The Internationalization Service in IBM WebSphere®

On the Development of Internationalized Applications in Distributed Heterogeneous Client-Server Environments

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

***Abstract**. None of the established and well-respected client-server architectures like CORBA, Java™ 2 Enterprise Edition (J2EE), or Microsoft™ DCOM presently provide any infrastructure for distributed internationalization. All of them tacitly assume that in multi-lingual environments, a server may freely impose its own locale and time zone on all the locale- and time zone-sensitive computation requests from clients.*

The Internationalization Service, a potentially new CORBA Common Object Service, provides a sound, robust, and flexible architectural infrastructure for properly localizing heterogeneous client-server applications distributed over IIOP. The internationalization service makes the 'internationalization context' available to the business methods. The internationalization context consists of the locale and time zone information under which a business method should execute. The internationalization service transparently creates and propagates the internationalization context during business method invocations. No extra parameters are needed in the method interfaces for accessing the internationalization contexts. With minimal extra programming effort, using a Java Naming and Directory Interface™ (JNDI) lookup, a business method can obtain the locale and time zone information from the internationalization context using the APIs provided by the internationalization service. The locale and time zone can be used to properly localize relevant computations in distributed environments.

The present paper discusses the principal issues behind the development of a sound architecture supporting distributed internationalization in client-server environments. The design of internationalization service is discussed in detail. The use of the internationalization service in IBM WebSphere® for developing internationalized J2EE applications is illustrated by appropriate examples. The ongoing and future internationalization work in client-server environments is also highlighted.

1. Introduction Traditional programming environments like XPG4 [26], Win32® [11, 19], or Java™ [3, 4] provide sound infrastructures for developing internationalized applications; applications which are developed once and which can be successfully executed in any language and country environment. There exists an abundance of literature dealing with the topic of internationalization in all the aforementioned popular environments. Unfortunately, almost all the present methodologies and available literature address the topic of internationalization only in single process or single JVM environments. Most of the available methodologies or literature do not address the issue of internationalization in distributed environments. In distributed client-server environments where clients and servers can reside in different machines, internationalization can be complicated. The present paper discusses an infrastructure which can be used for successfully developing enterprise quality internationalized distributed applications.

Section 2 identifies the principal issues that need to be addressed for successful internationalization in distributed client-server environments. Section 3 introduces the new approach for distributed internationalization as available in IBM WebSphere Enterprise Edition, release 4.0. Section 4 illustrates the approach with examples. Finally, Section 5 draws the conclusion.

2. Distributed Internationalization In a distributed client-server environment, application components may run in heterogeneous environments. Components can be hosted in machines having different endian architectures, configured to different code sets, locales, etc. A successful distributed internationalization infrastructure may have to deal with four issues in general.

2.1 Endian Mismatch Assume that a little-endian client is communicating with a big-endian server using UCS-2 [25] encoding. Also, assume that the client has transmitted 0xC30x6F, which corresponds to 濃, a Japanese Kanji character in little-endian format. If the server-side application literally accepts the transmitted UCS-2 code point, it will be interpreted as 썯, a Korean Hangul syllable in big-endian format. For correct interpretation, the client-server framework has to perform byte-swapping on the server-side to compensate for the endian-mismatch. OMG has formally addressed this endian-mismatch issue by introducing byte-order indicators [14, Chapter 15.3.1] in the marshaled data, and by formalizing the use of BOM [14, Chapter 15.3.1.6] for UTF-16 [25] encoded data. The Java world has simplified this endian-mismatch issue by mandating the big-endian format for all data marshaled by a Java ORB. DCOM uses [13] the little-endian format for externalizing data.

