `idlc` command parameter “`-mdllname=JCR`” causes the necessary import/export statements to be added to declarations and definitions. If you were only building the C++ server process and linking it from `.obj` files, then this flag would not be necessary in this example. However, the same `JCR.hh` files will be used later to implement the Java client, and in this case a DLL, and the associated import/export statements, will be needed.

Now you need to provide an implementation of `JCR_JCRMessages_Impl::getMessage()`. To do this run the `idlc` command to produce the implementation file by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
idlc -eic JCR.idl
```

This causes the `JCR_I.cpp` file to be emitted. This file contains:

```
char* JCR_JCRMessages_Impl::getMessage () {}
```

Write the C++ code to implement getMessage

Write C++ code to implement the getMessage() method. Use an editor to add the actual implementation to the getMessage() method. This section of the JCR_I.cpp file then becomes:

```
char* JCR_JCRMessages_Impl::getMessage ()
{
    return CORBA::string_dup(
        "Hello World! (From remote C++ implementation)");
}
```

Note: When the idlc command is told to produce an _I.cpp file (that is, when the “ic” emitter is specified with the -e or -s flag) the idlc command will attempt to produce a new copy of the file. Because this new file would wipe out any previous file that might contain manually introduced implementations, the idlc command will fail if there is a previously existing _I.cpp file. If a new version of the _I.cpp file is needed then you should rename the original, re-emit the new, and then manually re-integrate your implementations.

Write the C++ code for the main server program

Write C++ code for the main server program. Now you need a main program that initializes and connects to the ORB, creates an instance of the JCRMessages object, and makes a CORBA IOR available to the client. From the command line, navigate to the ...\\HelloWorld\\JavaCppRemote directory by entering:

```
cd %TOPDIR%\\HelloWorld\\JavaCppRemote
```

Create a file named JCRServer.cpp and use an editor to put into it the following contents:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <corba.h>

#include "JCR.ih"

void main(int argc, char *argv[])
{
    static CORBA::ORB_ptr op;
    static CORBA::BOA_ptr bp;
    CORBA::Status stat;
    ofstream fout("OBJREF.OUT");

    // Initialize the ORB
    CORBA::ImplDef * imp = new CORBA::ImplDef();
    imp->set_protocols("SOMD_TCPIP");
    op = CORBA::ORB_init(argc, argv, "DSOM");
```

```

bp = op->BOA_init(argc, argv, "DSOM_BOA");
bp->impl_is_ready(imp,0);

// Create an Instance of a JCR_JCRMessages_Impl object
JCR_JCRMessages_Impl msgs_Impl;

// Create a "stringified" IOR
char *asd = op->object_to_string(&msgs_Impl);
fout << asd;
fout.close();
CORBA::string_free(asd);

cout << endl;
cout << "server listening..." << endl;
cout.flush();

// Start the BOA request loop.
stat=bp->execute_request_loop(CORBA::BOA::SOMD_WAIT);

bp->deactivate_impl(imp);
CORBA::release(imp);
}

```

In the real world, objects in the server process might be created by factory objects and the client might learn the IORs for the factory objects from a name server. By creating a stringified IOR you avoid implementing factories and dealing with name servers, but you must have a way for the client to learn the IOR for the remote object. The simple solution used in this example is for the server to write the IOR string to a file with the well known name "objref.out". You will soon see the client code open this file and read the IOR string.

Compile and link the C++ pieces to the DLL

Compile and link the server program that hosts the implementation. Now you have all of the pieces needed to build the server process. It is time to compile and link by entering.

```

cd %TOPDIR%\HelloWorld\JavaCppRemote
icc /Ti+ /Ge+ /c JCRServer.cpp
icc /Ti+ /Ge+ /c /DSOM_DLL_JCR /DSHASTIFIED JCR_S.cpp
icc /Ti+ /Ge+ /c /DSOM_DLL_JCR /DSHASTIFIED JCR_I.cpp
ilink /DEBUG /OUT:JCRServer.exe JCRServer.obj JCR_S.obj JCR_I.obj \
      somsh.lib somshcpi.lib somorori.lib

```

Compile and link the DLL

Compile and link the DLL that contains the C++ portions of the client program. Now it is time to build the client.

Working your way back from the implementation toward the client, you see:

- The actual object implementation is in a remote server, and you must deal with an ORB to talk to the remote server.
- The ORB you are using in this example is written in C++, so you must have a C++ entity to talk to the ORB.
- The client is written in Java, so you need to use IOM and its run-time support to allow the Java client to talk with the C++ binding.
- The Java client program deals with a Java stub, that to the client looks like it provides an implementation of the object.

You can build the C++ binding from the JCR_I.cpp and JCR_S.cpp files that were emitted earlier. For use in the client, however, you should compile them for use in a DLL and then link the DLL. Type:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
icc /Ti+ /Ge- /c /DSOM_DLL_JCR /DSHASTIFIED JCR_S.cpp
icc /Ti+ /Ge- /c /DSOM_DLL_JCR /DSHASTIFIED JCR_I.cpp
ilib /GENI:JCR.lib JCR_S.obj JCR_I.obj
ilink /DLL /DEBUG /OUT:JCR.dll JCR_S.obj JCR_I.obj JCR.exp \
      somsh.lib somshcp1.lib somorori.lib
```

In a future step you will load this DLL into the Java client program.

Write and compile the Java client program

Write the Java client program and compile it. You can now prepare the Java portions of the client program. You can build the Java stub by running the idlc command to produce the stub files and then compiling these files. Type:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
idlc -euj JCR.idl
cd %TOPDIR%\HelloWorld\JavaCppRemote\JCR
javac *.java
```

Now you can build the Java main program. From the command line, navigate to the ... \HelloWorld\JavaCppRemote directory by entering:

```
cd %TOPDIR%\HelloWorld\JavaCppRemote
```

Use an editor to create a file named JCRClient.java and put into it the following contents:

```
import org.omg.CORBA.*;
import java.io.*;
import JCR.*;

class JCRClient
{
    public static void main (String argv[])
    {
        try
        {
```



```

System.loadLibrary("JCR");

ORB orb = org.omg.CORBA.ORB.init(argv, null);
BufferedReader fin = new BufferedReader(new InputStreamReader
    (new FileInputStream("OBJREF.OUT")));
String objref = fin.readLine();
fin.close();
org.omg.CORBA.Object obj = orb.string_to_object(objref);

JCR.JCRMessages msgs= JCR.JCRMessagesHelper.narrow(obj);

    System.out.println(msgs.getMessage());
}
catch(Exception e)
{
    System.out.println(e.toString());
}
}
}

```

In a previous step the server process wrote out the IOR to a file named OBJREF.OUT. The following lines read in this IOR and convert it to a usable object reference:

```

ORB orb = org.omg.CORBA.ORB.init(argv, null);
BufferedReader fin = new BufferedReader(new InputStreamReader
    (new FileInputStream("OBJREF.OUT")));
String objref = fin.readLine();
fin.close();
org.omg.CORBA.Object obj = orb.string_to_object(objref);

```

You can now compile the Java main program by entering:

```

cd %TOPDIR%\HelloWorld\JavaCppRemote
javac *.java

```

Run the application

You have now built the example and are ready to run it. You must first start the ORB daemon by entering:

```

cd %TOPDIR%\HelloWorld\JavaCppRemote
start somorbd

```

Wait for the daemon's window to appear and show a "location service ready" message. Start the server process that hosts the implementation by entering:

```

start JCRServer.exe

```

Wait for the server window to appear and show a "server listening ..." message. Run the Java client program by entering:

```

java JCRClient

```

At this point the message Hello World! (From remote C++ implementation) displays.

The ORB daemon and the server windows must be manually closed.

Chapter 17. Workload management



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker. See *OS/390 Component Broker Planning and Installation* for further information on workload management.

Programming model

This section includes the following topics:

- Overview
- Group identity
- Workload manageable objects
- WLM homes
- WLM persistent objects
- WLM transient objects

Overview

Workload management is the discipline of defining, monitoring and actively managing work in your distributed network. In a Component Broker context, “work” is taken to mean the dispatch, routing and receipt of requests between objects in the distributed network and their eventual execution within a Component Broker Application Server. As more clients use an application the amount of ‘work’ increases and the ‘load’ on the servers increases.

The aspect of WLM which is of most interest to application programmers is the workload distribution mechanism. This capability allows the Component Broker run time to dynamically allocate an Application Server to process a request. Clients normally locate resources at a pre-configured server location, perhaps a named server or the server running on their local host. Fixing the relationship between the client and server in this way is a problem for scalability in large enterprises. When multiple servers exist which could potentially service a client’s request, it is not desirable to force the client to go to one fixed server. The aim of workload distribution is to minimize client request response times and maximize server throughput by reducing load imbalance. This can be achieved in part by locating resources at a workgroup scope and allowing the choice of an appropriate, active server to be determined by the ORB.

The key to workload distribution in Component Broker is the use of a server group to define multiple Application Servers with a common configuration. In addition the server group must be configured with a Managing Host which designates one Host to provide management services across the group. This configuration is known as a *controlled server group*.

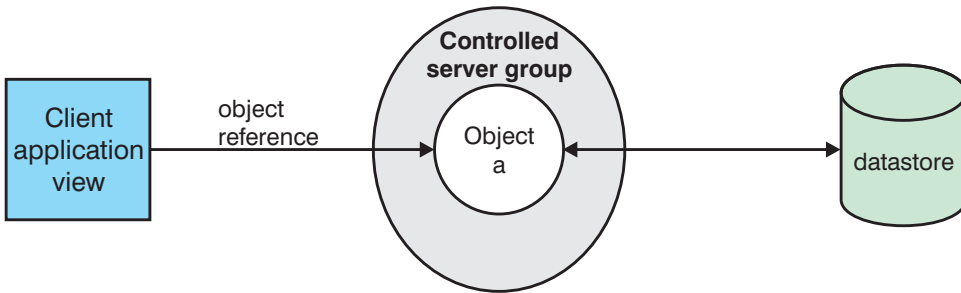
Further information on systems management requirements and recommendations are in the *WebSphere Application Server Enterprise Edition Component Broker System Administration Guide*.

Group identity

In the CORBA programming model an object 'belongs' to the server on which it is created. An object reference uniquely identifies the server (or "implementation") and the object instance which will handle requests. A client obtains the object reference to an object and sends requests to it. In the Component Broker environment, it will be typical to install an application on more than one server, that is by using a server group. Each server in the group can provide the application services required by a client. If the client needs to access a particular business object, then conceptually, it could be programmed to locate all the homes in the network which handle that type of business object. It could then use `findByPrimaryKey()` to retrieve object references to each object instance, for example one instance per server in the group. The client could then choose which of these objects to use to process any subsequent request to that business object. This approach is impractical and very intrusive to the client programming model. What is required is a way to give an object reference a form of "group identity", so that there is only one object reference and the object 'belongs' to the server group, rather than one particular server. This concept is known as the Single Object and Single Server Image property. The ideal is for the client application programmer to write the client application as if there is always one object reference and one server, whereas this may not be the reality.

Support for objects with group identity is provided in the standard Component Broker run time through the management services of controlled server groups and extensions to the object adaptor and application adaptor infrastructure on servers which are members of such groups.

