Implementing CORBA Internationalization Features

KUROSAKA Teruhiko [1], Internationalization Architect, IONA Technologies, PLC

Abstract

CORBA is a time-proven, platform-neutral, cross-language, application integration technology. Contrary to a popular perception that CORBA is soon to be replaced by Web Services, it has advantages over the newer technologies, and it still plays important roles in large enterprise IT environments.
This paper explains key internationalization features of CORBA, and describes how IONA implemented them in its product, and what challenges it faced.

1. Introduction to CORBA

The Common Object Request Broker Architecture (CORBA) is a distributed object invocation model and specification developed and maintained by the Object Management Group (OMG), an industry consortium. CORBA is platform independent and supports most major programming languages including C, C++, PL/1, COBOL, Python, and Java. In CORBA, the calling program does not need to be aware where the remote object is located or what programming language the object is written in. CORBA implementations from different vendors can interoperate with each other.

The core of any CORBA-based system, the Object Request Broker (ORB), hides the low-level details of platform-specific networking interfaces, allowing developers to focus on solving the problems specific to their application domains rather than having to build their own distributed computing infrastructures.

The interfaces of CORBA objects are described in the OMG Interface Definition Language (IDL). IDL, like programming languages, has a set of primitive datatypes. Two of such IDL datatypes that are of concern here are *char* and *string*. They correspond to the `char` and `char*` datatypes of the C and C++ languages.

Refer to [1] for the full specification of the latest CORBA specification.

CORBA is often seen as an old technology that is to be replaced by the newer technology, specifically SOAP and Web Services. However, CORBA and Web Services are different; each has its own strengths and weakness, and they in fact complement each other. The Appendix shows a comparison between the two technologies.

2. Internationalization Features in CORBA

2.1. History

CORBA version 2.0 did not have internationalization features at all. On the contrary, it even specified ISO 8859-1 (also known as Latin 1) be the only code set[2] that can be

[1] In the Japanese, SURNAME - given name order. The author prefers to be called Kuro-san.

recognized by various CORBA components, excluding all other code sets. The IDL datatypes *char* and *string* could carry only ISO 8859-1 characters. CORBA programmers in the "rest of the world" had to use the IDL *octet* datatype to circumvent this strong restriction. The octet datatype represents a raw 8-bit byte without any semantics attached. When two ORBs whose underlying platforms use different code sets communicate with each other, it was the responsibility of application code to carry out a necessary code set conversion to the textual data represented by octets.

CORBA 2.1 [2] changed this landscape by introducing a set of internationalization features such as:

- *wide character* (IDL datatype *wchar*) and *wide string* (*wstring*)
- Support of code set other than ISO 8859-1
- Code Set Negotiation
- Automatic code set conversion

Subsequent versions of CORBA up to the latest version, 3.0, include only minor modifications to this internationalization feature set.

2.3. The Wide Datatypes

CORBA 2.1 introduced two new IDL datatypes, *wchar* and *wstring*.

The wchar datatype is analogous to the `wchar_t` datatype of the C and C++ languages, which is basically a fixed-size storage unit that is big enough to hold a character[3] of any byte size of the supported code sets in a uniform manner. This makes it easier to write a program that needs to deal with varying length character encodings, which are common in East Asian language (namely Chinese, Japanese and Korean) computing. In fact, the C and C++ *language bindings* specifies wchar be represented by `wchar_t`. The wchar data is also called *wide character*. (To contrast with the wide character, the traditional char, the single-byte storage unit, is often called *narrow* character.) The actual byte size of a wide character is platform-dependent. Although commonly it is 2 bytes (on Windows NT, for example) or 4 bytes (Solaris) in size, it can be just 1 byte if the platform chooses not to support multibyte code sets at all.

The wstring datatype is a string of wchars. Its binding for the C and C++ languages is `wchar_t *`.