2.2 Code Set Mismatch The selection of appropriate code sets for applications and all the associated code set conversions are topics that occupy a substantial portion of any internationalization debate. For non-Unicode® client-server applications, a potential code set mismatch can result in data loss or corruption. For example, assume a Windows client using Windows-1252 code set is communicating with an AIX® server configured to ISO-8859-15 code set. Also, assume that the client has transmitted the code point 0x80 corresponding to the € symbol. If the server literally attempts to interpret the transmitted code point, the € symbol will be trashed in the server—0x80 is not a valid code point in ISO-8859-15 code set. To compensate for the code set mismatches between client and server nodes, OMG has introduced the code set conversion framework [14, Chapter 13.10.2]. In this particular example, a code set conversion needs to be performed so that the 0x80 code point on the client side gets converted directly or indirectly to 0xA4 on the server-side for the correct transmission and interpretation of €. Java and in general the J2EE world has simplified this issue by making Unicode the process code set of a JVM. DCOM also uses Unicode for data transfer.

2.3 Locale Mismatch In traditional client-server environments, the results of any locale-sensitive client initiated computations are usually returned in the locale of the server. If the clients and servers are all configured to be in the same locale, the imposition of server's locale on client requests will remain unnoticed. With the advent and the wild popularity of internet-based business models, like eCommerce applications, client and server machines may be configured to different locales, and computations in the server's locale may not be acceptable to the clients. Figure 1 illustrates a locale-mismatch problem where a Spanish client is connected to an American server. The client invokes some locale-sensitive business methods on the server. The results are returned in the American locale. Ideally, following the rule of 'local-remote transparency' in distributed environments, the client should expect the locale-sensitive results in it's (Spanish) locale and not in the server's (American) locale.

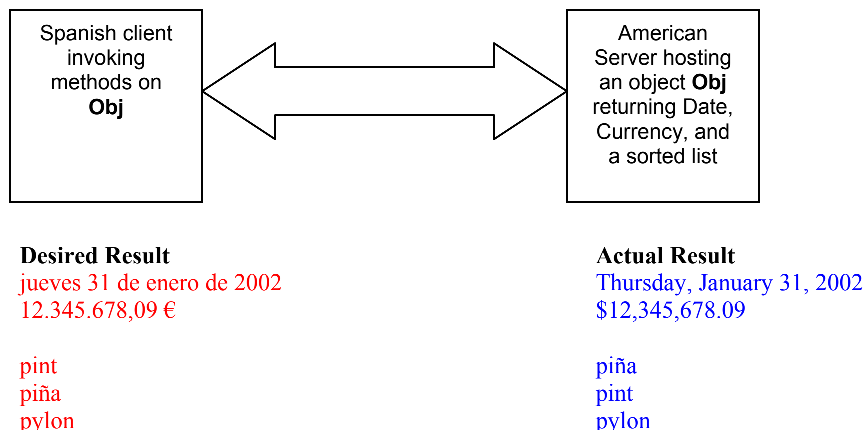

Figure 1

Neither CORBA [14] nor J2EE [22] address the issue of locale mismatch architecturally. DCOM [5] also ignores the locale mismatch issue. Application developers can pass extra parameters carrying the locale information to locale-sensitive business methods. However, this approach is intrusive and error-prone. In a call chain of execution, even if the final method is locale-sensitive, the 'locale' parameter has to be passed all the way starting from the beginning of the call chain. Again, the business methods can generate exceptions containing text messages. Calling nodes should receive the text messages formatted in their locales and not the server's locale. For generating localized exception messages, the 'locale' parameter needs to be added in all the business methods. This simple parameter addition approach also suffers from another serious disadvantage. The interfaces of most of the existing applications can not easily be changed—any interface change will necessitate a complex re-deployment.

2.4 Time Zone Mismatch In modern eCommerce environments, client and server machines can be located across different time zones resulting in time zone mismatches between clients and servers. Some business method computations can be time zone-sensitive in nature. Financial transaction containing the originating client request's timestamp can be cited as an example. Again quite similar to the case of locales, a server usually imposes its own time zone on client requested time zone-sensitive computations.

