Managing Internationalization Contexts in IBM WebSphere®

On the Declarative Management of the Internationalization Contexts in Distributed Heterogeneous Client-Server Environments

Debasish Banerjee, WebSphere Internationalization Architect, IBM Rochester, MN, USA
Jeffrey A. Frey, WebSphere System/390 Chief Architect, IBM Poughkeepsie, NY, USA
Robert H. High, Jr., WebSphere Lead Architect, IBM Austin, TX, USA

Abstract. The internationalization service in IBM WebSphere family of products makes two 'internationalization contexts' available to the business methods: the caller and the invocation internationalization contexts. The default internationalization context management policy is to set the invocation internationalization context identical to the caller internationalization context. However, some business applications may want to operate under an invocation internationalization context that can potentially be different from the caller internationalization context.

While one can be permitted to set the invocation internationalization context programmatically, this should not be the preferred approach. Any programmatic alteration of internationalization context can be considered to be hard-wired inside the business logic. One should be able to alter the invocation internationalization context of a business method declaratively, from outside, without any re-coding, or re-compilation.

The present paper introduces the notion of declarative management of internationalization contexts through XML deployment descriptors. The deployment descriptors are associated with business methods or components running in managed environments like Java™ 2 Enterprise Edition (J2EE) or Corba Component Model. Along with the obvious route of programmatic manipulation, three attributes for the container-managed internationalization policy are introduced for setting the invocation internationalization contexts in managed environments. The internationalization service works in conjunction with Enterprise JavaBeans™ (EJB) and Web containers to set the appropriate invocation internationalization contexts. The effect of the policies and attributes on the invocation internationalization contexts is discussed in detail. The ongoing and future internationalization work in client-server environments is also highlighted.

1. Introduction [3] introduced the concept of 'internationalization context' and 'internationalization service' for localization in distributed client-server environments. An internationalization context consists of a chain of locales and a time zone. The locale chain is ordered according to the caller's preference. Inside any J2EE component, two types of internationalization contexts are made available by the internationalization service: the caller internationalization context representing the caller's internationalization information and the invocation internationalization context representing the internalization environment under which a business method should execute. Internationalization Service provides a few APIs using which an application can access the two categories of internationalization contexts. A server-side business method should run under the locale(s) and time zone contained in the

invocation internationalization context. Though some background information about internationalization service is included in Section 2 of this paper, for a more complete description, the reader is referred to [3]. At any node in a J2EE environment, the initial implementation of the internationalization service unconditionally equated the invocation internationalization context to the caller internationalization context. While this serves the purpose for the majority of business applications, some applications may want to exercise greater control for the invocation internationalization contexts. The present paper introduces a deployment descriptor based approach using which application deployers can specify the desired invocation internationalization context.

Section 2 provides some background information. Section 3 discusses the basic motivation for the present approach for the external specification of the invocation internationalization context. Section 4 details the various policies for managing the internationalization context along with their effect. Section 5 discusses the invocation internationalization context management policies for and their effects on various J2EE components. Section 6 draws the conclusion. Appendix A contains the DTD enhancements for Web applications and ejb-jars needed to accommodate the policies and attributes for the management of internationalization contexts.

2. Background The Internationalization Service of IBM WebSphere Enterprise Edition transparently propagates the internationalization context, consisting of locale and time zone information, in all remote invocations using the RMI-IIOP protocol. Server side J2EE components can use the propagated internationalization context in all the relevant locale and time zone-sensitive business methods.

The internationalization service works by associating an internationalization context with every thread of execution in an application. For propagating internationalization contexts to a target node, the upcoming version of internationalization service in IBM WebSphere, Enterprise Edition, release 5.0 uses the J2EE Activity Service [9]. When a client-side program invokes a remote method, the internationalization interceptor registered with the activity service transparently intercepts the request. The internationalization interceptor obtains the internationalization context associated with the current thread, and supplies it to the activity service in a serialized form. The sending side ORB marshals the activity service data, which includes the internationalization (and other applicable) context and propagates them to the target node along with the original remote method invocation. The ORB on the receiving side unmarshals the request along with all the contexts propagated in the activity service data including the internationalization context. The activity service running on the receiving side invokes the internationalization interceptor, which in turn extracts the propagated internationalization context and attaches the extracted context to the thread on which the remote business method executes. The internationalization service works harmoniously with activity service and ORBs to propagate this context on subsequent remote method invocations in the same manner, and thus distribute the context over an entire call chain.

The caller and the invocation, the two internationalization contexts are made available inside all IBM WebSphere Enterprise Edition J2EE methods for providing maximum flexibility in developing internationalized applications. The use of invocation internationalization context is rather obvious. A server-side business method can use the invocation internationalization information for properly localizing locale and time zone-sensitive results. The caller internationalization context permits one to pass non-canonical form of input parameters among other possible uses.