Because CORBA 2.1 introduced wchar and wstring, there has been a *common misconception* that one has to use wchar or wstring in order to support multibyte code sets as used in East Asian language computing. This is incorrect. The fact is that the narrow string can also be used to pass textual data that includes multibyte characters. It is correct, on the other hand, that a narrow char cannot be used to express a multibyte character, because the size of a char is defined to be exactly 1 octet, i.e. a byte of 8 bits.

---

[2] *Code set*, short for Coded Character Set, will be used through out this paper, rather than other terminologies of similar concept such as *encodings*, and *code pages*, in order to align with CORBA.
[3] To be exact, it is rather a Coded Character Element. More than one CC-Elements may represent a character in the ordinary sense, if a CC-Element is a combination mark, or an element of a character.

A narrow string, however, can still be used for this purpose. Use of the narrow string would probably provide a better transition story for existing applications, and can even yield a better performance in some cases than using wstring.

2.4. Code Set Support

With CORBA 2.1, code set support has been enhanced. It has become possible to use code sets other than ISO 8859-1 in char, string, wchar and wstring datatypes. For chars and strings, only *byte-oriented* code sets (i.e. no NUL bytes) can be used while for wchars and wstrings, there are no such restrictions.

This necessitates introduction of a way to uniquely identify a code set. CORBA 2.1 uses the numerical code set ids found in the OSF[4] Code Set Registry for this purpose. The latest version of OSF Code Set Registry [4] contains191 code sets. Table 1 shows sample code set ids.

Table 1. OSF Code Set Ids

Code Set Name ("Short Description")	*Code Set Id ("Registered Value")*
ISO 8859-1:1987	0x00010001
UCS-2, Level 1	0x00010100
UCS-4, Level 1	0x00010104
X/Open UTF-8	0x05010001
JIS eucJP	0x00030010
OSF Japanese SJIS-1	0x05000011

The conversion of code sets between the client and the server that use different code sets is automatically done within the ORB "under the hood." CORBA application programmers generally do not need to be aware of the code set the application uses when coding.

2.5 Transmission Code Set and Code Set Negotiation

Because the goal of CORBA is "integration of a wide variety of object systems" (a quote from Section 2 of CORBA 3.0 specification), we cannot assume that the client and server systems use a same code set. Even for English-language applications, the client may be running on an open system using ISO 8859-1 code set while the server may be running on a mainframe using EBCDIC code set.

Before the two ORBs talk, they must agree on the code set they will use between them. This code set is called *Transmission Code Set*, or TCS, in CORBA.

The TCS is dynamically determined per connection using an algorithm specified in CORBA 3.0 Section 13.10.2.6, from Compatible Code Sets, and Native Code Sets.

[4] OSF stands for Open Software Foundation, which is now The Open Group.

The *Native Code Set* is a code set that the ORB uses "natively," which depends on both the OS platform and the programming language. A CORBA application has two NCSs, one for the narrow char/string interface (referred as *NCS-C*) and the other for the wchar/wstring interface (*NCS-W*). They could be the same or different. The concrete way of determining the native code set is implementation dependent; the CORBA specification only suggests some possible ways.

On most open systems, the NCS-C for C++ language applications in the English locale is ISO 8859-1. NCS-W in the same settings is considered to be one of Unicode variants (i.e. UTF-16, UCS-2 or UCS-4), because a `wchar_t` normally holds an ISO 8859-1 character with zero padded, which is the Unicode code point for that character.

For Java language applications, NCS-W would be UTF-16, as the language specification vaguely specifies a two-byte form of Unicode, and the actual implementation uses UTF-16. The choice of NCS-N is somewhat artificial because Java does not really have a narrow character type as in char of C++, but UTF-8 would be a natural choice as it can be converted to and from UTF-16 algorithmically without loss of data (i.e., any character that can be expressed in UTF-8 can be expressed in UTF-16 and vice versa).