To avoid the time zone mismatch problem, some traditional applications always use GMT time zone. While GMT times may be suitable for machines, some users may consider it artificial for human interpretation. Application developers can also pass extra parameters carrying the client's time zone information to all the time zone-sensitive methods. The parameter passing approach suffers from the same disadvantages as mentioned earlier in Section 2.3.

3. Internationalization Service The Internationalization Service, presently available in IBM WebSphere Enterprise Edition provides a unique solution to both the locale and time zone mismatch issues in a J2EE environment. The internationalization service transparently propagates the internationalization context, consisting of locale and time zone information, in all remote invocations using the RMI-IIOP protocol [17]. 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. When a client-side program invokes a remote method, the internationalization service transparently intercepts. The service obtains the context associated with the current thread, marshals it and attaches the marshaled contents to the outgoing request. The caller-side ORB [14] marshals the outgoing request and transmits the request along with the attached internationalization context. At the server-side, the internationalization service again intercepts at the ORB level. The service detaches the caller's internationalization context from the incoming request, unmarshals it, and associates it with the thread on which the remote business method executes. The internationalization service works harmoniously with ORBs to propagate this context on subsequent remote method invocations in the same manner, and thus distribute the context over an entire call chain. Figure 2 depicts the propagation of internationalization context at a certain level of abstraction.

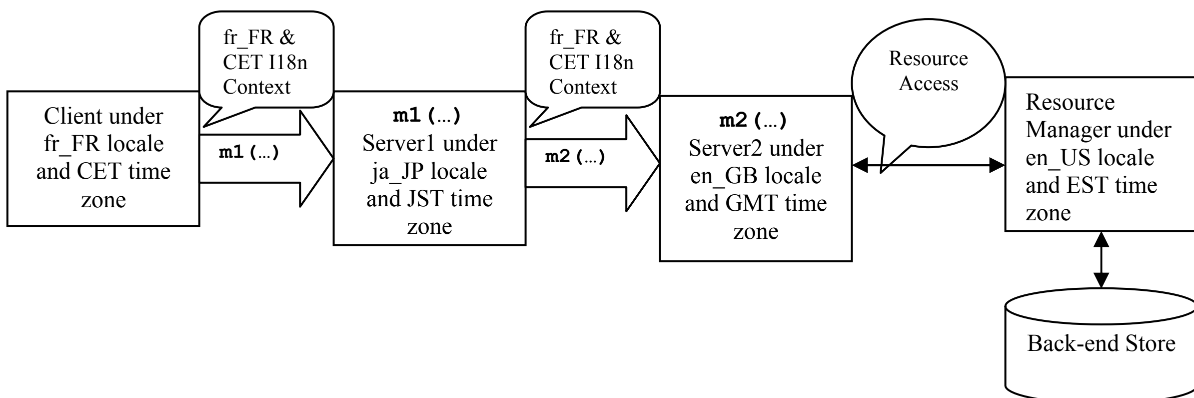

Figure 2

A French client running under Central European Time (CET) time zone makes a remote call to a business method `m1()` hosted in Server1, which in turn calls method `m2()` residing in Server2. Though Server1 and Server2 are configured to run under different locales and time zones as shown, all the remote calls are associated with the internationalization context consisting of the client's locale and time zone. The business logic of methods `m1()` and `m2()` can access the propagated internationalization contexts and localize results in the caller's locale and time zone if necessary. Figure 2 shows a resource manager invocation from method `m2()` for accessing data stored in a back-end data store. Presently there exists no architectural solution to make the resource manager aware of internationalization contexts. IBM intends to propose [2] a formal mechanism to pass the internationalization context to the back-end resource adapters using the J2EE Connector architecture [23].

3.1 Internationalization Contexts Inside all IBM WebSphere Enterprise Edition J2EE methods, two internationalization contexts are made available by the internationalization service: the caller internationalization context and the invocation internationalization context. The caller internationalization context corresponds to the caller's internationalization information, while the invocation internationalization context corresponds to the internationalization information under which a J2EE method executes. The presence of two internationalization contexts is intended to provide 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. For example, even if a client passes a numeric data as a formatted string, a server-side business method can parse the received data using the locale present in the caller internationalization context and subsequently use the numeric value in relevant computations.