Single object, single server image



Underlying reality

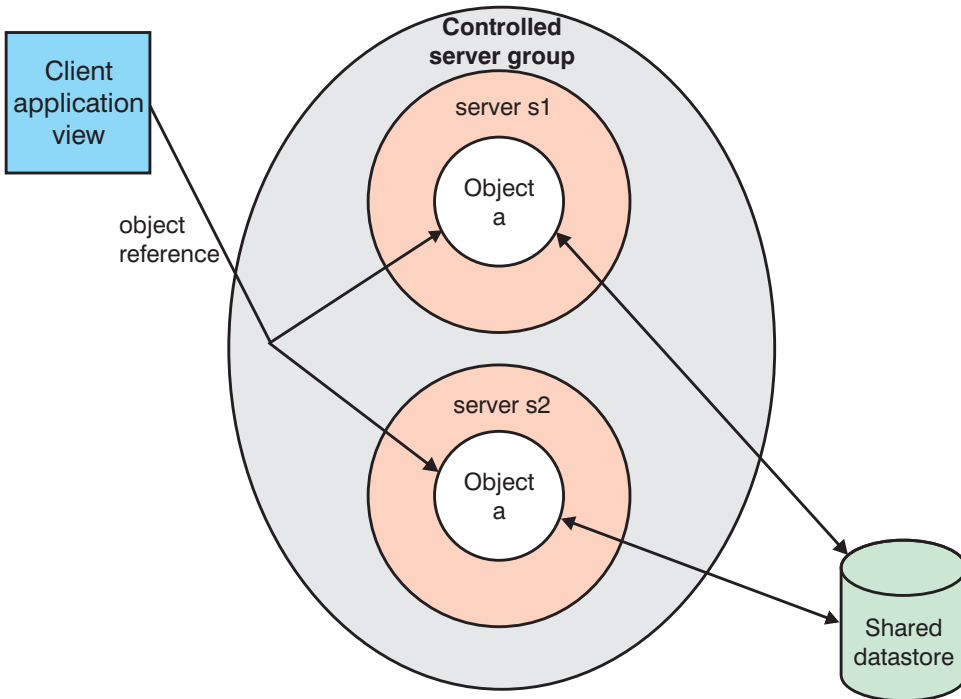


Figure 10. Single object, single server image

Workload manageable objects

Almost any managed object could be given a “group identity” and be subject to workload management. However, you do not use all managed objects in an Application in the same way and there are overheads caused by workload management which should be avoided if there will be little return from the investment.

The opportunity for workload distribution occurs when requests are sent remotely across the ORB from a 'client' to a server; not forgetting that servers can be clients of other servers. Obvious candidates for workload management are therefore homes and any managed object which may be called remotely.

In the design and packaging stages you should identify which objects are workload management candidates, that is, identify *workload manageable objects*. If the application is subsequently installed and configured to run on a controlled server group then these objects will be *workload managed* (WLM) objects.

WLM homes

A home is an excellent example of a workload manageable object as it is often one of the first types of object to be accessed remotely. Many client applications will locate a home for a required business object, perhaps using a factory finder, and then use that home to find or create that business object on the home's server. It is just as desirable to distribute the work involved in finding, creating and processing queries for business objects as it is for the work involved in subsequent operational requests.

WLM persistent objects

When an object instance stores its state persistently in a datastore then that object instance can exist in any server which supports its run time requirements and which can access the datastore. All servers which are members of a controlled server group are required to be equally capable of supporting their configured applications and all their Hosts are required to provide logically symmetrical access to the datastores. In this environment a particular object instance may be activated on any server in the group and, potentially, may be activated on more than one server at the same time. Persistent objects are therefore good WLM candidates.

However it may not always be necessary to workload manage object instances if the home is workload managed. If, for example, an object is located, updated and released within the scope of a transaction then there is minimal benefit in configuring the object as workload managed. It is sufficient that the request to locate the object, which is sent to its home, is subject to workload distribution to tie the workload generated by the transaction to a single server. If this chosen server fails then the transaction must be rolled back and the work restarted as a new transaction. If restarting the work includes refinding the object then a new server will be chosen by the request to the home and there is no need to provide workload distribution for the object itself. Remember that a performance loss is incurred in the client when processing any request to a WLM object and this must be weighed against the potential benefits.

If an object reference is held for a longer period of time then it may be appropriate to workload manage the object itself. The question to ask is whether it ever makes sense to switch workload to a different server when sending a request to that object. When an object is only used within a single transaction this will rarely be the case.

Although not unique to the topic of workload management, you will also need to consider the implications of an object instance existing concurrently in multiple servers. These considerations can influence datastore configuration including locking levels in DB2 and the selection and configuration of the application adaptor, for example the choice between using Embedded SQL or the Cache Service.

WLM transient objects

A transient object is not a natural candidate to be workload managed. The state of a transient object is maintained only in the memory of its owning server process; it is not possible to passivate and then reactivate the object on another server. In most situations, therefore, transient objects will not be workload managed. Using a WLM home in association with transient objects, however, can be a very useful design as described in the “Transient application objects” on page 454 scenario.

It is recommended that transient objects which hold state should exist only for a short time. The longer a transient object remains active holding state, the longer a client is obliged to continue sending work to the same server. Maximizing the opportunity to switch work between servers should be an application design goal to allow the maximum benefit to be derived from future improvements in workload distribution algorithms.

Scenario considerations

WLM objects can be used to good effect to increase the scalability and the availability of an application. However it probably will not come as a complete surprise that there are some restrictions and potential problems in certain application scenarios.

Affinity management

Workload distribution has the unique potential to cause a single object reference to be resolved to multiple target instances over the lifetime of a client proxy. The first time a client sends a request to a WLM object causes a server to be chosen to be the target of that request subject to configured bind policy. On the next request we have to decide whether to use the same server

or choose a different one. What we need to know is whether there is any residual affinity between the client and the first chosen server which dictates that it be reselected.

A good example of affinity is the strong affinity between a client and a server which is established if the client initiates a session or transaction context which will scope a series of method requests to the same objects. When a server is chosen to process the first request in that context it will cause an instance of that object to be activated on that server and for locks to be held by that server for that context. Subsequent requests to that object in the same scope must be dispatched to the same server or else lock conflicts will occur and the application will deadlock. Once the context is committed, or ends for some other reason, then the affinity is broken and the option is available to choose a new server for the next request.

Affinity management is a key technical requirement for workload distribution which is addressed today by the single, default client bind affinity configuration. This behavior sets up an affinity between a client process and the first chosen server which is never broken (unless the server fails or terminates for some reason). This results in a stronger affinity setting than is strictly required for transaction or session management because, even if a context ends and the affinity may be broken, we currently choose to continue sending requests from that client to the first chosen server.

An affinity also exists if an object instance is left activated in the first chosen server. Examples here include the use of transient objects or other objects with an application-managed activation cycle. The client bind affinity configuration works here too.

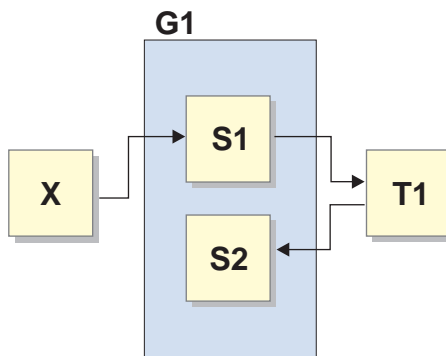
Multiple activation

Although we've just described transaction and session affinity as a client-server affinity it is more accurately an object-server affinity. Once an object instance has been activated within a transaction scope, for example, all subsequent requests to that object in the same scope must be dispatched to that same object instance. The client bind affinity configuration just mentioned ensures that this will be the case for all requests originating from the same client. In a more complex application deployment it is possible for requests to originate from multiple clients.

Consider the case where a client X accesses a WLM object C1 in application A1 on server S1, a member of controlled server group G1. Let's assume that C1 then creates an object P1 in application A2 on server T1. If C1 calls P1, passing an object reference to itself, and then P1 makes a return call to C1 then a problem can occur. We have a potential object flow like X->C1[S1]->P1[T1]->C1[S2]. Note that server T1 had no affinity with a server

in server group G1 and picked server S2 at random, just as X had chosen S1. The result is lock conflict caused by multiple activation of C1 within the same scope. The following figure shows this lock conflict.

Table 20. Lock conflict within the same scope



The workload distribution mechanism can cause multiple activation in certain multi-application topologies. It is an application design requirement to avoid multiple activation in the same scope. This is an application design restriction.

Note that multiple activation can occur in other scenarios not related to workload distribution, for example, in the use of home/key handles.

References to other objects

There are already good reasons to be careful about how references to other objects are managed. For example, imagine the case where an insurance agent uses a client application assigned to server S1 to create a new Policy object P1 and relate it to a new Customer object C1. Let's assume that P1 stores a reference to C1 persistently as a stringified object reference. Some time later another agent may have reason to check this policy and cause an object P1-prime to be activated, this time on server S2. If P1-prime now de-references the stored string form of C1 it will obtain a proxy object which relates back to the original instance of C1 on server S1. It is not normally desirable to increase the inter-server dependencies in this way. Alternative methods of storing object references are provided (that is, home/key handles and foreign key relationships) which avoid this problem.

In the above example, if we assume that Policy objects are workload managed but that Customer objects are not, then the advice to avoid stringified object references should not be ignored. However, it should be noted that references to workload managed objects will always be de-referenced to a local instance whenever possible. Thus, in this example, the use of stringified object references is acceptable if the Customer objects are also workload managed.

Local activation

One of the main principles of workload distribution in Component Broker is that work initiated by a server should always be implemented on that server whenever possible. So whenever a server imports a WLM object reference it will always de-reference to a local instance of that object if it exists. This avoids any network overhead in needlessly sending requests to remote servers which could have been processed locally.

Automatic rebind

When a client initiates a request to a WLM object, the client-side ORB function invites the workload distribution mechanism to select a target server for the request. If the ORB is subsequently unable to dispatch the request to that server the workload distribution mechanism is informed and has the opportunity to choose a different server. This will only occur if the request was not dispatched remotely. If the request was dispatched but a `CORBA::SystemException` is generated the exception will still be reported back to the client application. This includes situations in which the client request timeout is exceeded and when the server terminates unexpectedly while processing a request.

Scenario examples

This section aims to describe some of the common application design patterns and deployment topologies which benefit from the use of WLM.

Transient application objects

One of the simplest WLM scenarios is based on the thin-client application programming model in which the majority of the business logic is encapsulated by an application object (AO). The client application creates a transient application object on a server, sends one or more requests to that application object, then removes it when it is no longer required. An application object is expected to be used only by the client which creates it. The application object interacts with business objects (BOs) in order to process the client's request. The application object may either be "stateless" and process each request independently, or be "stateful" and hold transient state on behalf of the client application between successive requests. The application object and the business objects are packaged into a single application which is configured onto a controlled server group.

The CBToolkit WLM Tutorial Sample is based on this scenario. Use of transient application objects with a WLM home is fully supported and

recommended in applications which follow this pattern. This is illustrated in Figure 11.