Conversion Code Sets are code sets that an ORB supports in addition to its NCS. EBCDIC might be a typical example of CCS for the C++ ORB that runs on an open system in the English locale. The CCS can be an empty set. As is the case for the NCS, there are actually two sets of CCS, one for the narrow char/string and the other for the wide wchar/wstring datatypes.

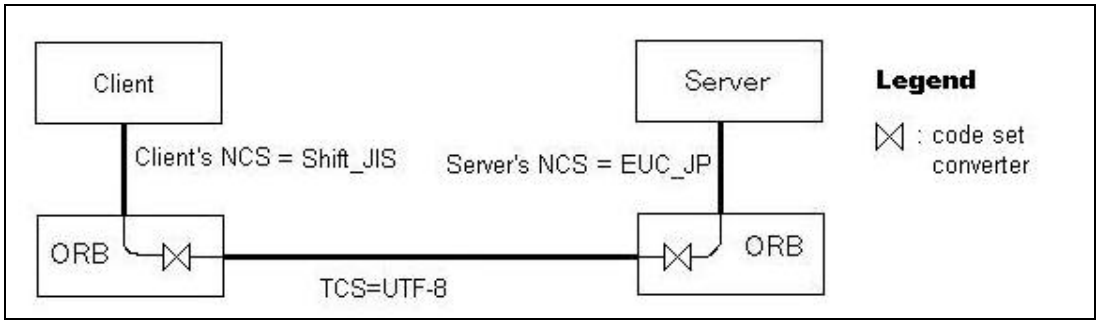
For each of the narrow char/string interface and the wchar/wstring interface, the NCS is determined using the algorithm illustrated below. Here, C*x*CS means Client's *x*CS and S*x*CS means Server's *x*CS:

> No SNCS specified ? TCS = ISO 8859-1 # Pre-2.1 compatibility.
> CNCS = SNCS ? TCS = CNCS # Same NCS, no problems!
> SNCS $\in$ CCCS ? TCS = SNCS # Client knows Server's NCS.
> CNCS $\in$ SCCS ? TCS = CNCS # Server knows Client's NCS.
> CCCS $\cap$ SCCS $\neq \emptyset$? TCS = first common code sets
> CNCS & SNCS are compatible ? TCS-C = UTF-8 or TCS-W = UTF-16
> If all fail, raise CODESET_INCOMPATIBLE exception

This algorithm is run in the client ORB. The server first publishes its NCS and CCS for both narrow and wide char/string interfaces inside the *object reference*, which represents the remote object. Once the TCS for has been selected using this algorithm in the client ORB, the TCS is sent to the server in a *service context* that is passed along with the request message

As you can imagine, the resulting TCS could be different from CNCS and from SNCS. This means code set conversions may need to be performed on both sides of the connection within both ORBs, as seen in Figure 1.

Figure 1. Client and Server Can Talk Using TCS Different Than Both NCSs



3. Implementation in ASP 6.0

3.1. IONA Application Server Platform

The IONA Application Server Platform (ASP) is a part of the Orbix family of products, a well-known CORBA platform. It combines the traditional CORBA platform together with a Java 2 Enterprise Edition (J2EE) application server and support for Web Services. It is available for variety of platforms including Windows NT/2000, Solaris, HP/UX, AIX and Linux. (For mainframes, a separate product, the ASP Mainframe Edition, is available. This product is outside of the effort described on this paper.)

ASP supports two programming languages, C++ and Java, each with their own separate ORB implementations

ASP version 5.1 was released in May 2002. It conforms to CORBA 2.4 [3], which includes the internationalization features described earlier. The implementation of the internationalization features in ASP 5.1, however, is very minimal; only ISO 8859-1, and various forms of Unicode are supported. Because no legacy code sets other than ISO 8859-1 are supported. CORBA programmers in the "rest of the world" still have to resort to using the octet datatype, doing all code set conversions necessary inside the application code.