In a call chain of method ($m_i, 1 \le i \le n$) executions, $m_1() \rightarrow m_2() \rightarrow \ldots \rightarrow m_n()$, the following condition always remain satisfied.

$$\text{Invocation Internationalization Context at } m_i = \text{Caller Internationalization Context at } m_{i+1},\ 1 \le i < n \qquad (2.1)$$

Note that at any node in a client-server environment, only the invocation internationalization context is propagated by the internationalization service. The propagated internationalization context at the invoked node assumes the role of the caller internationalization context.

In general, inside a J2EE server-side method, the invocation internationalization context can be different from the caller internationalization context. However, the IBM WebSphere Enterprise Edition, release 4.0 [6], always expects that the business methods will localize results in caller's (client's) locale and time zone. Hence, it always sets the invocation internationalization context to be identical to the caller internationalization context.

The internationalization service provides the internationalization context interfaces. J2EE methods executing in IBM WebSphere Enterprise Edition can use the methods exposed in the interfaces to access relevant locales and time zones.

```
public interface Internationalization
{
   // returns an ordered array of locales in decreasing order of preference
   public java.util.Locale[] getLocales();
   // returns the preferred locale—the first element of the locale array
   public java.util.Locale getLocale();
   // returns a time zone
   public java.util.TimeZone getTimeZone();
}

public interface InvocationInternationalization extends Internationalization
{
   // sets the ordered array of locales
   public void setLocales(java.util.Locale[] locales);
   // sets the preferred locale—the first element of the locale array
   public void setLocale(java.util.Locale locale);

   // sets a time zone
   public void setTimeZone(java.util.TimeZone timeZone);
   // an alternate simpler way to set a time zone
   public void setTimeZone(String timeZoneId);
}

public interface UserInternationalization
{
   // returns the Caller Internationalization Context
   public Internationalization getCallerInternationalization();
   // returns the Invocation Internationalization Context
   public InvocationInternationalization getInvocationInternationalization();
}
```

From inside a J2EE method, invocations of `getCallerInternationalization()` and `getInvocationInternationalization()` return the caller and invocation internationalization contexts. In case, where there is no caller internationalization context associated with the current thread of execution, on an invocation of `getCallerInternationalization()` method, internationalization service creates an internationalization context using the default locale and time zone of the underlying JVM, associates the context as the caller internationalization context with the current thread of execution, and finally returns the created context. The created default caller internationalization context satisfies the following two conditions at the underlying JVM.

```
UserInternationalization.getCallerInternationalization.getLocales() =
                          [java.util.Locale.getDefault()]          (2.2), and
UserInternationalization.getCallerInternationalization.getTimeZone() =
                         java.util.TimeZone.getDefault()           (2.3)
```

3. Managed Internationalization Consider Figure 1 where a Spanish client is invoking the server side business method `m()`. The internationalization service propagates the invocation internationalization context of the client consisting of the `es_ES` locale and `PST` time zone during the invocation of `m()`.

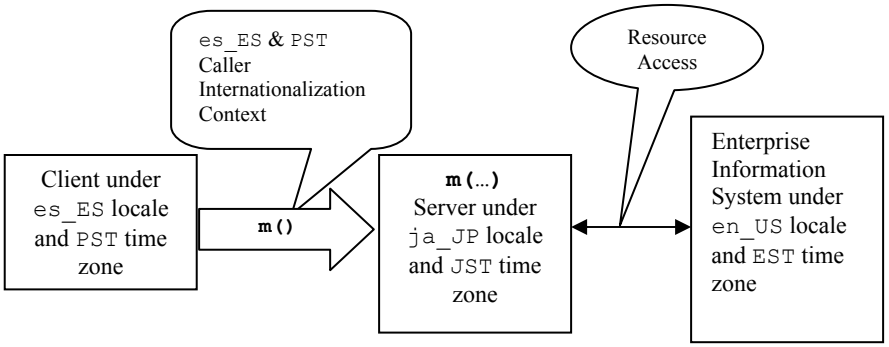

Figure 1

Following condition (2.1) one can easily see that, in the IBM WebSphere, Enterprise Edition, release 4.0, the method `m()` executes under the invocation internationalization context identical to that of the client. Let us assume that the business method `m()` during its execution interacts with an enterprise information system and the corresponding resource adapter uses the invocation internationalization context of `m()` using the proposed extension to the J2EE Connector Architecture [13] hinted in [3]. Assume further that the method `m()` issues the following SQL query

```
SELECT dvd_name FROM dvd_table WHERE dvd_name <= 'N'
```

The above mentioned query will not return any name starting with Ñ if the query gets executed in the Spanish locale, the invocation locale of the method `m()`. For returning all the names, the business method `m()` which uses the above mentioned query should preferably be executed in neutral locale, American for example.