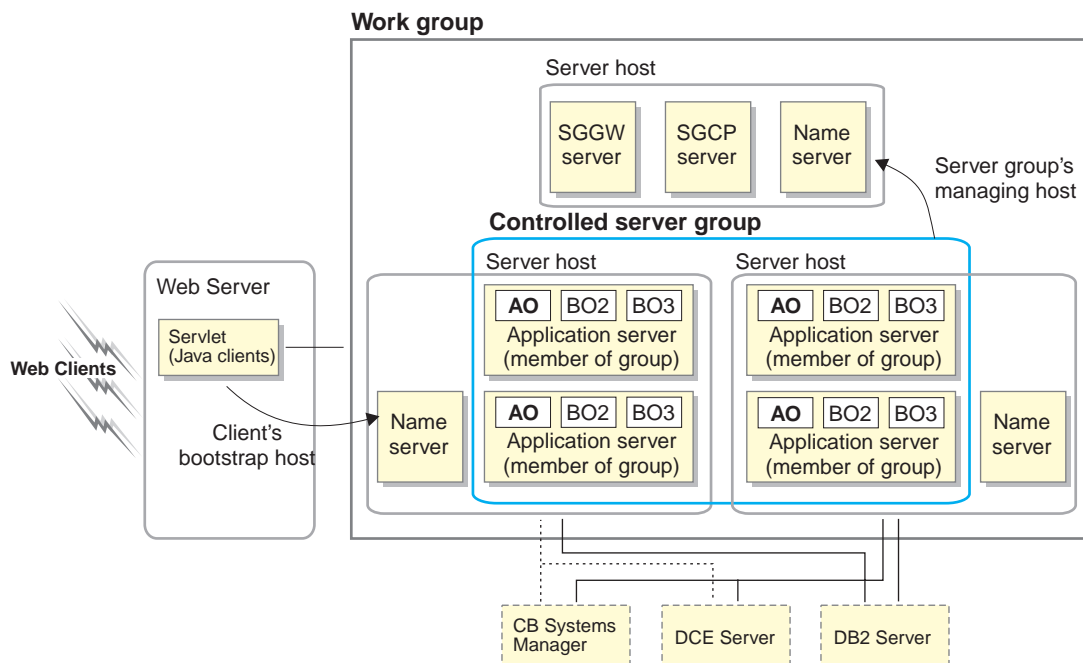


Figure 11. Basic horizontal topology with WLM application objects

Transient objects are usually not workload manageable because the very fact that they are transient ties their lifecycle to a single in-memory activation. This scenario relies only on the application object home being workload manageable. When the application object is created (for example by the default `createFromPrimaryKeyString()` method) the request to the home will be subject to workload distribution with the effect that one of the servers in the group will be chosen on which to create the application object. Subsequent requests to the transient application object will be routed to its owning server.

If the chosen server should become unavailable then this will be reported to the client application in the normal way. The client application may proceed by using the home to create another transient application object. The new application object will be created on one of the other servers in the group which is still available.

The application object has the responsibility to manage transactions on behalf of the application. If each request to the application object is a self-contained piece of work, it may be appropriate to configure the application object to use a container which uses RDB Transaction Service to provide an atomic (per

method) transaction quality of service. In this configuration each request to the application object runs as a transaction in the server. Alternatively the application object may use no Object Services and manage transactions itself, as is the case in the WLM Sample. With this approach it is possible for several successive calls to the application object to contribute to a single transaction. Please see "Application objects" in "The managed object framework" chapter of the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide* for more information.

Business objects with DB2 application adaptor persistence

Business objects are almost always persistent objects which must be activated within some transaction or session context to ensure adequate and correct behavior when accessed concurrently from multiple clients. In this example scenario, assume that the client application manages transaction scope and directly accesses WLM business objects which use the DB2 AA for their persistence. Figure 12 illustrates this.

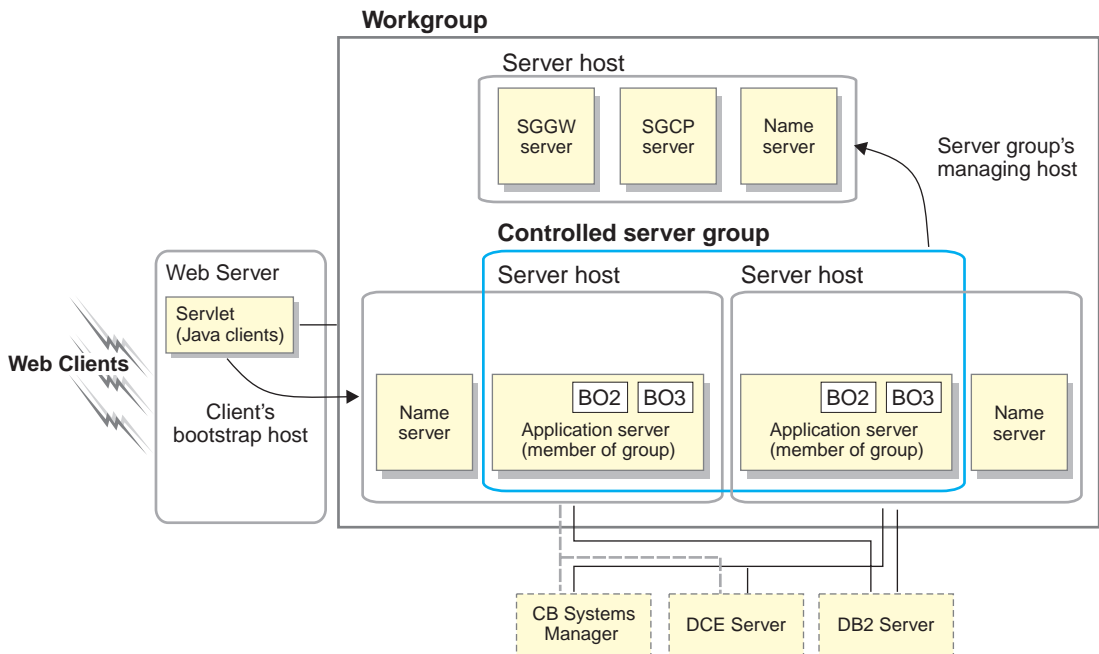


Figure 12. Basic horizontal topology with WLM business objects

The client program locates a business object home in the usual way. A transaction is then begun and the client finds or creates a new business object. All WLM objects have WLM homes therefore the find or create request causes an available server to be chosen at random from the home's server group.

Requests are then sent to the business object for processing until the client requests to commit or, exceptionally, roll back the transaction which it began earlier.

Although the transaction is initiated in the client, the actual transaction coordination is performed on a server. By default the first server to be accessed within the transaction will coordinate the transaction and this will usually be either the application server chosen by the bind policy or, in the case of non-WLM enhanced clients, the Gateway server. This relies on the use of the 'deferred begin' capability in the Transaction Service. Note that in the current release, client applications which cannot use the transaction 'deferred begin' capability require a single server to be designated to act as the transaction coordinator. It is not currently possible to designate a controlled server group in this role. It is recommended that non-WLM enhanced clients which fall into this category should be configured to use the Gateway server. This can be achieved by setting the *factoryFinder* attribute on the **Main** tab of the client style to the absolute name by which the client may resolve the `<servergroup>-Sggw Server-server-scope factory finder`.

If the chosen server fails while a request is pending then this will be reported to the client application in the normal way and any established transaction context must then be terminated, that is, the client application should roll back the transaction. Client affinity with the failed server is broken at this point. The client may then proceed by beginning a new transaction and then either using the home to find or create the business object again or, if the BO exists, just continuing to use its object reference. The next request will cause a new server to be selected and a new client affinity will be established.

If the chosen server fails when no request is pending then the next request will discover that the chosen server is unavailable and the request will then be redispached to a newly chosen server. Client affinity is thus broken and reassigned without concerning the client application. Depending on the state of the transaction it may be possible for the workload to switch seamlessly to the new server if the request that discovers the initial server failure is the very first flow within a new transaction. If transaction context has already been established then the new server will attempt to immediately register itself with the context coordinator. If the failed server was also the transaction coordinator then this registration will fail and the client request will be rejected. In other circumstances registration may succeed and requests may continue to be processed, however the transaction will eventually be rolled back when the coordinator discovers the server failure at commit time.

Depending on the state of the active context and the resources involved, it may not always be possible to retry the same business transaction using another server in the group. It may be necessary for the original server to

restart and perform recovery of the aborted transaction before all locks are released. However, it should be possible to continue to use the application in the general sense.

Application objects using remote business objects with DB2 application adaptor persistence

This scenario is a more complex deployment topology for the Transient Application Object design pattern in which the application is separated into two separately configurable applications within the same application family. The first application consists of the application object with its WLM home and the second application consists of the business objects. If both applications are configured on the same controlled server group then this topology reduces to the simpler scenario presented previously. However the business objects may themselves be configured as workload manageable objects and then the second application can be configured on a separate controlled server group. Figure 13 illustrates this.

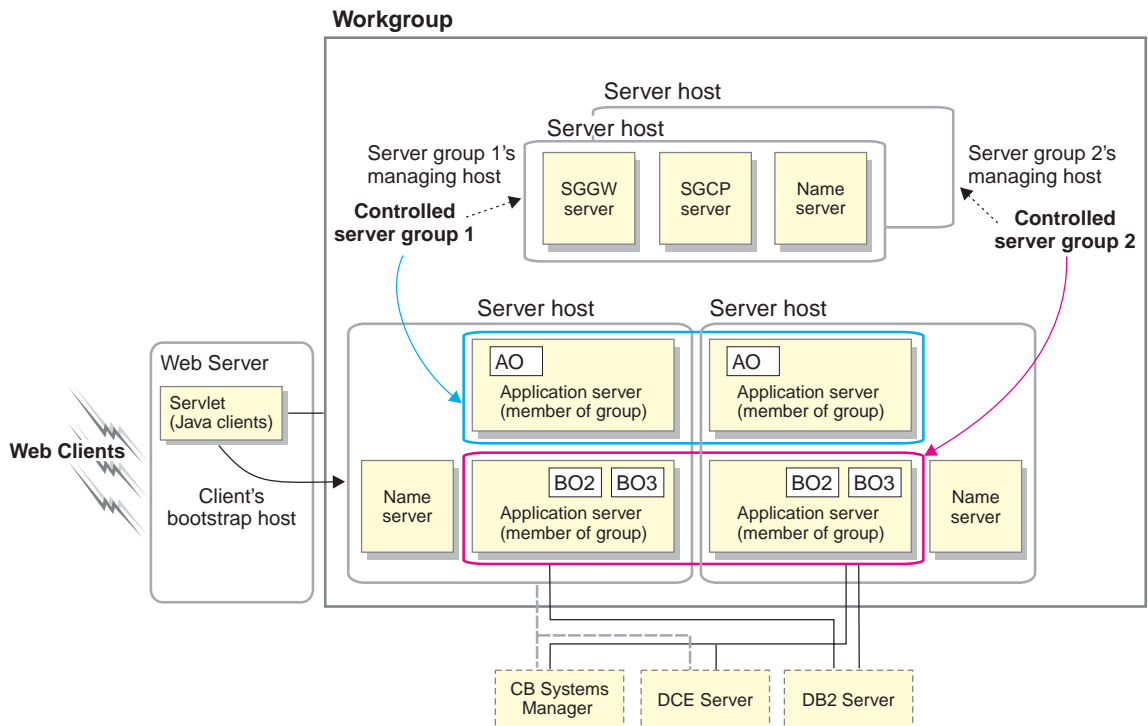


Figure 13. Complex topology with two controlled server groups

If we restrict this scenario to use only business objects which have DB2 application adaptor persistence then the requests between the client and the application object home and application object are exactly as described in

“Transient application objects” on page 454. Requests between the application object and business objects are equivalent to the requests from client to BO described in the “Business objects with DB2 application adaptor persistence” on page 456 scenario.

Other business objects

Use of WLM business objects which do not follow patterns described above, should be considered as technology still under evaluation. Use in production applications is not supported.

Application adaptors

Application adaptors add a quality of service to the managed objects. One of the quality of services is the support of WLM. Supporting WLM causes some side effects to the programming model. These are described in the “An Overview of Application Adaptors” topic in the “Assembling and installing business objects” section of the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*.

BOIM application adaptors

When objects are workload managed, each client has its own copy of an object. Please refer to the “An Overview of BOIM” topic in the “Assembling and installing business objects” section of the *WebSphere Application Server Enterprise Edition Component Broker Programming Guide*, which discusses the impact of application adaptors when developing business objects, and the issues involved with having multiple copies of the same object.