In order to give the full power of CORBA to developers and users worldwide, IONA expanded the code set support to include all popularly used code sets in the major markets it serves in its latest major release of ASP, version 6.0. This required some major changes in the architecture of the product. This effort started in early 2002, and completed in the release of Service Pack 1 of ASP 6.0 in February 2003. ASP 6.0 also conforms to CORBA 2.4.

The rest of the paper describes this experience of implementing the CORBA internationalization features in ASP 6.0, Service Pack 1.

3.2. Selection of Code Set Converters

Because the TCS and the NCS can differ, a code set conversion must be performed inside each ORB.

OSF Code Set Registry has 191 code sets. Removing obscure, unclear, of rareuse, or seemingly duplicated code sets from the registry, we still have more than 100 code sets. The number of pair-wise combinations of these code sets exceeds 10000. It was impractical for us to write code set converters from scratch. Instead, we decided to license existing code set converters form outside sources.

For the Java ORB, the choice was obvious: use the built-in `java.io.*` of the standard JDK (Java Developer Kit) from Sun Microsystems, Inc. (or from other vendors for some platforms).

For the C++ ORB, we considered several alternatives, and we have chosen the converter technology available from IBM's International Components for Unicode (ICU)[4] open-source project because it has a large number of supported code sets, it is actively maintained, and it is licensed free of charge. As its name implies, it is a Unicode-based technology, and all conversion is done between a code set and Unicode. Therefore, conversion between two legacy code sets must be done via Unicode. This is a potential source of performance degradation, but in practice such degradation is not noticeable. In its standard distribution, ICU has the Unicode converters for 236 code sets, and more code set converters are available as optional modules.

3.3. Code Set Id and Name Mapping

One of the difficulties in internationalization is the lack of widely accepted standards in naming code sets, or similar concepts known as “encodings,” “charsets,” or “code pages.”
Although charset names in the IANA Charset Registry[6] are widely accepted by the Internet community, the underlying systems tend to employ different code set naming schemes.
Java started with its own naming scheme of character encodings and is migrating to IANA Charset Registry. For example, the code set used in Japanese DOS (prior to Windows), and now a part of the national JIS standard X 0208, has the Java-only historical name “SJIS” and the IANA name “Shift_JIS”[5].

Similarly, ICU has its own registry of code set names and its own naming convention, but it also recognizes names from other naming schema, including IANA and Java. It seems ICU lacks a code set converter that supports Shift_JIS directly. Shift_JIS is treated as equivalent to the code set named “ibm-943_P14A-1999.”

As mentioned earlier, CORBA uses the numeric code set ids in OSF Registry to identify a code set. OSF Registry is unique among all code set naming schema in that it uses

[5] To complicate matters, "Shift_JIS" was mapped to "MS932", an extension of Shift_JIS proper for Microsoft Windows, in JDK 1.3.x and 1.4.0, causing incompatible Unicode mapping. This has been fixed in JDK 1.4.1.

numerical ids, rather than names. (Although the Short Description field in the registry tends to have a name-like strings such as "OSF Japanese SJIS-1," their names are not easy to use because in some entries, it is not clear where the name ends and where the description starts, and also because these names have spaces in them.) The numeric id for the Japanese DOS code set in the OSF Code Set Registry is 0x05000011.

CORBA is based on the OSF Code Set Registry, but neither ICU nor JDK recognizes the numerical OSF code set ids. Therefore, we need to convert the OSF code set ids to one of the encoding schema ICU or JDK understands.

3.4. Implementation of Code Set Negotiation

3.4.1. Implementation Decisions

Implementation of the algorithm itself was straightforward except for the following two decisions:

1. How do we determine the NCS and CCS?
2. How do we determine whether two code sets are compatible?

3.4.2. Determining NCS and CCS

For the C++ ORB, we had a problem that there was no widely accepted standard API that can be used to get the code set name or id. The closest API that we could use was the ANSI-C standard `setlocale()`, which returns a locale name.

We could build a table that maps a locale name to a code set id. But having such table would not be an easy task because we would have to make a table for each platform we support, for which we would need to find a reliable source of information. And then we would have to update them as new locales become available.