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, 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. This should serve the purpose of the majority of internationalized client-server applications.

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

$$\text{Invocation Internationalization Context at } m_i = \text{Caller Internationalization Context at } m_{i+1},\ 1 \le i < n \qquad (3.1.1)$$

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

3.2 Internationalization Context Interfaces 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();
}
```

No J2EE component should be able to alter the caller internationalization information, and the interface definition of the caller internationalization context provides read-only access. The present WebSphere implementation enforces a Container Managed Internationalization (CMI) policy for all server-side J2EE components. Though the invocation internationalization context provides read-write access, the underlying CMI policy will throw a `java.lang.IllegalStateException` exception on any attempt to modify the invocation internationalization context from inside Servlets, JavaServer Pages™ (JSP)s, or Enterprise JavaBean™ (EJB)s, the three server-side J2EE components.

3.3 Internationalization Contexts in J2EE Components The following sections describes various J2EE components that can create and use the internationalization contexts.

3.3.1 J2EE Application Client The internationalization service runs at J2EE client containers in IBM WebSphere Enterprise Edition, and a J2EE application client is permitted to set the invocation internationalization context using appropriate `setxxx()` methods. In the absence of any explicit setting, the internationalization service will create the default invocation internationalization context satisfying the following conditions at the underlying JVM.

```
UserInternationalization.getInvocationInternationalization.getLocales() =
                           [java.util.Locale.getDefault()]   (3.3.1.1), and
UserInternationalization.getInvocationInternationalization.getTimezone() =
                           java.util.Timezone.getDefault()   (3.3.1.2)
```

Since a J2EE client is the root of any call chain of invocations, the internationalization service sets the caller invocation internationalization context to be identical to the invocation internationalization context at any J2EE client.

A client-server pair can use the ordered array of locales in an attempt to maintain the locale-remote transparency in J2EE environments. Figure 3 shows a situation where a Japanese Java client is communicating with a Korean EJB server. Assume that the client issues a remote request to execute method `m()` under Taiwanese locale. Also, assume that the required Taiwanese resources are missing in both client and server, while both the nodes contain the required resources for the execution of `m()` under Japanese locale. If the method `m()` is located in the client, the resource bundle search strategy of JDK [3] would have resulted in the use of Japanese resource. But if only the `zh_TW` locale is propagated in the internationalization context, and used at the server, JDK's resource bundle search strategy would have resulted in the execution of `m()` under Korean locale, the default locale of the server JVM. However, a

client can also pass its default locale (`ja_JP` in this case) in the caller internationalization context, and the server-side method `m()` may decide to use the second locale if localized resources are missing for the first one, to maintain the much desirable local-remote transparency.

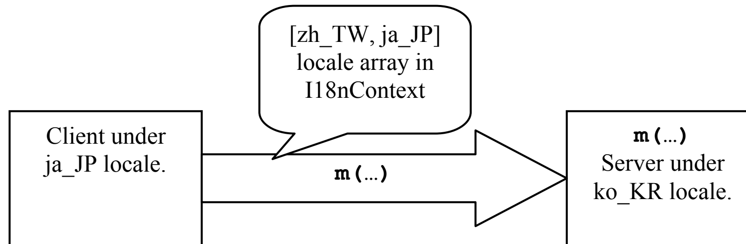

Figure 3

3.3.2 Java Applet, HTTP Clients, and Web Container The internationalization service does not exist in Applet container or in a simple browser environment, and hence no web client has access to any internationalization context interface. A Java applet or a Web HTTP client communicates with Java Servlets or JSPs in a J2EE environment using HTTP protocol. The internationalization service running at the J2EE Web container derives the caller internationalization information from the "Accept-Language" HTTP header [7] in accordance with the Servlet specifications [21]. Inside a Servlet (or a JSP), the internationalization service will satisfy the following conditions in the JVM hosting the Web container. Note that, all HTTP communications are supposed to be in GMT time zone.