Using Object Builder

The discussions in the previous sections have been intended to provide information about workload management considerations, specifically workload distribution, relevant to your application design. This section provides additional information to help you implement and configure Component Broker to support your application. There are two steps:

1. Define WLM containers which are configured to contain WLM objects.
2. Designate WLM managed objects as you package them into an Application.

Adding a container

A container is an application adaptor’s unit of configuration management. For each unique set of configuration criteria which your application requires you

will have to define a separate container. One of these configuration criteria concerns workload management. When a reference to a managed object is exported from a Component Broker Application Server, the application adaptor refers to the object's container to determine whether the object is a candidate for workload management. When you define a new container using Object Builder you designate that the container will contain workload managed objects using the **Workload managing** checkbox. The application adaptor will then know at run time to export additional metadata.

In the Application Family definition which Object Builder creates, each container which has been configured to contain WLM objects will have an associated Policy Group object. A Policy Group is a Systems Management object which feeds configuration information to the workload distribution mechanism at run time. Because Component Broker currently supports only one Policy Group configuration there is no provision in Object Builder to modify the Policy Group settings.

Adding a managed object to an application

One of the final steps in creating an application package with Object Builder is the packaging of the managed objects and other ancillary files into applications and client applications within an application family. It is at this time that you must identify the workload manageable objects in your application either by configuring them into WLM containers, which will denote a WLM object with its WLM home, or just configuring them with a WLM home.

When you select to *Add a Managed Object* to an application you are presented with the Configure Managed Object wizard. If you select the **Workload managing** checkbox on the container page then only WLM containers, which also match the other MO characteristics, will be presented for selection. You should make sure that a suitable WLM container has already been added to the model or also select the **Create a new container for this managed object** checkbox.

On the home page you must select either a default home implementation or a specialized home implementation. If you have configured the MO with a WLM container then you must also choose a WLM home. The choice of a default home will be restricted to the one which is applicable. If you select a specialized home then the specialized home MO class itself must be a WLM object. Note that when you add a specialized home MO to an application and configure it with a WLM container the choice of the correct WLM default home is made automatically and is not selectable on the home page. It is always possible to select a WLM home for a MO, even if a MO is not itself configured with a WLM container. This allows support for workload distribution in most application designs.

If an application containing workload manageable objects is subsequently configured on a controlled server group then the Component Broker run time will identify WLM MOs and provide object references with a group identity. Method requests to instances of the MO class will be subject to workload distribution.

Client programming model

This section includes the following topics:

- Using factory finders
- Exceptions and recovery

Using factory finders

The Component Broker client programming model depends to a large extent on the factory finding capabilities of the LifeCycle Service. Almost all applications will use a factory finder to locate the homes of their managed objects. When an application is first loaded by a server, each home has the opportunity to register with the LifeCycle Service. This registration effectively records the existence of each home as a resource in the distributed Name Space in such a way that a factory finder, given a suitable "Location" in which to look, can search for an object by its specified properties.

The introduction of objects with "group identity" causes some disruption to the basic LifeCycle registration process. Homes are normally registered as server and host-scope resources, that is, resources belonging to the server where they exist and to the host on which that server is configured. Individual homes can optionally be registered as workgroup or cell resources and all the homes on a server can also be made visible at the workgroup or cell scope. The difference with WLM homes is that the home is now a server group resource, rather than a server resource. The server group is also not restricted to just one host. WLM homes are therefore resources of the server group and the common workgroup preferred by all participating hosts. They may optionally be individually registered at the cell level and the server group may be made visible at the cell level.

When first learning to use factory finders a commonly cited example is to use the system-provided host-scope factory finder which can be resolved from the Name Space with the name `/host/resources/factory-finders/host-scope`. This factory finder will only look for the requested home as a resource on the local host. It will therefore not be able to find any WLM home. As your first application is unlikely to feature WLM objects, this is probably not a problem. However it is something you need to take into account as you add WLM capabilities to an application.

Assuming you only have one server group per workgroup configured for a particular application, then client applications could use the `/workgroup/resources/factory-finders/workgroup-scope` factory finder. This assumes that all the clients will run on, or be bootstrapped by, a host which prefers the same workgroup as the hosts participating in the server group configuration. This is a reasonable assumption as it is necessary for client-initiated workload distribution anyway. Note that this factory finder will not find non-WLM homes unless they have been configured to be visible at the workgroup level.

For server-side code which needs to use a factory finder to locate homes within the same application, use of the default `workgroup-scope` factory finder is not recommended. If the application is not installed on a controlled server group then the homes will not be WLM. If they are made visible at the workgroup level, a general `workgroup-scope` search is not guaranteed to find the instance on the same server. This same problem can arise using `host-scope` if more than one server supporting the same application is installed on the same host. In the workgroup scenario the coincidence of multiple servers is expected to be much higher.

The recommended approach for server-side code is to use the system-provided multi-location factory finder bound at `/host/resources/factory-finders/<server name>-server-scope-widened`. This will search for the registered home as a resource of both the local server and, when the server is a member of a controlled server group, the appropriate server group.

For more information on use of factory finders refer to “Concepts of factory finders” on page 114.

Exceptions and recovery

In a CORBA environment there are many opportunities for the system to generate exception conditions which are reported back to the originating application code. An application must adopt a rigorous programming discipline to catch, handle, report and recover from these exceptions.

The workload distribution mechanism can often reduce the probability of exceptions being seen by the client application, because it is able to reroute requests to an available server and preserve the single server image property. Where requests are dispatched remotely but subsequently some failure occurs, the client application will still receive appropriate exceptions.

Note that the `CORBA::NO_IMPLEMENT` exception is normally generated by the ORB to signify that the server targeted by a request is not available or is no longer defined. In a controlled server group environment this exception

will be generated if the Server Group Control Point is unavailable and therefore the client cannot determine which servers in the group are available. It can also be caused by the fact that no servers have registered with the SGCP (that is, no servers are available) or that an attempt has been made to contact each available server without success.

Chapter 18. Interface repository



The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker.

This section contains procedures for the interface repository (IR):

- Using the configuration tool
- Configuring ODBC for AIX
- Building an interface repository database
- Displaying the contents

Using the configuration tool



It is recommended that ODBC be configured by Component Broker installation using the configuration tool. The following procedure is used if the IR database needs to be recreated or reconfigured.

Creating the IR database in DB2


To create the CBSORBIR database, go to an DB2 prompt and type:

```
create database CBSORBIR
connect to CBSORBIR
create table IRREPDAT (
    IID CHAR(32), REPID VARCHAR(2048),
    NAME CHAR(254), KIND CHAR(12), STREAM LONG VARCHAR)
create index REPINDEX ON IRREPDAT(IID)
create table IRREFER (
    KEYFIELD CHAR(64), IID CHAR(32),
    REFERTOIID CHAR(32))
create index REFINDEX ON IRREFER(KEYFIELD)
create table IRINHER (
    KEYFIELD CHAR(64), IID CHAR(32),
    PARENTIID CHAR(32))
create index INHINDEX ON IRINHER(KEYFIELD)
create table IRBUCKET (
    KEYFIELD CHAR(64), BUCKETIID CHAR(32),
    IID CHAR(32))
create index BKTINDEX ON IRBUCKET(KEYFIELD)
create table IRWORK ( IID CHAR(32))
create index WINDEX ON IRWORK(IID)
CATALOG SYSTEM ODBC DATA SOURCE CBSORBIR
CONNECT RESET
```

Notes about the DB2 commands:

1. Each create table and index must be entered on a single line.
2. Alternatively, these commands can be put in a DB2 script file and invoked from a DB2 command line environment using the DB2 -f <script_file>.

Configuring ODBC for AIX

 It is recommended that ODBC be configured by Component Broker installation using the configuration tool. However if you move your ODBC data files or drivers, you can use these instructions to reconfigure.

1. To create the CBSORBIR database, go to an DB2 prompt and type:

```
create database CBSORBIR
connect to CBSORBIR
create table IRREPDAT (
    IID CHAR(32), REPID VARCHAR(2048),
    NAME CHAR(254), KIND CHAR(12),STREAM LONG VARCHAR)
create index REPINDEX ON IRREPDAT(IID)
create table IRREFER (
    KEYFIELD CHAR(64), IID CHAR(32),
    REFERTOIID CHAR(32))
create index REFINDEX ON IRREFER(KEYFIELD)
create table IRINHER (
    KEYFIELD CHAR(64), IID CHAR(32),
    PARENTIID CHAR(32))
create index INHINDEX ON IRINHER(KEYFIELD)
create table IRBUCKET (
    KEYFIELD CHAR(64), BUCKETIID CHAR(32),
    IID CHAR(32))
create index BKTINDEX ON IRBUCKET(KEYFIELD)
create table IRWORK ( IID CHAR(32))
create index WINDEX ON IRWORK(IID)

CONNECT RESET
```

Notes about the DB2 commands:

- a. Each create table and index must be entered on a single line.
 - b. Alternatively, these commands can be put in a DB2 script file and invoked from a DB2 command line environment using the DB2 -f <script_file>.
2. Determine if you have an .odbc.ini configuration file that is already installed on your system.

The ODBC configuration file is located in your \$HOME directory. Change to your \$HOME directory and check to see if the .odbc.ini file exists. Type:

```
ls -a
```

- If the \$HOME/.odbc.ini file does exist, you will need to add the contents of the etc/odbc.ini file to the \$HOME/.odbc.ini file.

- If the \$HOME/.odbc.ini file does not exist, copy the odbc.ini file that is found in the etc directory to the \$HOME/.odbc.ini file.
3. Edit the \$HOME/.odbc.ini file.
 - a. Change the Driver line to be the location of the db2.o share library. The db2.o file is located in the lib directory on the sqllib image.
 - b. Change the InstallDir line to be the location of the sqllib/odblib directory.

The .odbc.ini file looks like this:

```
.odbc.ini sample =====

[ODBC Data Sources]
CBSORBIR=IBM DB2 ODBC DRIVER
[CBSORBIR]
Driver=/home/inst1/sqllib/lib/db2.o      <-Change this line
Description=IBM Interface Repository
InstallDir=/home/inst1/sqllib/odblib    <-Change this line
```

Note: Where /home/inst1 is the location the DB2 instance used by Component Broker.

4. Determine if you have an .odbcinst.ini configuration file that is already installed on your system.

The ODBC configuration file is located in your \$HOME directory. Change to your \$HOME directory and check to see if the .odbcinst.ini file exists.

Type:

```
ls -a
```

- If the \$HOME/.odbcinst.ini file does exist, you will need to add the contents of the etc/odbcinst.ini file to the \$HOME/.odbcinst.ini file.
 - If the \$HOME/.odbcinst.ini file does not exist, copy the odbcinst.ini file that is found in the etc directory to the \$HOME/.odbcinst.ini file.
5. Edit the \$HOME/.odbcinst.ini file.

Change the Driver line to be the location of the db2.o share library. The db2.o file is located in the lib directory on the sqllib image.

The .odbcinst.ini file looks like this:

```
.odbcinst.ini sample =====

[IBM DB2 ODBC DRIVER]
Driver=/home/inst1/sqllib/lib/db2.o      <-Change this line
```

Note: Where /home/inst1 is the location the DB2 instance used by Component Broker.

Building an interface repository database

The idlc IR emitter can be used to create information in the interface repository database that is representative of an Interface Definition Language (IDL) file. The information in the interface repository database can be accessed at run time by an application using the interface repository framework API. Typical use of the interface repository in a run time environment includes retrieving interface related information for use with the dynamic invocation interface (DII) or with workload management.