Given the number of OS platforms that we support and the time and the development resources available, we concluded it was not feasible at the time. Instead, we decided to make the NCS and CCS configuration variable, to be set by the administrator at the deployment time. Code 1 shows the four configuration variables and their sample settings. Configuration variables are set in a file, or in a configuration repository, a DNS-like name-value database for the configuration variables within an ASP configuration domain.

Code 1. Configuration Variables

```
plugins:codeset:char:ncs = "0x00010001"; # ISO 8859-1
plugins:codeset:char:ccs = ["0x05010001", "0x10020025"]; # UTF-8, EBCDIC(IBM-037)
plubins:codeset:wchar:ncs = "0x00010001"; # ISO 8859-1
plugins:codeset:wchar:ccs = ["0x00010109";]; # UTF-16
```

For the Java ORB, we decided to implement a defaulting mechanism so that Java-written CORBA application can be run reasonably well without effort of manual configuration. This was possible because:

1. Java always uses UTF-16 as internal representation of its char and String[6].
2. Java run time provides the default external file encoding information.

The default value of NCS-W, the NCS for the wide char/string, is fixed to be UTF-16, just because it is really the native encoding of Java.

The default value of NCS-C, the NCS for the narrow char/string, is determined in two steps. If the Java run-time system property `file.encoding` indicates that the external file encoding is US-ASCI, ISO 8859-1, or Windows Code Page 1252, we set the default NCS-C to ISO 8859-1. Otherwise, NCS-C is set to UTF-8. UTF-8 is a natural choice because conversion from UTF-16 can be done algorithmically without any data loss. ISO 8859-1 is used as a special case default value for a performance reason, which we will discuss later.

The default setting of CCS-N is, as a principle, the file encoding plus ISO 8859-1. Because ISO 8859-1 has been the only code set supported by CORBA before CORBA 2.1, many ORBs use it as their NCS-C, including previous versions of IONA's ASP and other ORBs from different vendors. Also, CORBA services such as the Naming Service use ISO 8859-1 as their NCS-C. Without having ISO 8859-1 in CCS-N, CORBA applications would not be able to use these services, or to communicate with remote objects (servers) that run on the ORBs of different vendors, or on older versions of ASP.

If NCS-C is ISO 8859-1 itself, we use UTF-8 as CCS-C. We are merely swapping the two code sets in this case.

As a very special case, if the file encoding is either EUC-JP or Shift_JIS, CCS-C is set to include both of these code sets with the file encoding being first, in addition to ISO 8859-1. This special case is to accommodate a common situation in Japan where the client on a Windows platform uses Shift_JIS while the server on a Unix platform uses EUC-JP.

The default setting of CCS-W is, as a principle, UCS-2 and the file encoding. UCS-2 is used because some vendors consider that Java uses UCS-2 and their ORBs might advertise UCS-2 as NCS-W. The file encoding is there for the interoperability with the C++ ORB running on a platform that doesn't use Unicode as `wchar_t` values, but uses a proprietary value, typically made by shifting and shuffling bits of the byte-oriented form of a multibyte character. This is quite common among Unix platforms running in traditional locales that do not use UTF-8. Because CORBA prohibits use of such proprietary values for `wchar_t`, if `wchar_t` is not Unicode based, it must be converted to something standard. The only portable way to achieve this is to convert `wchar_t` (or `wchar_t *`) to its original multibyte form of the file encoding, i.e. NCS-C first, using the ANSI C standard function `wctomb()` or `wcstombs()`, then converting the resulting character (string) in NCS-C again to the TCS if necessary.

[6] Java Language Spec, which was written when Unicode was 16 bits, states that it uses Unicode in the char datatype and the String class. The JDK implementation assumes UTF-16.

If NCS-C is ISO 8859-1, we have only UCS-2 in CCS-W because that seemed to be the industry practice.

If NCS-C is either EUC-JP or Shift_JIS, we add both encodings, because of the same reason we had for CCS-C.