Modern application components may again in some circumstances be connected to legacy systems with very limited localization capabilities, some of which may work properly only in American locale. To interact properly with legacy systems, an application component may want to work under an invocation internationalization context consisting of a fixed locale of choice.

While it is possible to alter the invocation internationalization context programmatically inside business methods using the internationalization service APIs, this should not be considered as the preferred approach for most usage scenarios. Any programmatic alteration of the invocation internationalization context can be considered hard wired in the application component. If a different invocation internationalization context is ever desired, the application component may need re-coding and re-compilation. Moreover, a J2EE application component developer may not be aware of the idiosyncrasies of the deployment environment. An EJB developer should not make any restrictive assumptions about its back-end store--the back-end can be an ultra modern database, or it can be a legacy EIS resource with limited localization capabilities.

Clearly there is a need for controlling the contents of the invocation internationalization context without taking the obvious route of programmatic manipulation. The application deployer should be able to specify declaratively, from outside, the 'desired invocation internationalization context' for a component or a business method without going through the cumbersome processes of re-coding and recompilation.

But for certain specialized applications, a server-side servlet or EJB programmer may want to take complete programmatic control of the invocation internationalization context. A server-side business method may have the capability to localize in only a few commonly occurring locales, and it may want to perform closest locale match using some proprietary algorithm. For example, assume that a business component decided to translate all the displayable texts in simplified Chinese and not in traditional Chinese. If the application detects the traditional Chinese locale in the caller internationalization context, it may decide to create an invocation internationalization context consisting of the simplified Chinese locale, which in general may roughly serve the purpose for most of the traditional Chinese clients.

For the above mentioned reasons, we introduce a few externally specifiable invocation internationalization context management policies and attributes applicable to the server side J2EE business components and methods.

4. Management Policies There are two internationalization context management policies: Application-managed Internationalization (AMI) and Container-managed Internationalization (CMI). The first policy allows programmatic alteration of the invocation internationalization contexts, while the second one is the preferred container-managed policy where a developer is prevented from altering the invocation internationalization context. In the following, we discuss the policies in some detail.

4.1 Application-managed Internationalization (AMI) This internationalization context management policy is applicable to all J2EE methods and components. Under this deployment policy, a programmer takes complete control of the invocation internationalization context, and can use the APIs provided by the internationalization service for programmatically setting the invocation internationalization context. In the absence of any explicit setting, the internationalization service will use the underlying JVM's default locale and time zone to establish a default invocation internationalization context. Formally speaking, for a J2EE component or method deployed as AMI, in the absence of any explicit programmatic setting, the invocation internationalization context satisfies the following two conditions at the underlying JVM.

```
UserInternationalization.getInvocationInternationalization.getLocales() =
                            [java.util.Locale.getDefault()]          (4.1.1), and
UserInternationalization.getInvocationInternationalization.getTimeZone() =
                           java.util.TimeZone.getDefault()           (4.1.2)
```

It should be noted in this context that in a call chain of execution, internationalization service satisfies condition (2.1). Hence if a J2EE method $m_1()$ deployed as AMI and executing under the default invocation internationalization context invokes another J2EE method $m_2()$, internationalization service will propagate the default invocation internationalization context and make it available as the caller internationalization context at $m_2()$.

4.2 Container-managed Internationalization (CMI) CMI is the preferred internationalization context management policy for the server-side J2EE components and methods, where the internationalization service working harmoniously with the Web or EJB containers sets the invocation internationalization contexts. From inside a CMI method, a programmer can only query, but may not able to alter the invocation internationalization context. Any attempt to do so will result in a `java.lang.IllegalStateException`. CMI is the default internationalization management policy for all server-side J2EE components and methods.

Container Management policies can have any of the following three attributes associated with them.

4.2.1 RunAsCaller This is the default CMI attribute. Setting this attribute ensures that inside a J2EE server-side method deployed as CMI, the invocation internationalization context will be same as the caller internationalization context, satisfying the following two conditions.

```
UserInternationalization.getInvocationInternationalization.getLocales() =
UserInternationalization.getCallerInternationalization.getLocales()      (4.2.1.1)
                                    and
UserInternationalization.getInvocationInternationalization.getTimeZone() =
UserInternationalization.getCallerInternationalization.getTimeZone()     (4.2.1.2)
```

In the absence of a caller internationalization context, which can happen if the caller is a back-level WebSphere or a non-IBM J2EE node, the internationalization service will create the caller and hence the invocation internationalization context using the default locale and time zone of the underlying JVM. Under such a circumstance, in addition to the two core conditions (4.2.1.1) and (4.2.1.2), four other default conditions: (2.2), (2.3), (4.1.1), and (4.1.2) will also be satisfied at the underlying JVM.

4.2.2 RunAsServer The internationalization service will create the invocation internationalization context using the default locale and time zone of the underlying JVM satisfying conditions (4.1.1) and (4.1.2) at the underlying JVM.