The idlc IR emitter will emit code which, when compiled and linked, can be run to populate the interface repository.

The following steps show how to use the idlc IR emitter to create information in the IR database:

1. Run the idlc IR emitter to generate the source code to create objects in the interface repository database:

```
idlc -eir test.idl
```

This creates a source file named test_IR.cpp that contains logic that creates the appropriate objects in the interface repository database.

2. After compiling and linking the generated source code, run the resulting application with the following command:

```
test_IR
```

Populating the IR with Component Broker's definitions

Populate the IR with Component Broker's definitions. If the IR database has been recreated without using the configuration tool, you should populate it using this procedure.

From a command prompt, type:

```
irpop1  
irpop2
```

These two programs will populate the IR with Component Broker's IDL definitions.

Displaying the contents

The irdump provides a means to access the contents of the IR from a command line program. The IRBrowser provides a rich user interface, but cannot be run from a script or command line in an automated fashion.

The `irdump` tool is invoked using by typing `irdump > irdump.out` from a command prompt.

This section contains:

- A simple IDL file called `Test.idl`
- The makefile used to create the executable
- The command to run the makefile
- The command to run the executable
- The command to invoke `irdump` with its output

The `Test.idl` file

The example that follows shows a simple IDL file `Test.idl`.

```
module sample_module {
    interface sample_interface
    {
    }; /* end module */
```

The makefile

The following is a simple makefile that generates source code to create objects in the IR and compile the source.

```
#-----
The targets we need to build
#-----
SRC_FILES = \
    test_IR.cpp
IR_OBJ_FILES = \
    test_IR.obj
EXE_FILES = \
    test_IR.exe

MAP_FILES = $(EXE_FILES:.exe=.map)

#-----
The CBServer system libraries we need to link our DLL
#-----
IR_EXE_LIBS = \
    somorori.lib

#-----
# Pseudo targets to run as part of an "nmake clean"
#-----

CLEAN_TARGETS = \

#-----
# After setting our variables but before defining our recipes
# we include "obdll.mk" which contains variables and recipes
# common to makefiles generated by the CBToolkit Object Builder
#-----
```

```

!include $(IVB_DRIVER_PATH) \bin\obd11.mk

#-----
# Recipes for the interface files
#-----

test_IR.cpp: test.idl
    $(IDL_) $(IDL_IR_FLAGS) test.idl

test_IR.obj: test_IR.cpp

test_IR.exe: test_IR.obj
    $(LDEXE_) $(LD_EXE_FLAGS) /OUT:$@ /MAP:$*.map test_IR.obj
$(IR_EXE_LIBS)

clean_test:
    $(REMOVE_) test_IR.exe
    $(REMOVE_) test_IR.obj
    $(REMOVE_) test_IR.cpp

#-----
# End
#-----

```

Running the makefile

To run the makefile, type:

```
c:\irtest->make -f test.mak
```

Running the executable

To run the executable you just built, type:

```
c:\irtest->test_IR.exe
```

Running irdump

Run irdump to show the IR contents, type:

```
c:\irtest->irdump
```

The following code is the output from the irdump command.

```

=====
InterfaceDef: ::Object
=====
RepositoryID:      IDL:Object:1.0
Defined in:        The Repository
Version:           1.0

Interface is empty!
=====
InterfaceDef: ::sample_module::sample_interface
=====
RepositoryID: IDL:sample_module/sample_interface

```

Defined in: sample_module
Version: 1.0

Interface is empty!

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licenses of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

For Component Broker:

IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

For TXSeries:

IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available

sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

AFS	IMS
AIX	MQSeries
AS/400	MVS/ESA
CICS	OS/2
CICS OS/2	OS/390
CICS/400	OS/400
CICS/6000	PowerPC
CICS/ESA	RISC System/6000
CICS/MVS	RS/6000
CICS/VSE	S/390
CICSplex	Transarc
DB2	TXSeries
DCE Encina Lightweight Client	VSE/ESA
DFS	VTAM
Encina	VisualAge
IBM	WebSphere

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle and Oracle8 are registered trademarks of the Oracle Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

OSF and Open Software Foundation are registered trademarks of the Open Software Foundation, Inc.

* HP-UX is a Hewlett-Packard branded product. HP, Hewlett-Packard, and HP-UX are registered trademarks of Hewlett-Packard Company.

Orbix is a registered trademark and OrbixWeb is a trademark of IONA Technologies Ltd.

Sun, SunLink, Solaris, SunOS, Java, all Java-based trademarks and logos, NFS, and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Some of this documentation is based on material from Object Management Group bearing the following copyright notices:

Copyright 1995, 1996 AT&T/NCR
Copyright 1995, 1996 BNR Europe Ltd.
Copyright 1991, 1992, 1995, 1996 by Digital Equipment Corporation
Copyright 1996 Gradient Technologies, Inc.
Copyright 1995, 1996 Groupe Bull
Copyright 1995, 1996 Expersoft Corporation
Copyright 1996 FUJITSU LIMITED
Copyright 1996 Genesis Development Corporation
Copyright 1989, 1990, 1991, 1992, 1995, 1996 by Hewlett-Packard Company
Copyright 1991, 1992, 1995, 1996 by HyperDesk Corporation
Copyright 1995, 1996 IBM Corporation
Copyright 1995, 1996 ICL, plc
Copyright 1995, 1996 Ing. C. Olivetti &C.Sp
Copyright 1997 International Computers Limited
Copyright 1995, 1996 IONA Technologies, Ltd.
Copyright 1995, 1996 Itasca Systems, Inc.
Copyright 1991, 1992, 1995, 1996 by NCR Corporation
Copyright 1997 Netscape Communications Corporation
Copyright 1997 Northern Telecom Limited
Copyright 1995, 1996 Novell USG
Copyright 1995, 1996 02 Technolgies
Copyright 1991, 1992, 1995, 1996 by Object Design, Inc.
Copyright 1991, 1992, 1995, 1996 Object Management Group, Inc.
Copyright 1995, 1996 Objectivity, Inc.
Copyright 1995, 1996 Oracle Corporation
Copyright 1995, 1996 Persistence Software

Copyright 1995, 1996 Servio, Corp.
Copyright 1996 Siemens Nixdorf Informationssysteme AG
Copyright 1991, 1992, 1995, 1996 by Sun Microsystems, Inc.
Copyright 1995, 1996 SunSoft, Inc.
Copyright 1996 Sybase, Inc.
Copyright 1996 Taligent, Inc.
Copyright 1995, 1996 Tandem Computers, Inc.
Copyright 1995, 1996 Teknekron Software Systems, Inc.
Copyright 1995, 1996 Tivoli Systems, Inc.
Copyright 1995, 1996 Transarc Corporation
Copyright 1995, 1996 Versant Object Technology Corporation
Copyright 1997 Visigenic Software, Inc.
Copyright 1996 Visual Edge Software, Ltd.

Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.



This software contains RSA encryption code.

Other company, product, and service names may be trademarks or service marks of others.

Index

Special Characters

- *any 128
- _create 163
- *ignore 129
- *local 129
- *localname 129
- *servergroupname 128
- *servername 128

A

- absolute and relative names 208
- access control 229
- access the TransactionCurrent object 270
- access to CICS applications 294
- access to IMS applications 293
- acquiring a credential on a thread 227
- activation, local 454
- activation, multiple 452
- affinity management 451
- aNewNamingContext (variable) 202
- ao (application object) 454
- application, design a Transaction Service 266
- application, manage transactions in your
 - access the TransactionCurrent object 270
 - set a time limit for all new transactions 271
 - start a transaction using the TransactionCurrent interface 272
 - suspend a transaction from the current thread 273
- application naming context 123
- application objects 454
- architecture and design
 - Transaction Service
 - application, 262
 - non-recoverable client 263
 - recoverable server 265
 - transactional server 265
- array, queries that result in a data 328, 377

- associations, principals, credentials, and secure
 - principal 222
- asynchronous events, communicating
 - events
 - channels 24, 52
 - consumers 24, 52
 - structured 52
 - suppliers 24, 52
 - topics 24, 52
 - filters 52
- attributes, standard syntax
 - model 214
- attributes of a credential, acquiring the security 224
- authentication and end-users 231
- authenticity, using environment variables to establish 236
- automatic rebind 454

B

- beneficiary object 113
- bind_new_context() 201
- bind_new_context_with_string() 201
- binding an object with a name
 - bind() 196
 - bind_with_string() 196
 - CosNaming::AlreadyBound exception 196
 - Naming Service 197
 - ORB::
 - resolve_initial_references 197
 - resolve() 197
 - target naming context (variable) 196
 - resolve() 196
- BOIM application adaptors 459
- bootstrap host 207
- bootstrap host, local root naming context and
 - bootstrap host 207
 - client machines 188
 - resolve_initial_references() 207
- Builder, using Object
 - adding a container 459
 - managed object, adding a 460
- business objects 459

C

- C++ and Java, communication between 424
- C++ client of a local Java object
 - compile and link to client 429
 - compile Java file 428
 - create IDL file 425
 - implement getMessage 427
 - produce C++ client file 428
 - run application 429
 - run idlc command 426
 - write C++ client main program 429
- C++ client of a remote Java object 434
 - C++ ORB adapter 436
 - compile and link client program 439
 - compile and link the server 438
 - compile Java pieces 436
 - create client program 438
 - create IDL files 435
 - implementation of
 - getMessage() 436
 - Java implementation bindings 435
 - Java server code 436
 - run application 439
- C++ object, local (nt only)
 - Java client 430
- C++ object (remote), Java client of
 - C++ implementation classes 441
 - compile and link DLL 443
 - create IDL file 441
 - implement getMessage 442
 - Java client program 444
 - link the C++ pieces to DLL 443
 - main server program 442
 - run application 445
- Cache Service
 - configuration attributes 393
 - DB2 limits 396
 - Oracle locking 397
- Cache Service and DB2 limits 396
- cell name tree 187, 190
- changePrice() 46
- channels, event
 - configuring 41, 73
 - connecting 42, 74