```
Considering any loop index i
      for 0 ≤ i < L.length, L[I] = (java.util.Locale)E.nextElement()
      for i = L.length, E.hasMoreElements() = false, where
  L = UserInternationalization.getCallerInternationalization.getLocales(), and
  E = javax.servlet.ServletRequest.getLocales()                    (3.3.2.1)
                          and
UserInternationalization.getInvocationInternationalization.getTimezone() =
                          java.util.Timezone.getTimeZone("GMT")    (3.3.2.2)
```

3.3.3 EJB Container Internationalization service running at the EJB container associates internationalization contexts with the threads used by the EJB container to execute the methods defined in Session or Entity beans. If the incoming request does not contain a caller internationalization context, the internationalization service creates a default internationalization context satisfying conditions (3.3.1.1) and (3.3.1.2) in the underlying JVM.

3.3.4 Accessing Internationalization Contexts In IBM WebSphere Enterprise Extension 4.0, from inside a J2EE application client, or a server-side component (Servlet, JSP, and EJB) a J2EE programmer can access the `UserInternationalization` interface by using a simple JNDI lookup. The JNDI lookup is performed on the initial JNDI context using the URL `java:comp/websphere/UserInternationalization`. If for some reason the `UserInternationalization` interface is unavailable, the JNDI lookup throws a `javax.naming.NamingException.`

3.3.5 Programming Model and Style The server-side J2EE components are expected to localize relevant business computations in the locale and time zone information present in the invocation internationalization context. The JDK provides a rich set of classes and methods for common localization needs. Most of the JDK provided localization functions have two variants: one accepts locale as an explicit parameter, the other one does not have any placeholder for the locale parameter. The second variants are used mostly by the programmers in non-distributed environments, and they actually execute in the default locale of the underlying JVM. For proper localization in distributed environments, programmers can not blindly use the default locale of the underlying JVM. They should extract the appropriate locale from the invocation

internationalization context and use that locale as an input parameter when invoking any relevant method on Java's localization classes.
To access the invocation or caller internationalization contexts one needs to perform a JNDI lookup. The JNDI lookup can be performed only once during a component's lifetime. The reference(s) to the `InvocationInternationalization` or `CallerInternationalization` interfaces can also be obtained only once and cached for later use. For a Servlet, the references to the internationalization contexts can be obtained in either variety of the `init()` method [21]. Since the internationalization contexts are interfaces containing no static information, it is perfectly safe to cache their references in a Servlet instance, even for Servlets, which can potentially serve multiple requests at the same time using different threads of execution. Internationalization contexts are thread scoped and the internationalization service will either retrieve the locales and time zone from, or associate them with the specific thread under which the methods of Section 3.2 are invoked and executed. For a session bean, the references to the internationalization contexts can be obtained and cached in the `setSessionContext()` method. For an entity bean, the logical place to obtain and cache the references is the `setEntityContext()` method.

It should be mentioned that if new threads are created in an application, the internationalization contexts associated with the parent thread are not automatically transferred to the newly spawned threads. If any such child thread, like any user generated thread in the `service()` method of a Servlet, or any system generated event handling thread in an AWT client, invokes a remote method, the present implementation of internationalization service will propagate the default locale and time zone of the underlying JVM.

4. Examples This section illustrates the use of internationalization contexts by means of a few simple examples. [8] can be consulted for a real life sample application using internationalization contexts.

Example 4.3 Servlet The following code snippet illustrates the use of invocation internationalization context inside a Servlet. The reference to the `InvocationInternationalization` object is obtained in the `init()` method.