4.2.3 RunAsSpecified<locale(s), time zone> The internationalization service will create the invocation internationalization context from the locale(s) and time zone specified in the deployment descriptor.

5. Programming Model This section discusses the internationalization context management policies applicable to or allowed for various J2EE components, and the effect of those policies on selecting the appropriate J2EE programming style for successful distributed internationalization.

5.1 J2EE Applet and HTTP Clients The J2EE applet container is not required to support the RMI-IIOP protocol, and hence a J2EE applet client may not be able to propagate internationalization context (or any other contexts) to the Web container. The same is also true for a HTTP client running under the control of a browser. In fact, from the viewpoint of internationalization, an applet client running in an applet container can simply be looked as a Web HTTP client communicating directly with Web-side server component(s) using HTTP. Since there is no RMI-IIOP support available inside an applet or in a HTTP client, neither an applet nor a HTTP client can perform any JNDI lookup, which immediately precludes the possibility of accessing or managing internationalization service within applets or HTTP clients. Internationalization service does not exist in the applet container or in a simple browser environment, and no internationalization context management policies can be specified for a J2EE applet or a HTTP client.

There is a subtle but important difference between an applet client and a Web HTTP client regarding internationalization. For both an applet and a Web HTTP client the host browsers's language settings get propagated as the 'Accept-Language' HTTP headers [5], from which the server-side Web components derive the clients' locale(s) according to the servlet specification [11]. An applet runs in a JVM inside a browser. The JVM's default locale is derived from the default locale of the underlying operating system. It should be noted that none of the present day browsers carry the JVM's locale information in the HTTP header of applet client requests. Now potentially there can be a mismatch between the browsers's language settings and the locales of the applet JVMs. For executing reliable internationalized applet clients, the applet programmer should make sure that the following condition remains satisfied during all remote invocations.

`java.util.Locale().getDefault()` at the JVM hosting the applet =
`javax.servlet.ServletRequest.getLocale()` at the server-side Web component. (5.1.1)

5.2 Java Application Client J2EE application clients run essentially in non-managed environments. They will always be deployed implicitly as AMI. No explicit internationalization deployment descriptors are supported for J2EE application clients.

An application client is free to set any desired list of locales and time zone information in the invocation internationalization information using the APIs provided by the internationalization service. For a J2EE application client, which is always the very first originating element in a call chain of execution, the internationalization service establishes a default caller internationalization context satisfying conditions (2.2) and (2.3) at the underlying JVM. In the absence of any explicit setting of the invocation internationalization context, conditions (4.1.1) and (4.1.2) will be satisfied at the underlying JVM.

5.3 Server-side Web Components Servlets and JSPs, the two server-side J2EE Web components can get deployed as either AMI or CMI. The IBM WebSphere extended deployment descriptor can be used to specify the internationalization context management policy, with CMI being the default in the absence of the explicit internationalization deployment descriptor.

Since a servlet (or a JSP) has only one predefined method, the `javax.servlet.service()` method, which accepts client requests, there is no need to be able to specify the internationalization context management policy at method level for the J2EE server-side Web components. For a servlet (or a JSP)

handling HTTP client requests, the caller internationalization context is determined [3] from the "Accept-Language" HTTP header.

Besides the most visible `javax.servlet.service()` method, there are other categories of Web methods [11]. Filters, servlet life-cycle methods, Web application event listeners, and servlet event listeners are also used in Web applications. The internationalization context management policy specified for a servlet may not be applicable in general to any of the above mentioned non-service method categories. Refer to [2, 4] for comprehensive treatment of internationalization contexts inside filters and other non-service methods of a Web application.

5.4 Enterprise Java Beans Session and message-driven beans can be deployed as either AMI or CMI, with CMI being the default in the absence of any explicit internationalization deployment descriptor. For prescribing a disciplined programming model in the managed J2EE environment, we restrict entity beans to be only of CMI variety. Any business logic that needs to use an AMI entity bean method, can introduce a AMI session bean invoking the entity bean method deployed as CMI with **RunAsCaller** attribute.

For session and entity beans, the internationalization context management policies and relevant attributes can be specified both at the bean level, as well as at the method level for the usual business methods, home methods, and finder methods exposed in `EJBObject`, `EJBLocalObject`, `EJBHome`, and `EJBLocalHome` interfaces.

Theoretically speaking, for session and entity beans, we could have allowed a bean developer, using IBM WebSphere extended deployment descriptors, to designate some bean methods to run under the AMI policy, specifying the rest of the relevant methods to be executed under the CMI policy. This strategy would have imparted more flexibility in using the internationalization contexts. However, this flexibility can potentially create serious confusion among bean developers and deployers; it may also generate extreme complexity in the 'persistence manager' implementations needed to allocate connections to the back-end stores. After a careful consideration of all the complexity issues, we decided not to allow a mixture of AMI and CMI methods in an EJB.