- channels, event (*continued*)
 - creating 39, 71
 - disconnecting 76
 - event channel samples
 - changePrice() 46
 - constructor 47
 - disconnect
 - _push_consumer() 49
 - disconnect_push_supplier() 46
 - ec 47
 - echome 47
 - FilterFactory and filters 77
 - locating 38, 70
 - schemas
 - multiple 29, 60
 - single 29, 59
- checkpoint and reset session context
 - example of checkpointing and resetting 302
- CICS applications, access to 294
- client, non-recoverable 263
- client machines 188, 207
- client programming model
 - exceptions and recovery 462
 - factory finders, using 461
- client programming
 - model,common 294
- client use of the Session Service 290
- clients and Java BO example, Java 385
- code-set conversion for
 - remote method invocations 400
- collection, queries that result in an object 328, 376
- collections
 - queryable collection 364
 - result collection 364
- collections, optimizations for object 106
- collections, queries on queryable
 - query over reference collections 322, 368
- collections, queries over unnamed
 - example using the query evaluator interface, an 378
- collections, query over reference
 - createCollectionFor() 322, 368
 - DB2 search engine 352, 369
- collections, topology of query evaluators and
 - evaluate_to_data_array() 325, 374
 - evaluate_to_iterator() 325, 374
- commit process, the one-phase 258
- commit process, the two-phase
 - heuristic damage 258
 - heuristic mixed outcome 258
- common client programming
 - model 294
- communication between C++ and Java 424
- communication models 25, 54
- Component Broker, IOM and 421
- concepts of factories
 - application factory 114
 - beneficiary object 113
 - managed object factories 113
 - Policy object 113
 - PolicyHolder object 113
- Concurrency Service, purpose of a
 - resource access
 - lock mode 3
 - read lock 3
 - write lock 3
 - simultaneous updates (table) 2
- Concurrency Service in a transactional environment 4
- Concurrency Service tasks
 - change mode of lock 17
 - complete top level transactions 14
 - configure run-time support 21
 - CosNaming::AlreadyBound 196
 - creating a lock set 10
 - define a lock set 10
 - handle exceptions 20
 - managing objects 20
 - non-transactional locks, use 15
 - obtain and release locks from a lock set 12
 - preventing deadlocks 18
 - related lock sets 4
 - relating lock sets 11
 - release locks in a transactional framework 13
 - troubleshooting 21
- concurrency supports locking 3
- conditions required for queries 351, 390
- configuring a server to use the Transaction Service 285
- conflict resolution 1
- conflicting locks in a lock set 6
- considerations, programming
 - granularity 8
 - manage objects 8
 - resources, shared 8
- considerations for the server, other security
 - AIX default locations 230
 - WIN default locations 230
- constant_random_id attribute 106
- consumers 23
- consumers, event 33
- container, adding 459
- contents of a naming context, listing the
 - CosNaming::NamingContext::list() 203
 - list_with_string() 204
 - network latency 203
 - next_n() 204
 - next_one() 204
- contents of the IR, displaying 468
- context, transaction scope and
 - explicit propagation 254
 - implicit propagation 254
 - scope 254
 - transaction context 253
- context and bootstrap host, local root naming
 - bootstrap host 207
 - client machines 188
 - resolve_initial_references() 207
- context to another thread, pass a transaction 275
- control, access 229
- conversion of objects to string form 399
- converting between name-string and name-structure
 - aNameString (variable) 218
- CORBA standard exceptions
 - CORBA::INITIALIZE 273
 - CORBA::INVALID_TRANSACTION 273
 - CORBA::PERSIST_STORE 273
- CosConcurrency::LockCoordinator
 - drop_locks() 4, 11
- CosNaming::AlreadyBound
 - exception 196, 203
- CosNaming NamingContext 195, 209
- creating a new naming context
 - aNewNamingContext (variable) 202
 - bind_new_context() 201
 - bind_new_context_with_string() 201
 - CosNaming::AlreadyBound exception 203

- creating a new naming context
 - (*continued*)
 - ORB::
 - resolve_initial_references 200
 - target naming context 201
- credential, acquiring the security
 - attributes of a 224
- credential on a thread, acquiring
 - a 227
- credentials, and secure associations, principals,
 - principal 222
- credentials, manipulating
 - acquiring a credential on a thread
 - invocation-credential 227
 - own-credential 227
 - received-credential 227
 - getting a current object 224
 - security attributes of a
 - credential 224
- credentials object 240
- current object 240
- current object, getting a 224
- current thread, resume a transaction
 - on the 274
- current thread, suspend a transaction
 - from the 273

D

- data array, queries that result in
 - a 328, 377
- data array query 310, 358, 378
- database, building an interface
 - repository 468
- database management systems
 - (DBMS) 318, 365
- DB2 limits, Cache Service and 396
- DB2 LOBs and DB2 data types 352, 391
- DB2 locking, Cache Service and 396
- DBMS (database management systems) 318, 365
- DBMS pushdown rules 318, 365
- deadlocks
 - prevent 18
 - timeout value 18
- decisions, heuristic 258
- design a Transaction Service
 - application 266
- determination, problem 285
- differences between OOSQL and SQL
 - correlation ids 358
 - dereference operator 310, 358
 - FROM clause 309, 358
 - home collections 309, 358

- dii, (dynamic invocation interface) 410
- disconnect_push_supplier() 46
- distributed object system, naming
 - objects in the 181
- distributed object system, security in
 - the 221
- dynamic invocation interface
 - (DII) 410

E

- end-users, authentication and 231
- environment variables, logging in
 - with
 - SCSCCELLNAME (variable) 236
 - SCSPASSWORD (variable) 236
 - SCSPRINCIPAL (variable) 236
- environment variables to establish
 - authenticity, using 236
- establish authenticity, using
 - environment variables to 236
- evaluate_to_data_array() 377, 379
- evaluator, get the server name of a
 - query 371
 - client vs. server process 371
 - strategies for finding server name
 - ask an administrator 372
 - ask the user 372
 - create an anchor 373
 - know collections to know
 - server 372
 - specialize your
 - collections 374
- evaluator interface, an example
 - using the query
 - data array query 378
 - evaluate_to_iterator() 379
- evaluators and collections, topology
 - of query
 - evaluate_to_data_array() 325, 374
 - evaluate_to_iterator() 325, 374
- event channels
 - configuring 41, 73
 - connecting 42, 74
 - creating 39, 71
 - disconnecting 76
 - event channel samples
 - changePrice() 46
 - constructor 47
 - disconnect
 - _push_consumer() 49
 - disconnect_push_supplier() 46
 - ec 47
 - echome 47
 - FilterFactory and filters 77

- event channels (*continued*)
 - locating 38, 70
 - schemas
 - multiple 29, 60
 - single 29, 59
- events
 - consumers 33, 64
 - consuming 34, 65
 - suppliers 30, 60
 - supplying 31, 62
 - topics
 - multiple event channel
 - schemas 29, 60
 - single event channel
 - schemas 29, 59
- events, communicating
 - asynchronous
 - events
 - channels 24, 52
 - consumers 24, 52
 - structured 52
 - suppliers 24, 52
 - topics 24, 52
 - filters 52
 - events, structured 56
- example, Java clients and Java
 - BO 385
- example of a transaction, an 251
- example using the query evaluator
 - interface, an
 - data array query 378
 - evaluate_to_iterator() 379
- exceptions
 - and recovery 462
 - NoFactory 161
- exceptions, handle 20, 280
 - invalid transaction 281
 - transaction required 281
 - transaction rolledback 281
- explicit propagation 254
- explicit session propagation 300

F

- factories, concepts of
 - application factory 114
 - beneficiary object 113
 - managed object factories 113
 - Policy object 113
 - PolicyHolder object 113
- factory interface 157
- factory keys
 - CosLifecycle::Key 157
 - id 157
 - IExtendedLifecycle::
 - FactoryKeyString 157
 - kind 157

- factory keys (*continued*)
 - object interface 161
 - federated name tree 193
 - file, server key-tab
 - key-tab file 233
 - protecting the key-tab file 233
 - in AIX 235
 - in NT file system 235
 - rgy-edit 233
 - flat transactions, top-level and nested transaction 253
 - subtransaction 253
 - force a transaction to rollback 277
 - foreign key pattern 353, 392
 - form a query 326, 375
- G**
- get the server name of a query evaluator 371
 - client vs. server process 371
 - strategies for finding server name
 - ask an administrator 372
 - ask the user 372
 - create an anchor 373
 - know collections to know server 372
 - specialize your collections 374
 - getting a current object 224
 - granularity
 - intention read mode 9
 - intention write mode 9
 - group identity 448
- H**
- handle exceptions 280
 - invalid transaction 281
 - transaction required 281
 - transaction rolledback 281
 - heuristic damage 258
 - heuristic decisions 258
 - heuristic hazard 255
 - heuristic mixed outcome 258
 - home collections 321, 367
 - homes, WLM 450
 - host, local root naming context and bootstrap
 - bootstrap host 207
 - client machines 188
 - resolve_initial_references() 207
 - host name tree 187, 189
 - host name tree, local and
 - locations to bind resources 189
 - ORB::
 - resolve_initial_references 189
 - host name tree, local and (*continued*)
 - organization rules for resources 188
- I**
- id 157
 - identity, group 448
 - ids and passwords, user 232
 - IExtendedNaming
 - NamingContext 198, 209
 - implementations, IOM interfaces and 424
 - implementing the Naming Service
 - NamingStringSyntax
 - StandardSyntaxModel 212
 - NamingStringSyntax
 - StringName 212
 - string syntax object, the 211
 - XFN (X/Open Federated Naming) specification
 - standard 212
 - implicit propagation 254
 - implicit session context
 - propagation 300
 - implicitly propagate transaction context
 - to a remote object 277
 - IMS applications, access to 293
 - infrastructure scope boundaries 128
 - inheritance 313, 362
 - instance variables
 - price (variable) 46
 - pxyPushC (variable) 46
 - integration of system name spaces
 - host tree, bind 192
 - remote name context
 - binding 192
 - specifying hosts 193
 - intention read lock 6
 - interface, factory 157
 - interface, summary of the naming context
 - CosNaming NamingContext
 - introduces operations 209
 - IExtendedNaming
 - NamingContext introduces operations 209
 - interface repository
 - database, building an 468
 - displaying the contents
 - executable, running the 470
 - irdump, running 470
 - makefile 469
 - makefile, running the 470
 - test.IDL file 469
 - interface repository (*continued*)
 - ODBC for AIX, configuring 466
 - ODBC for NT, configuring 465
 - interfaces, Transaction Service objects and 267
 - invocation, remote method 399
 - invocation interface (DII),
 - dynamic 410
 - IOM and Component Broker 421
 - IOM interfaces and
 - implementations 424
 - IR (interface repository)
 - database, building an 468
 - displaying the contents
 - executable, running the 470
 - irdump, running 470
 - makefile 469
 - makefile, running the 470
 - test.IDL file 469
 - ODBC for AIX, configuring 466
 - ODBC for NT, configuring 465
 - is_identical() 103, 104
- J**
- Java, communication between C++ and 424
 - Java BO example, Java clients and 385
 - Java client
 - local C++ object (nt only),
 - compile and link to DLL 431
 - compile the Java files 433
 - create IDL file 430
 - implement getMessage 431
 - implementation-side C++
 - binding files 430
 - produce the client stub 432
 - run application 433
 - write and compile Java client program 433
 - Java client of a remote C++ object
 - C++ implementation classes 441
 - compile and link DLL 443
 - create IDL file 441
 - implement getMessage 442
 - Java client program 444
 - link the C++ pieces to DLL 443
 - main server program 442
 - run application 445
 - Java clients and Java BO example 385
 - Java object, C++ client of a local
 - compile and link to client 429
 - compile Java file 428
 - create IDL file 425
 - implement getMessage 427