```
import java.util.*;
import java.text.*;
import javax.naming.*;

import javax.servlet.*;
import javax.servlet.http.*;
// . . .
// Internationalization Service imports
import com.ibm.websphere.i18n.context.*;

public class InternationalizedServlet extends HttpServlet
{
   public void init() throws ServletException
   {
      //obtain the initial context for JNDI lookup in IBM WebSphere 4.0
      try
      {
         Properties properties = new Properties();
         properties.put("java.naming.provider.url", "iiop:///");
         initialContext = new InitialContext(properties);
      }
      catch (NamingException namingException) { /* . . . */ }

      // Obtain a reference to the UserInternationalization object.
      try
      {
         userInternationalization = (UserInternationalization)
           initialContext.lookup("java:comp/websphere/UserInternationalization");
      }
      catch (NamingException namingException) { /* . . . */ }

      invocationInternationalization =
                 userInternationalization.getInvocationInternationalization();
```

```
      . . .
   }  // end of init

   public void doPost(HttpServletRequest request, HttpServletResponse response)
                                throws IOException, ServletException
   {
      // obtain the preferred locale
      Locale locale = invocationInternationalization.getLocale();
      // obtain the time zone
      TimeZone timeZone = invocationInternationalization.getTimeZone();
      // locale and timezone now can be used in relevant computations.
      NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
      . . .
   }
   . . .
   private InitialContext initialContext = null;
   private UserInternationalization userInternationalization = null;
   private InvocationInternationalization invocationInternationalization = null;
 }
```

Example 4.2 J2EE Application Client The following code snippet roughly represents the J2EE application client of Figure 3. The client issues a remote call to execute `m()`, a remote method of `mySessionBean` session bean.

```
import java.util.*;
import javax.naming.*;

// imports for creating and accessing the mySessionBean EJB
// . . .
// Internationalization Service imports
import com.ibm.websphere.i18n.context.*;

public Class J2EEClient
{
   public static void main(String[] args)
   {
      UserInternationalization userInternationalization = null;
      InitialContext initialContext = null;

      //obtain the initial context for JNDI lookup in IBM WebSphere 4.0
      try
      {
         Properties properties = new Properties();
         properties.put("java.naming.provider.url", "iiop:///");
         initialContext = new InitialContext(properties);
      }
      catch (NamingException namingException) { /* . . . */ }

      // Obtain a reference to the UserInternationalization object.
      try
      {
         userInternationalization = (UserInternationalization)
           initialContext.lookup("java:comp/websphere/UserInternationalization");
      }
      catch (NamingException namingException) { /* . . . */ }

      // obtain the reference to the EJB object
      // MySession mySession = . . .
      // set the invocation internationalization context
       InvocationInternationalization invocationInternationalization =
                   userInternationalization.getInvocationInternationalization();
       Locale[] locales = { new Locale("zh", "TW"), Locale.getDefault() };
       invocationInternationalization.setLocales(locales);
       invocationInternationalization.setTimeZone("JST"); //Japanese Standard Time
       // Invoke the remote method m. The internationalization service will propagate
       // the contents of the invocationInternationalization object.
       String returnedValue = mySession.m();
       . . .
      }
```

```
}
```

Example 4.3 EJB The following code snippet roughly represents the server-side bean implementation code of Figure 3. The `m()` method of `mySessionBean` EJB invoked by the client of Example 4.2 uses the second locale of the invocation internationalization context for localization.

```
import java.util.*;
import javax.naming.*;

// Internationalization Service imports
import com.ibm.websphere.i18n.context.*;

public class MySessionBean implements javax.ejb.SessionBean
{
   . . .
   public void setSessionContext(javax.ejb.SessionContext sessionContext)
   {
      //obtain the initial context for JNDI lookup in IBM WebSphere 4.0
      try
      {
         Properties properties = new Properties();
         properties.put("java.naming.provider.url", "iiop:///");
         initialContext = new InitialContext(properties);
      }
      catch (NamingException namingException) { /* . . . */ }