Note that, a message-driven bean does not have any home or component interfaces, and it has only one method: the `onMessage()` method, which can potentially contain business logic. For a message-driven bean, the internationalization context management policy can be specified only for its `onMessage()` method. Hence a message-driven bean can be deployed as either AMI or CMI, a mixing of these two policies is not even a theoretical possibility for message-driven beans. Though the present EJB specifications [10] prohibits a Message-driven bean's `onMessage()` method from getting associated with any inbound transaction or security contexts, we are allowing, in principle, the `onMessage()` method to be associated with a caller internationalization context. Theoretically speaking, the caller internationalization contexts for message-driven beans can be specified in a standardized way in JMS messages [12]. Since presently there exists no standard definition of the internationalization information in a JMS message, for the time being, a message-driven bean, even though deployed as CMI with **RunAsCaller** attribute, will effectively run as CMI with **RunAsServer** attribute satisfying conditions (4.1.1) and (4.1.2) at the underlying JVM.

Besides the business, home, finder and the `onMessage()` methods, there are a few other methods defined in `EJB(Local)Home` and the `EJB(Local)Object` interfaces; a session bean, an entity bean, and a message-driven bean also contain a number of life cycle methods. The non-business, non-home, non-finder, and non-JMS message consuming methods present in home, component, and bean interfaces, cannot have any internationalization context management policies associated with them. Refer to [2, 4] for detailed discussions regarding the behaviour of the internationalization service and internationalization contexts inside the above mentioned EJB methods.

5.4.1 EJB Local Methods The EJB 2.0 specification [10] has introduced the concept of local home and local component interfaces. The parameters in a local call are passed by reference, while the parameters in a remote call are passed by value. In distributed client-server environments, one can view the service

contexts, like security context, transaction context, internationalization context, etc. as implicit parameters propagated and managed by the underlying middleware.

Contexts, viewed as implicit parameters, can also be passed by reference. This approach can potentially create very confusing and error-prone programming model.

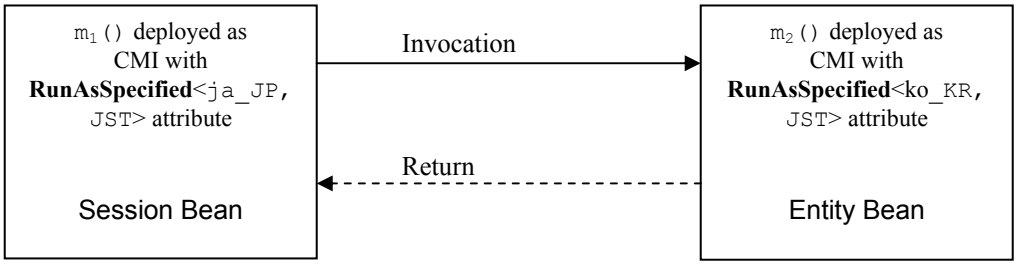

Figure 2

As an example, consider Figure 2, where a session bean method `m1()` makes a local call to an entity bean method `m2()` collocated in the same JVM.

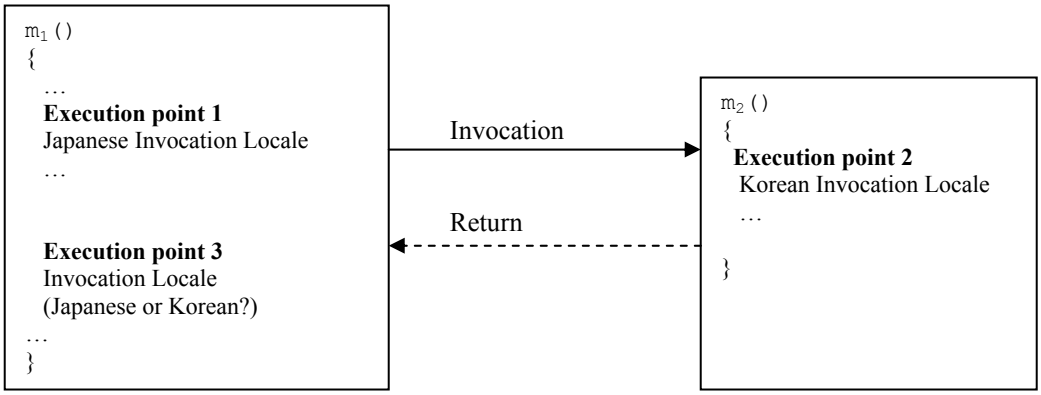

Figure 3

Figure 3 tries to depict the execution sequence of Figure 2 focusing on the invocation internationalization context. `m1()` initially executes under Japanese invocation locale. After the invocation of `m2()`, the thread of execution gets associated with Korean locale--the invocation locale for the method `m2()`. When the control returns to the session bean, after the completion of `m2()`, what should be value of the invocation locale under which the remaining portion of `m1()` will execute? Considering contexts to the implicit parameters, by-reference semantics should set the invocation locale at execution point 3 to be Korean. This can be very confusing to a J2EE programmer—a part of `m1()` executes under Japanese invocation locale leaving the rest of `m1()` to run under Korean invocation locale!