- Java object, C++ client of a local *(continued)*
 - produce C++ client file 428
 - run application 429
 - run idlc command 426
 - write C++ client main program 429
- Java object, C++ client of a remote 434
 - C++ ORB adapter 436
 - compile and link client program 439
 - compile and link the server 438
 - compile Java pieces 436
 - create client program 438
 - create IDL files 435
 - implementation of getMessage() 436
 - Java implementation bindings 435
 - Java server code 436
 - run application 439
- K**
 - key-tab file 233
 - key-tab file, server
 - key-tab file 233
 - protecting the key-tab file 233
 - in AIX 235
 - in NT file system 235
 - rgy-edit 233
 - keys, factory
 - CosLifeCycle::Key 157
 - id 157
 - IExtendedLifeCycle::FactoryKeyString 157
 - kind 157
 - object interface 161
 - kind field 155, 157
- L**
 - Language, object-oriented Structured Query
 - result collection 308, 356
 - returned collection 308, 356
 - structured query language (SQL) 308, 356
 - leaks in the Transaction Service, prevent memory 282
 - LifeCycle Service
 - distributed system 111
 - factory-finders 112
 - location object 112
 - lifetime of a transaction 253
 - limit for all new transactions, set a time 271
 - limits, Cache Service and DB2 396
 - limits (timeouts), transaction time
 - timed out 261
 - limits, transaction retry 261
 - listing the contents of a naming context
 - CosNaming::NamingContext::list() 203
 - list_with_string() 204
 - network latency 203
 - next_n() 204
 - next_one() 204
 - local activation 454
 - local and host name tree
 - locations to bind resources 189
 - ORB::
 - resolve_initial_references 189
 - organization rules for resources 188
 - local C++ object (nt only)
 - Java client 430
 - local only, object 124
 - local root naming context and bootstrap host
 - bootstrap host 207
 - client machines 188
 - resolve_initial_references() 207
 - location based factory finding
 - infrastructure scope boundaries
 - values/scopes (table) 128
 - topology scope boundaries
 - cell 128
 - host 128
 - values/scopes (table) 129
 - workgroup 128
 - location object implementations and proximity
 - compound-conditional location object 121
 - political proximity 120
 - locations
 - proximity scopes 116
 - compound 116
 - compound-conditional 116
 - compound-temporal-conditional 116
 - geographical 116
 - infrastructural 116
 - physical 116
 - political 116
 - proximal 116
 - temporal 116
 - topological 116
 - lock(), (blocking) 12
 - lock requests in a lock set, servicing 7
 - lock sets
 - conflicting locks in a 6
 - creating 10
 - defining 10
 - locks and 5
 - obtain and release locks from a 12
 - related 4
 - servicing lock requests in a 7
 - locking, Cache Service and DB2 396
 - locking, Cache Service and Oracle 397
 - locks
 - change mode of lock 17
 - exclusive 5
 - in a lock set, conflicting 6
 - intention 5
 - lock sets and 5
 - modes 5
 - non-transactional, use
 - get_resource() 16
 - obtain and release from a lock set 12
 - possession, multiple 7
 - read and write 5
 - release in a transactional framework 13
 - requests in a lock set, servicing 7
 - shared 3
 - thread-based 3
 - log, the Transaction Service 282
 - logging in with environment variables
 - SCSCELLNAME (variable) 236
 - SCSPASSWORD (variable) 236
 - SCSPRINCIPAL (variable) 236
 - loginhelper object 241
 - M**
 - manage transactions in your application
 - access the TransactionCurrent object 270
 - set a time limit for all new transactions 271
 - start a transaction using the TransactionCurrent interface 272
 - suspend a transaction from the current thread 273
 - manageable objects, workload 449
 - managed object 124
 - managed object..., adding a 460

- managed object developer use of the Session Service 292
- managed objects and local only objects
 - application factory 124
- management, affinity 451
- management, memory 350, 388
- managing updates to resources
 - server processors 4
- manipulating credentials
 - acquiring a credential on a thread
 - invocation-credential 227
 - own-credential 227
 - received-credential 227
 - getting a current object 224
 - security attributes of a credential 224
- memory leaks in the Transaction Service, prevent 282
- memory management 350, 388
- message protection
 - confidentiality and integrity protection 238
 - confidentiality protection 238
 - data encryption service (des) 238
 - integrity protection 238
 - none 238
- method invocation, remote 399
- methods 311, 360
- model, client programming
 - exceptions and recovery 462
 - factory finders, using 461
- model, programming
 - group identity 448
 - references to other objects 453
 - WLM homes 450
 - workload manageable objects 449
- model grammar, standard syntax
 - attributes, standard syntax model 214
 - standard model rules for parsing string names 213
- models, communication 25, 54
- modes, lock
 - intention read lock 6
 - intention write lock (IW) 6
 - lock mode capability (table) 6
 - read lock mode (R) 6
 - upgrade lock mode (U) 6
 - write lock mode (W) 6
- mounted 187, 207
- multiple activation 452
- multiple concurrent threads
 - collaborate on session outcome amongst 305
- multiple lock possession
 - unlock requests, number 7

N

- name, binding an object with a
 - bind() 196
 - bind_with_string() 196
 - CosNaming::AlreadyBound exception 196
 - Naming Service 197
 - ORB::
 - resolve_initial_references 197
 - resolve() 197
 - target naming context (variable) 196
 - resolve() 196
- name (variable) 199, 204
- name binding 181
- name component 194
- name of a query evaluator, get the server 371
 - client vs. server process 371
 - strategies for finding server name
 - ask an administrator 372
 - ask the user 372
 - create an anchor 373
 - know collections to know server 372
 - specialize your collections 374
- name space, navigation in the system
 - cell and .: 191
 - workgroup 191
- name spaces, integration of system
 - host tree, bind 192
 - remote name context binding 192
 - specifying hosts 193
- name-string and name-structure, converting between
 - aNameString (variable) 218
- name strings 195
- name-structure, converting between name-string and
 - aNameString (variable) 218
- name tree, local and host
 - locations to bind resources 189
 - ORB::
 - resolve_initial_references 189
 - organization rules for resources 188
- name trees 183
- named object, resolving a 198
- names, absolute and relative 208
- names, object
 - compound name 194
 - CosNaming::NamingContext 195
 - name component 194
 - name strings 195
 - simple name 194
- NameStringSyntax
 - StandardSyntaxModel 210
- naming context, creating a new
 - aNewNamingContext (variable) 202
 - bind_new_context() 201
 - bind_new_context_with_string() 201
 - CosNaming::AlreadyBound exception 203
 - ORB::
 - resolve_initial_references 200
 - target naming context 201
- naming context and bootstrap host, local root
 - bootstrap host 207
 - client machines 188
 - resolve_initial_references() 207
- naming context interface, summary
 - CosNaming NamingContext introduces operations 209
 - IExtendedNaming NamingContext introduces operations 209
- naming objects in the distributed object system 181
- Naming Service, implementing the
 - NameStringSyntax StandardSyntaxModel 212
 - NameStringSyntax StringName 212
 - string syntax object, the 211
 - XFN (X/Open Federated Naming) specification standard 212
- NameStringSyntax
 - StringName 212
- navigation in the system name space
 - cell and .: 191
 - workgroup 191
- nested transaction 253
- non-managed single-location scope, creating a
 - cell name tree 187
 - mounted 187

- non-managed single-location scope, creating a (*continued*)
 - scope-structure, using
 - `_create(scope)` 163
 - static functions for creating
 - `_create` 163
 - workgroup name tree 187
- non-recoverable client 263
- non-transactional locking, transactional and 3
- NTFS (NT file system) 235

O

- object, access the
 - `TransactionCurrent` 270
- object, resolving a named 198
- Object Builder, using
 - adding a container 459
 - managed object, adding a 460
- object collection, queries that result in an 328, 376
- object developer use of the Session Service, managed 292
- object from a naming context, unbinding an 205
- object implementations and proximity, location
 - compound-conditional location object 121
 - political proximity 120
- object interface 157, 161
- object names
 - compound name 194
 - `CosNaming::NamingContext` 195
 - name component 194
 - name strings 195
 - simple name 194
- object-oriented Structured Query Language
 - result collection 308, 356
 - returned collection 308, 356
 - structured query language (SQL) 308, 356
- object relationship support 353, 391
- object system, naming objects in the distributed 181
- object system, security in the distributed 221
- object with a name, binding an
 - `bind()` 196
 - `bind_with_string()` 196
 - `CosNaming::AlreadyBound` exception 196
 - Naming Service 197
 - ORB::
 - `resolve_initial_references` 197

- object with a name, binding an (*continued*)
 - `resolve()` 197
 - target naming context (variable) 196
 - `resolve()` 196
- objects
 - collections, optimizations for 106
 - comparing 103
 - `constant_random_id()` 104
 - `is_identical()` 104
 - multiple 104
 - two 104
 - visibility of named 187
- objects, application 454
- objects, business 459
- objects, managing 20
- objects, Security Service
 - credentials object 240
 - current object 240
 - loginhelper object 241
 - principal object 240
- objects, transient 456
- objects and interfaces, Transaction Service 267
- objects to string form, conversion of 399
- one-phase commit process 258
- OOSQL (object-oriented Structured Query Language) 308, 356
- OOSQL and SQL, differences between
 - correlation ids 358
 - dereference operator 310, 358
 - FROM clause 309, 358
 - home collections 309, 358
- optimistic cache options (table) 395
- optimizations, query 317, 364
- optimizations for object collections 106
- Oracle locking, Cache Service and 397
- ORB::`resolve_initial_references` 200
- outcomes, transaction
 - commit 255
 - heuristic hazard 255
 - heuristic mixed 255
 - rolled back 255
- over unnamed collections, queries
 - example using the query evaluator interface, an 378

P

- parameter lists not named collections 353, 391
- parsing string names, standard model rules for 213
- passwords, user ids and 232
- Policy object 113
- PolicyHolder object 113
- possession, multiple lock
 - unlock requests, number 7
- prevent memory leaks in the Transaction Service 282
- principal object 240
- principals, credentials, and secure associations
 - principal 222
- priorities, resource 290
- problem determination 285
- programming considerations
 - granularity 8
 - manage objects 8
 - resources, shared 8
- programming model
 - group identity 448
 - references to other objects 453
 - WLM homes 450
 - workload manageable objects 449
- programming model, client exceptions and recovery 462
 - factory finders, using 461
- programming model, common client 294
- protecting the key-tab file 233
- protection, message
 - confidentiality and integrity protection 238
 - confidentiality protection 238
 - data encryption service (des) 238
 - integrity protection 238
 - none 238
- proximity, location object implementations and
 - compound-conditional location object 121
 - political proximity 120
- pseudo-host name tree 190
- purpose of a Concurrency Service
 - resource access
 - lock mode 3
 - read lock 3
 - write lock 3
 - simultaneous updates (table) 2
- push down 317, 364

pushdown rules, DBMS 318, 365

Q

queries on queryable collections
 query over reference
 collections 322, 368

queries over unnamed collections
 example using the query
 evaluator interface, an 378

queries that result in a data
 array 328, 377

queries that result in an object
 collection 328, 376

query, form a 326, 375

query evaluator, get the server name
 of a 371
 client vs. server process 371
 strategies for finding server name
 ask an administrator 372
 ask the user 372
 create an anchor 373
 know collections to know
 server 372
 specialize your
 collections 374

query evaluator interface, an
 example using the
 data array query 378
 evaluate_to_iterator() 379

query evaluator usage 391

query evaluators and collections,
 topology of
 evaluate_to_data_array() 325,
 374
 evaluate_to_iterator() 325, 374

query optimizations 317, 364

query over persistent objects 352,
 391

query over reference collections
 createCollectionFor() 322, 368
 DB2 search engine 352, 369