      // Obtain a reference to the UserInternationalization object.
      try
      {
         userInternationalization = (UserInternationalization)
           initialContext.lookup("java:comp/websphere/UserInternationalization");
      }
      catch (NamingException namingException) { /* . . . */ }

      invocationInternationalization =
                   userInternationalization.getInvocationInternationalization();
      . . .
   }  // end of setSessionContext

   public String m() throws RemoteException
   {
      Locale[] locales = invocationInternationalization.getLocales();
      Locale invocationLocale;
      // if ResourceBundle exists for locales[1] set invocationLocale to locales[1]
      // otherwise set invocationLocale to locales[2]
      // use invocationLocale to perform locale sensitive operations
      . . .
   } // end of m

   private InitialContext initialContext = null;
   private UserInternationalization userInternationalization = null;
   private InvocationInternationalization invocationInternationalization = null;
}
```

5. Conclusions The present paper described internationalization service and illustrated its use in the context of J2EE environments. Though not presently implemented, the concept is equally applicable and easily implementable in the domain of CORBA business objects. In fact, we use CORBA IDL definitions and the IIOP protocol for marshalling, propagating, and unmarshalling the internationalization contexts in all remote invocations. Internationalization service(context) closely parallel the transaction service(context) [15], and the security service(context) [16] as specified by OMG and the J2EE inter-operability model [20].

The classical XPG4 [26] localization model widely followed in the C/C++ CORBA business objects will not be able to do much with the propagated internationalization contexts. A `setlocale()` call changes the locale of the entire process, thereby changing the locale of all the executing threads. The ICU class library [9] is a natural fit for the purpose of localization using internationalization contexts inside C/C++

CORBA business objects. In the C++ CORBA world, the ANSI C++ class library [12, 18] can be another choice, though presently ANSI C++ does not include [12] any time zone support.
Certain applications may need more than one locale for business processing. For example, one locale is used for accessing translated resources, another one is used for data formatting, still another locale may be used for collating sequences of strings. The locale array present in the internationalization context interfaces can be used for this type of applications. Though the users are encouraged to use the ISO-639 two-character language codes, and the ISO-3166 two-character country codes for constructing locales, the internationalization service does not prevent anybody from constructing locales using arbitrary strings. Multi-locale applications may decide to agree on internal application-level protocols to separate different categories of locales passed in the locale array. As an example, an application client may intentionally construct 'demarcation locales' using specialized marker strings (something like `LOC_FORMAT`, `LC_COLLATE`, etc.) and place them in proper positions in the locale array of `InvocationInternationalization` before invoking server-side methods.

To our knowledge, [10] is the only attempt to provide an infrastructure for internationalization in a distributed environment. The scope of the system described in [10] is rather limited. The infrastructure is applicable only to the Java™ 2 Standard Edition (J2SE) applications running under the Tivoli® system management framework. In contrast to our approach, which intends to cover the entire J2EE domain, the approach of [10] is not extensible and can be considered as a static variant of the conventional parameter passing approach. Various locale and time zone parameters are stored statically as application preferences in a Tivoli database. However, the JDK-ICU4J wrapper classes of [10] can simplify programming when used in conjunction with the invocation internationalization context in J2EE environments.