Again, in addition to viewing contexts as implicit parameters, one can also consider, both intuitively as well as formally, the context management policies and attributes to be strictly *scoped* only to the methods for which the policies and attributes are specified--the method `m1()` of Figures 2 and 3 must always execute under Japanese invocation locale.

To satisfy the above mentioned scope criterion and also to avoid somewhat inconsistent semantics as explained in connection to Figures 2 and 3, we use the 'call-by-value' semantics for contexts, when viewed as implicit parameters, for both remote and local EJB calls.

It should be noted that EJBs can have internal helper methods which are not exposed in component interfaces. Helper methods of an EJB can only be invoked internally by other methods defined in the same EJB; they cannot be directly invoked from any other EJB or J2EE component methods. Internationalization context management attributes cannot be specified for EJB internal methods, and internationalization context, when viewed as implicit parameters assumes a call-by-reference semantics inside these internal helper methods. Explicit parameters inside EJB internal methods, of course follows Java's normal call-by-value semantics. Table T1 summarizes the parameter passing semantics for all the EJB methods. Table T1 should be applicable not only for internationalization context but also for any other relevant service contexts as well.

Method Category / **Parameter Type**	`EJBHome`	`EJBObject`	`EJBLocalHome`	`EJBLocal Object`	Internal EJB Methods
Explicit	by value	by value	by reference	by reference	by value
Implicit (Context)	by value	by value	by value	by value	by reference

Table T1

6. Conclusions The work presented in the paper is continuation of the distributed internationalization work presented earlier in [3]. The present paper simplified and enhanced the use of internationalization service in J2EE applications by introducing the concept of declarative management policies and attributes for internationalization contexts in managed environments. The invocation internationalization context for server-side J2EE components and methods can be precisely specified by a application deployer using XML deployment descriptors. The container-managed internationalization (CMI) is the preferred internationalization context management policy, where the internationalization service working harmoniously with the Web or EJB containers sets the desired invocation internationalization context. If necessary, different invocation internationalization contexts for server-side J2EE components or methods can be effected by redeploying applications after appropriately modifying the relevant internationalization deployment descriptors—no programmatic alteration is necessary. By including the application-managed internationalization (AMI) policy, the present paper also allows a J2EE server-side programmer to take complete programmatic control over the invocation internationalization context.

The applicability and availability of the different internationalization context management policies and attributes for various J2EE components, and the associated programming model and style generated thereof are discussed in detail. Subtleties, newly introduced by the EJB local calls, in the operational semantics of the run time service contexts, when viewed as implicit parameters are also elaborated.

Though not presently implemented, the concept of the declarative management of internationalization contexts is equally applicable and implementable for CORBA business objects in the domain of CORBA component model [8].

The present version of the internationalization service is implemented in IBM WebSphere Enterprise Edition, release 5.0. The applicability of the internationalization service is extended [1] to the domain of enterprise Web Services [7]. A technical preview of internationalized enterprise Web Services is available in the IBM WebSphere Enterprise Edition, release 5.0. IBM is actively exploring the possibility of appropriate and smooth introduction of internationalization context management policies and attributes in the XML descriptions of enterprise Web Service elements.

We intend to enhance the internationalization service in the domain of J2EE, by introducing a simpler and easier-to-use programming model for distributed internationalization. With appropriate support from J2EE

deployments tools and containers, the programming model will be built on the top of the existing internationalization service. It may allow J2EE server-side programmers to utilize internationalization service for properly localizing relevant computation with much ease and transparence. In the future, IBM plans to incorporate the high level programming model under investigation in the enterprise edition or some other version of the WebSphere product.

IBM intends to make all the internationalization service publicly available through the Java Community Process. The emerging JSR 150 specification [2] contains the detailed and more formal treatment of the axiomatic semantics and runtime behaviour of the internationalization service.

Acknowledgements
David Zavala of IBM, Rochester, MN, USA implemented the core internationalization service. Logan Colby of IBM Rochester, MN, USA participated in many stimulating discussions, and implemented the visual and XMI tooling aspects of the internationalization deployment descriptors for Web and EJB containers. Betsy Baartman of IBM, Rochester, MN, USA provided numerous suggestions for improving the quality of presentation.