Query Service tips
 conditions required for
 queries 351, 390
 DB2 LOBs and DB2 data
 types 352, 391
 deferred updates 352, 390
 foreign key pattern 353, 392
 object relationship support 353,
 391
 parameter lists not named
 collections 353, 391
 query evaluator usage 391
 query over persistent
 objects 352, 391
 timeout settings 352, 391

query statement processing 352,
 390

queryable collections, queries on
 query over reference
 collections 322, 368

R

R (read lock mode) 6

rebind, automatic 454

recoverability 254

recoverable server 265

recovery and exceptions 462

reference collections, query over
 createCollectionFor() 322, 368
 DB2 search engine 352, 369

register sessionable resources 303

related lock sets 4

relationship between transactions
 and sessions 289

relative names, absolute and 208

remote C++ object, Java client of a
 C++ implementation classes 441
 compile and link DLL 443
 create IDL file 441
 implement getMessage 442
 Java client program 444
 link the C++ pieces to DLL 443
 main server program 442
 run application 445

remote Java object, C++ client of
 a 434
 C++ ORB adapter 436
 compile and link client
 program 439
 compile and link the server 438
 compile Java pieces 436
 create client program 438
 create IDL files 435
 implementation of
 getMessage() 436
 Java implementation
 bindings 435
 Java server code 436
 run application 439

remote method invocation 399

remote method invocations
 code-set conversion for
 char code sets 402
 native char code set 401
 native wchar code set 401
 translation enabled 401
 use ISO-Latin1 401
 wchar code sets 402

remote name context binding 192

remote object, implicitly propagate
 transaction context to a 277

replicated data store 194

repository, interface
 database, building an 468
 displaying the contents
 executable, running the 470
 irdump, running 470
 makefile 469
 makefile, running the 470
 test.IDL file 469
 ODBC for AIX, configuring 466
 ODBC for NT, configuring 465

requests in a lock set, servicing
 lock 7

reset session context, checkpoint and
 example of checkpointing and
 resetting 302

resolve() 196

resolve_initial_references() 207

resolving a named object 198

resource priorities 290

resources, managing updates to
 server processors 4

resources, register sessionable 303

result collection 364, 381

result in an object collection, queries
 that 328, 376

resume a session, suspend and 299

resume a transaction on the current
 thread 274

retry limits, transaction 261

rollback, force a transaction to 277

rules, DBMS pushdown 318, 365

rules, visibility
 CICS applications, access to 294
 common client programming
 model 294
 IMS applications, access to 293

run-time support, configure 21

running system, Transaction Service
 in a
 configuring a server, Transaction
 Service 285
 problem determination 285
 server start-up, types of 284
 Transaction Service log 282
 .csh 283
 .ctl 283
 .nnn 283
 somtrnnn (variable) 283

S

scope 254

- scope, creating a non-managed
 - single-location
 - cell name tree 187
 - mounted 187
 - scope-structure, using
 - _create(scope) 163
 - static functions for creating 163
 - _create 163
 - workgroup name tree 187
- scope and context, transaction
 - explicit propagation 254
 - implicit propagation 254
 - scope 254
 - transaction context 253
- scope of location, defining
 - container 128
 - home 128
 - server 128
- scope of sessions, the 287
- scope structures and strings
 - boundry value 155
 - kind value 155
 - scope boundary 155
- scopes, Component Broker location
 - infrastructure scope boundaries
 - values/scopes (table) 128
 - topology scope boundaries
 - cell 128
 - host 128
 - values/scopes (table) 129
 - workgroup 128
- secure associations, principals, credentials, and
 - principal 222
- security attributes of a credential, acquiring the 224
- security considerations for the server, other
 - AIX default locations 230
 - WIN default locations 230
- security in the distributed object system 221
- Security Service objects
 - credentials object 240
 - current object 240
 - loginhelper object 241
 - principal object 240
- server, other security considerations for the
 - AIX default locations 230
 - WIN default locations 230
- server, recoverable 265
- server, transactional 265
- server key-tab file
 - key-tab file 233
- server key-tab file (*continued*)
 - protecting the key-tab file 233
 - in AIX 235
 - in NT file system 235
 - rgy-edit 233
- server name of a query evaluator, get the 371
 - client vs. server process 371
 - strategies for finding server name
 - ask an administrator 372
 - ask the user 372
 - create an anchor 373
 - know collections to know server 372
 - specialize your collections 374
- server processors 4
- server start-up 284
- server to use the Transaction Service, configuring a 285
- Service, implementing the Naming
 - NamingStringSyntax
 - StandardSyntaxModel 212
 - NamingStringSyntax
 - StringName 212
 - string syntax object, the 211
 - XFN (X/Open Federated Naming) specification
 - standard 212
- Service, LifeCycle
 - distributed system 111
 - factory-finders 112
 - location object 112
- Service application, design a
 - Transaction 266
- Service log, the Transaction 282
- Service objects and interfaces, Transaction 267
- session, suspend and resume a 299
- session, the timeout value associated with a 289
- session context, checkpoint and reset
 - example of checkpointing and resetting 302
- session outcome, collaborate on
 - amongst multiple concurrent threads 305
- Session Service, client use of the 290
- Session Service, managed object
 - developer use of the 292
- Session Service, using the
 - client use of the Session Service 290
- Session Service, using the (*continued*)
 - managed object developer use of the Session Service 292
- Session Service tasks
 - begin a session 295
 - end a session 297
 - set a time limit 295
- sessionable resources, register 303
- sessions, relationship between
 - transactions and 289
- sessions, the scope of 287
- set a time limit for all new transactions 271
- single-location scope, creating a non-managed
 - cell name tree 187
 - mounted 187
 - scope-structure, using
 - _create(scope) 163
 - static functions for creating 163
 - _create 163
 - workgroup name tree 187
- SQL, differences between OOSQL and
 - correlation ids 358
 - dereference operator 310, 358
 - FROM clause 309, 358
 - home collections 309, 358
- SQL (structured query language) 308, 356
- standard exceptions, CORBA
 - CORBA::INITIALIZE 273
 - CORBA::INVALID_TRANSACTION 273
 - CORBA::PERSIST_STORE 273
- standard model rules for parsing
 - string names 213
- standard syntax model
 - attributes 214
- standard syntax model grammar
 - attributes, standard syntax model 214
 - standard model rules for parsing string names 213
- StandardSyntaxModel
 - StringName 210
- start a transaction using the
 - TransactionCurrent interface
 - CORBA::INITIALIZE 273
 - CORBA::INVALID_TRANSACTION 273
 - CORBA::PERSIST_STORE 273
- start-up, server 284
- string form, conversion of objects to 399

- string syntax object, the 211
 - structured events 56
 - structured query language,
 - object-oriented
 - result collection 308, 356
 - returned collection 308, 356
 - structured query language (SQL) 308, 356
 - subtransaction 253
 - summary of the naming context interface
 - CosNaming NamingContext introduces operations 209
 - IExtendedNaming NamingContext introduces operations 209
 - summary of the Transaction Service 286
 - suppliers 23
 - suppliers, event 30
 - suspend a transaction from the current thread 273
 - suspend and resume a session 299
 - syntax model grammar, standard
 - attributes, standard syntax model 214
 - standard model rules for parsing string names 213
 - syntax object, the string 211
 - system name space 123
 - system name space, navigation in the
 - cell and `::` 191
 - workgroup 191
 - system name spaces, integration of
 - host tree, bind 192
 - remote name context binding 192
 - specifying hosts 193
- T**
- target naming context (variable) 196, 198
 - tasks, Concurrency Service
 - change mode of lock 17
 - complete top level transactions 14
 - configure run-time support 21
 - CosNaming::`AlreadyBound` 196
 - creating a lock set 10
 - define a lock set 10
 - handle exceptions 20
 - managing objects 20
 - non-transactional locks, use 15
 - tasks, Concurrency Service (*continued*)
 - obtain and release locks from a lock set 12
 - preventing deadlocks 18
 - related lock sets 4
 - relating lock sets 11
 - release locks in a transactional framework 13
 - troubleshooting 21
 - tasks, Session Service
 - begin a session 295
 - end a session 297
 - set a time limit 295
 - thread, acquiring a credential on a 227
 - thread, pass a transaction context to another 275
 - thread, resume a transaction on the current 274
 - thread, suspend a transaction from the current 273
 - time limit for all new transactions, set a 271
 - time limits (timeouts), transaction
 - timed out 261
 - timed out 261
 - timeout settings 352, 391
 - timeout value associated with a session, the 289
 - timeouts, transaction time limits
 - timed out 261
 - tips, Query Service
 - conditions required for queries 351, 390
 - DB2 LOBs and DB2 data types 352, 391
 - deferred updates 352, 390
 - foreign key pattern 353, 392
 - object relationship support 353, 391
 - parameter lists not named
 - collections 353, 391
 - query evaluator usage 391
 - query over persistent objects 352, 391
 - timeout settings 352, 391
 - top-level and flat transactions
 - nested transaction 253
 - subtransaction 253
 - topology of query evaluators and collections
 - `evaluate_to_data_array()` 325, 374
 - `evaluate_to_iterator()` 325, 374
 - topology scope boundaries 128
 - transaction, an example of a 251
 - transaction, lifetime of a 253
 - transaction context 253
 - transaction context to another thread, pass a 275
 - transaction context to remote object
 - implicitly propagate a 277
 - transaction from the current thread, suspend a 273
 - transaction on the current thread, resume a 274
 - transaction outcomes
 - commit 255
 - heuristic hazard 255
 - heuristic mixed 255
 - rolled back 255
 - transaction retry limits 261
 - transaction scope and context
 - explicit propagation 254
 - implicit propagation 254
 - scope 254
 - transaction context 253
 - Transaction Service, configuring a server to use the 285
 - Transaction Service, prevent memory leaks in the 282
 - Transaction Service, summary of the 286
 - Transaction Service application,
 - architecture and design
 - non-recoverable client 263
 - recoverable server 265
 - transactional server 265
 - Transaction Service application, design a 266
 - Transaction Service in a running system
 - configuring a server, Transaction Service 285
 - problem determination 285
 - server start-up, types of 284
 - Transaction Service log 282
 - `.csh` 283
 - `.ctl` 283
 - `.nnn` 283
 - `somtrnnn` (variable) 283
 - Transaction Service log, the 282
 - Transaction Service objects and interfaces 267
 - transaction time limits (timeouts)
 - timed out 261
 - transaction to rollback, force a 277

- transaction using the
 - TransactionCurrent interface, start a
 - CORBA::INITIALIZE 273
 - CORBA::INVALID_TRANSACTION 273
 - CORBA::PERSIST_STORE 273
- transactional and non-transactional locking 3
- transactional environment,
 - Concurrency Service 4
- transactional framework, release locks in a 13
- transactional server 265
- TransactionCurrent interface
 - start a transaction using the 272
- TransactionCurrent interface, start a transaction using the
 - CORBA::INITIALIZE 273
 - CORBA::INVALID_TRANSACTION 273
 - CORBA::PERSIST_STORE 273
- TransactionCurrent object, access the 270
- transactions, complete top level
 - get_coordinator() 15
 - uninvolve_in_transaction() 15
- transactions, set a time limit for all new 271
- transactions, top-level and flat
 - nested transaction 253
 - subtransaction 253
- transactions and sessions, relationship between 289
- transactions in your application, manage
 - access the TransactionCurrent object 270
 - set a time limit for all new transactions 271
 - start a transaction using the TransactionCurrent interface 272
 - suspend a transaction from the current thread 273
- transient objects 456
- troubleshoot 21
- try_lock(), (non-blocking) 12
- two-phase commit process
 - heuristic damage 258
 - heuristic mixed outcome 258

U

- U (upgrade lock mode) 6
- unbinding an object from a naming context 205
- unnamed collections, queries over
 - example using the query evaluator interface, an 378
- updates to resources, managing server processors 4
- user ids and passwords 232
- using environment variables to establish authenticity 236
- using factory finders 461
- using Object Builder
 - adding a container 459
 - managed object, adding a 460
- using the Session Service
 - client use of the Session Service 290
 - managed object developer use of the Session Service 292

V

- variables, logging in with environment
 - SCSCCELLNAME (variable) 236
 - SCSPASSWORD (variable) 236
 - SCSPRINCIPAL (variable) 236
- variables to establish authenticity, using environment 236
- visibility of named objects 187
- visibility rules
 - CICS applications, access to 294
 - common client programming model 294
 - IMS applications, access to 293

W

- W (write lock mode) 6
- WLM (workload manageable) objects 449
- WLM homes 450
- workgroup name tree 187, 189
- workgroups 190
- workload manageable objects 449

X

- X/Open Federated Naming (XFN) specification standard 212
- XFN(X/Open Federated Naming) specification standard 212



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-4443-00



Spine information:



WebSphere

Advanced Programming Guide

Version 3.0

SC09-4443-00