The present version of the internationalization service is implemented in IBM WebSphere Enterprise Edition, release 4.0, which is compliant with the J2EE 1.2 specifications [24]. In the future, IBM intends to release a J2EE 1.3 specification [22] compliant version of the WebSphere product. IBM also plans to enhance the existing internationalization service, and incorporate the enhancements in the enterprise edition or some other version of the WebSphere product. The enhanced version of internationalization service intends to provide precise semantics of the internationalization contexts for message-driven beans, and for the local home and local component interfaces introduced in the EJB 2.0 specifications [20]. It also intends to introduce and formalize the notion of container driven internationalization context management policies [1] expressible in the form of XML deployment descriptors. The deployment descriptors for internationalization context may allow server-side components to specify invocation internationalization contexts, which can be different from the inbound caller internationalization contexts. The deployment descriptors for internationalization context may also allow a Servlet or an EJB developer to programmatically alter the invocation internationalization contexts. IBM also intends to extend the concept of distributed internationalization in the emerging domain of enterprise WebServices [6].

IBM intends to propose the formal specification for the internationalization service as applicable to the J2EE environment. IBM's initial proposal [2] of internationalization service has been accepted for development in the Java Community Process.

Acknowledgements
David Zavala of IBM, Rochester, USA provided the implementation of internationalization service. Ute Schuerfeld of IBM, Germany and David Zavala participated in many stimulating discussions. Kentaroh Noji of IBM, Japan provided the CJK ideograph and the Hangul syllable. Betsy Baartman of IBM, Rochester, USA provided numerous suggestions for improving the quality of presentation.

References
1. Banerjee D., et al. On the Formal Management Policies of Internationalization Context. In preparation.
2. Banerjee D. JSR 150: Internationalization Service for J2EE. http://jcp.org/jsr/detail/150.jsp.
3. Chan P., Lee R., and Kramer D. *The Java Class Libraries, Second Edition, Volume 1*, Addison-Wesley, Reading, Mass., 1998.

4. Deitsch A. and Czarnecki, D. *Java Internationalization*, O'Reilly, Sebastopol, CA, Mar. 2001.
5. Eddon, G. and Eddon, H. *Inside Distributed COM*, Microsoft Press, Redmond, WA, 1998.
6. Ferguson, D. JSR 109: Implementing Enterprise Web Services. http://jcp.og/jsr/detail/109.jsp.
7. 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 20*68, Jan. 1997.
8. IBM Corporation. IBM WebSphere Application Server Enterprise Edition, Version 4.0, July 2001.
9. IBM Corporation. International Components for Unicode, Version 1.8.1. http://oss.software.ibm.com/icu, June 2001.
10. Ishimoto K. and Higuchi S. Preference Object-Based Internationalization for Distributed Application Framework in Java. *18th International Unicode Conference*, Hong Kong, Apr. 2001.
11. Kano, N. *Developing International Software for Windows 95 and Windows NT*, Microsoft Press, Redmond, WA, 1995.
12. Myers, N. C. The Standard C++ Locale. http://www.cantrip.org/locale.html.
13. Microsoft Corporation. MSDN CD-ROM, July 2001.
14. Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.5*, Sep. 2001.
15. Object Management Group. *Transaction Service Specification, Version 1.2*, May 2001.
16. Object Management Group. *Security Service Specification, Version 1.7*, Mar. 2001.
17. Object Management Group. *Java to IDL Mapping*. June 1999.
18. Plaguer, P. J. *The Draft Standard C++ Library*, Prentice Hall, EngleWood Cliffs, NJ, 1995.
19. Schmitt, D. A. *International Programming for Microsoft Windows*, Microsoft Press, Redmond, WA, 2000.
20. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0, Final Release*, Palo Alto, CA, Aug. 2001.
21. Sun Microsystems. *Java Servlet Specifications, Version 2.3, Final Release*, Palo Alto, CA, Aug. 2001.
22. Sun Microsystems. *Java2 Platform Enterprise Edition Specifications, v1.3, Final Release*, Palo Alto, CA, July 2001.
23. Sun Microsystems. *J2EE Connector Architecture Specification, JSR 016, Version 1.0, Final Release*, Palo Alto, CA, July 2001.
24. Sun Microsystems. *Java2 Platform Enterprise Edition Specifications, Version 1.2*, Palo Alto, CA, Dec. 1999.
25. The Unicode Consortium. *The Unicode Standard, Version 3.0*, Addison-Wesley, Reading, Mass., 2000.
26. X/Open Group. *Internationalisation Guide, Version 2*, Prentice Hall, EngleWood Cliffs, NJ, 1994.

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:
WebSphere System/390 AIX Tivoli IBM

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.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation 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.