References

1. Banerjee D. and Swenson C. Towards the Internationalization of Web Services in IBM WebSphere: On the Development of Internationalized Web Service Applications in Distributed Heterogeneous Client-Server Environments. To appear in the 22nd International Unicode Conference, San Jose, CA, Sep. 2002.
2. Banerjee D. Internationalization Service For J2EE, JSR 150. In preparation. http://jcp.org/jsr/detail/150.jsp
3. Banerjee D., Frey J. A., and High R. H., Jr. The Internationalization Service in IBM WebSphere: On the Development of Internationalized Applications in Distributed Heterogeneous Client-Server Environments. 20th International Unicode Conference, Washington, DC, Jan./Feb. 2002.
4. Banerjee D. and Zavala D. Managing Internationalization Context. System Design and Architecture Document. IBM Internal Document. Dec. 2001.
5. Fielding R., Gettys, J, Mogul, J., Frystyk, H., Masinter, M. Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol—HTTP/1.1. Network Working Group, *RFC 206*8, Jan. 1997.
6. IBM Corporation. *WebSphere Application Server Enterprise Edition 4.0: A Programmer's Guide*, RedBook SG24-6504-00, IBM Corporation, Armonk, NY, Feb. 2002. http://www.redbooks.ibm.com/
7. Knutson, J. , Kreger, H. *Web Services for J2EE, Version 1.0, Public Draft v0.3*, JSR 109, Apr. 2002. http://jcp.og/jsr/detail/109.jsp.
8. Object Management Group. CORBA Components Model: Volumes 1, 2 and 3. http://www.omg.org/cgi-bin/doc?orbos/99-07-01, http://www.omg.org/cgi-bin/doc?orbos/99-07-02, and http://www.omg.org/cgi-bin/doc?orbos/99-07-03, 1999.
9. Robinson, I. J2EE Activity Service Specification, JSR 095. http://jcp.org/jsr/detail/95.jsp.
10. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0, Final Release*, Palo Alto, CA, Aug. 2001.
11. Sun Microsystems. *Java Servlet Specifications, Version 2.3, Final Release*, Sun Microsystems, Palo Alto, CA, Aug. 2001.
12. Sun Microsystems. *Java Message Service, Version 1.0.2b*, Sun Microsystems, Palo Alto, CA, Aug. 2001.
13. Sun Microsystems. *J2EE Connector Architecture Specification, JSR 016*, Version 1.0, Final Release, Sun Microsystems, Palo Alto, CA, Jul. 2001.

Appendix A For the deployment descriptors of all WebSphere extensions to the core J2EE 1.3 model, including internationalization service, IBM presently uses XMI. For simplicity, we are presenting not the real life XMI syntax, rather the schematic DTDs for both Web applications and ejb-jars regarding the management of internationalization contexts following the style of [10, 11]. The DTDs presented here, where all the newly introduced elements are underlined for emphasis, abstractly represent the real life XMI syntax and semantics.

A.1 Servlets and JSPs J2EE server-side Web components can get deployed as either AMI or CMI. A servlet or a JSP can get deployed in a web application with an 'internationalization-type' element.

The DTD of a servlet (or a JSP) regarding the management of internationalization context can be described as follows.

```
<!ELEMENT servlet (icon?, servlet-name, …, security-role-ref*,
                   internationalization-type?)>

<!--
The internationalization-type element specifies a server-side component's
internationalization type. The internationalization-type element must be one of
the following:
<internationalization-type>Application</internationalization-type>
<internationalization-type>Container</internationalization-type>

The default is Container.

Used in:  servlet.
-->
<!ELEMENT internationalization-type (#PCDATA)>

<!--
The Web application deployer can use the ‘container-internationalization’
element to specify the CMI attributes for the servlets which are specified to
have CMI policy in their deployment descriptor. The ‘container-
internationalization’ element must not be specified for servlets, which have
‘Application’ as its ‘internationalization-type’.

The ‘web-app’ element can optionally contain a list of ‘container-
internationalization’ elements for specifying the container-managed
internationalization attributes of servlets. A ‘container-internationalization’
element must not be used to specify the CMI attributes of multiple servlets.
For a Web application, the application deployer can optionally specify the CMI
attributes for all the servlets running under CMI policies using a separate
‘container-internationalization’ element for each servlet.

Used in:  WAR
-->
<!ELEMENT web-app (icon?, display-name?, ..., ejb-ref*,
                   container-internationalization*)>

<!--
The container-internationalization element specifies the CMI attributes of a
specific servlet. The ‘servlet-name’ must be one of the servlets used in the
Web application which are supposed to run under CMI policy.

Used in:  web-app
-->
<!ELEMENT container-internationalization (servlet-name, description?,
                   container-internationalization-attribute)

<!--
The ‘container-internationalization-attribute’ specifies the CMI attributes.
The element must be one of the following, ‘RunAsCaller’ being the default.
```

```
      <container-internationalization-attribute>RunAsCaller</container-
internationalization-attribute>
      <container-internationalization-attribute>RunAsServer</container-
internationalization-attribute>
      <container-internationalization-attribute>
      <RunAsSpecified> ... </RunAsSpecified>
      </container-internationalization-attribute>

Used in:  container-internationalization
-->
<!ELEMENT container-internationalization-attribute (description?,
                  (#PCDATA | RunAsSpecified)?)>

<!--
The 'RunAsSpecified' element can be used to explicitly specify the invocation
Internationalization Context of a servlet.

Used in:  container-internationalization-attribute
-->
<!ELEMENT RunAsSpecified (description?, locale+, time-zone)>

<!--
The 'locale' element is modeled after JDK's java.util.Locale element. It
consists of four elements: an optional description, an optional language code,
an optional country code, and an optional variant string, with the following
two restrictions.
      (a)  The language-code, the country-code and the variant all three can
not be missing at the same time.
      (b)  Only the variant can not be present while both the language-code and
the country-code elements are missing.

Used in:  RunAsSpecified
-->
<!ELEMENT locale (description?, language-code?, country-code?, variant?)>

<!--
The 'language-code' element can be a two character ISO-639 symbol for a
language.

Used in:  locale
-->
<!ELEMENT language-code (#PCDATA)>

<!--
The 'country-code' element can be a two character ISO-3166 symbol for a
country.

Used in:  locale
-->
<!ELEMENT country-code (#PCDATA)>

<!--
The 'variant' element can be any valid string.

Used in:  locale
-->
<!ELEMENT variant (#PCDATA)>

<!--
The 'time-zone' element should be any of the valid time zone Ids of the JDK.
The JDK Ids are derived from the Olson database. In case the
internationalization service is supplied with an unrecognized time zone Id, a
```

```
SimpleTimeZone object representing GMT will be used to create the invocation
Internationalization Context.

Used in:  RunAsSpecified
-->
<!ELEMENT time-zone (description?, #PCDATA)>
```

A.2 EJBs An EJB can get deployed as either AMI or CMI, but not as a mixture of both policies with the additional restriction that an entity bean can not be deployed as AMI.

The DTD for the EJB deployment descriptor regarding internationalization service can be described as follows.

```
<! --
...
The 'internationalization-type' element can optionally appear in the 'message-
driven' element of an ejb-jar.

Used in:  enterprise-beans
-->
<!ELEMENT message-driven (description?, display-name?, ..., resource-env-ref*,
                    internationalization-type?)>

<! --
...
The 'internationalization-type' element can optionally appear in the 'session'
element of an ejb-jar.

Used in:  enterprise-beans
-->
<!ELEMENT session (description?, display-name?, ..., resource-env-ref*,
                    internationalization-type?)>

<!--
The 'internationalization-type' element can be used to override the default CMI
policy of session or message-driven EJBs. The 'internationalization-type'
element must be one of the following two, 'Container' being the default.
       <internationalization-type>Application</internationalization-type>
       <internationalization-type>Container</internationalization-type>

The value of 'Application' signifies all the relevant methods of the bean under
concern will run under Application-managed Internationalization (AMI).
Similarly, 'Container' indicates that all the relevant methods of the bean
under concern will run under Container-managed Internationalization (CMI)
.
Used in:  message-driven and session
-->
<!ELEMENT internationalization-type (#PCDATA)>

<!—
…
An application deployer can use the 'container-internationalization' to specify
the container management policy of various bean methods. Note that while
individual bean developers may specify the 'internationalization-type' element,
it is the duty of application deployer to specify the 'container-
internationalization' element.

The assembly-descriptor element may contain the internationalization attributes
for EJB methods which are deployed to run under CMI policies.
```

```
A 'container-internationalization' element must only be used to specify the CMI
attributes of the selected methods for a single EJB. To specify the CMI
attributes of multiple EJBs in an ejb-jar, multiple instances of 'container-
internationalization' elements must be present, one for each EJB.

Used in:  ejb-jar
-->
<!ELEMENT assembly-descriptor (security-role*, method-permission*, …,
                    , exclude-list?, container-internationalization*)>

<!--
The container-internationalization element specifies how the container must
manage the invocation internationalization context for the enterprise bean's
method invocations. The element consists of a list of method elements, and an
internationalization attribute. The internationalization attribute is to be
applied to all the specified methods.

Used in:  assembly-descriptor
-->
<!ELEMENT container-internationalization (description?, method+,
      container-internationalization-attribute)>

<!--
The 'container-internationalization-attribute' specifies the CMI attributes.
The element must be one of the following, 'RunAsCaller' being the default.
      <container-internationalization-attribute>RunAsCaller</container-
internationalization-attribute>
      <container-internationalization-attribute>RunAsServer</container-
internationalization-attribute>
      <container-internationalization-attribute>
            <RunAsSpecified> ... </RunAsSpecified>
      </container-internationalization-attribute>

Used in:  container-internationalization
-->
<!ELEMENT container-internationalization-attribute (description?,
            (#PCDATA | RunAsSpecified)?)>
```

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:
IBM WebSphere System/390

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information in this presentation addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.