Table 2: Java ORB defaults for the Native Code Set and Conversion Code Sets for Narrow Char/String

File Encoding of the Locale	Default NCS-C	Default CCS-C
US-ASCII, ISO 8859-1, MS CP-1292	ISO 8859-1	UTF-8
Shift_JS	UTF-8	Shift_JIS, EUC-JP, ISO 8859-1
EUC-JP	UTF-8	EUC-JP, Shift_JIS, ISO 8859-1
Other	UTF-8	*File Encoding*, ISO 8859-1

Table 3: Java ORB defaults for the Native Code Set and Conversion Code Sets for Wide Char/String

File Encoding of the Locale	Default NCS-W	Default CCS-W
US-ASCII, ISO 8859-1, MS CP-1292	UTF-16	UCS-2
Shift_JS	UTF-16	UCS-2, Shift_JIS, EUC-JP
EUC-JP	UTF-16	UCS-2, EUC-JP, Shift_JIS
Other	UTF-16	UCS-2, *File Encoding*

Note that all of these default values can be overridden by specifying new values for the appropriate configuration variables in the configuration domain.

3.4.3. Compatibility of Code Sets

The CSN algorithm in the CORBA specification says that NCS should fall back to UTF-8 for NCS-C and UTF-16 for NCS-W, only when the client's NCS and the server's NCS are *compatible*. The CORBA specification, however, does not clearly define when two code sets are compatible; it states only that French and Korean are incompatible. (The author feels this example is improper because it is talking about the languages, not code sets.) What constitutes the compatibility between two code sets is up to implementation.

It seems natural to think that the code sets that target the same language are compatible to each other. But a closer analysis shows this is not always the case. For example, two widely used Japanese code sets, EUC-JP and Shift_JIS, might seem to be compatible. However, they cannot be truly compatible because EUC-JP can represent characters from JIS X 0212 standard while Shift_JIS cannot. On the other hand, JIS X 0212 characters are not often used in actual textual data, because most of the characters in JIS X 0212 are rarely used and because they are incompatible with Shift_JIS. So EUC-JP and Shift_JIS can be said to be "compatible enough" for real-world use. Similarly, Microsoft Code Page 932, the Windows extension of Shift_JIS, can be considered compatible with Shift_JIS, as long as the data does not include characters that appear only in Code Page 932. Thinking further, we can say EUC-KR, the Korean version of EUC, and ISO 8859-1 can be considered compatible, as long as the two CORBA system exchanges are actually limited to the ASCII range.

If we take a stand that code set compatibility means absolute compatibility with any character expressible in either code set, hardly any code sets are compatible with each

other. Even EBCDIC and US-ASCII would have to be considered incompatible, making the fallback encoding step in the CSN algorithm useless.

As this analysis shows, it does not actually make sense to talk about compatibility of two code sets outside of the context of the actual data to be handled. Yet the CSN algorithm requires the CORBA ORB vendors to implement this compatibility test. One vendor we have talked to decided to take an extreme approach: any code set is incompatible with any other code sets. With this approach, fallback to UTF-8/16 will never occur.

We decided to take a middle-of-the-road approach where:

- Latin-*n* code sets (ISO 8859-1~4, -9~10[7]) plus Windows Code Page 1252 are compatible with each other.
- UCS-2, UCS-4, UTF-8 and UTF-16 are compatible with each other
- All other combinations are incompatible

Latin-*n* code sets are considered compatible because they share many common characters with the same code values. But they are not truly compatible; conversion could fail depending on the actual data. Conversion to UCS-2 from other Unicode forms can fail if the data contain characters from outside of Basic Multilingual Plane, but that possibility is considered low.

Note that, with the proper configuration by the administrator, the fall-back step should not be executed anyway, so the difference in compatibility test semantics should not be relevant in real-world usage.

3.5. Custom Code Set Converter

Although ICU and most Java implementations support a large number of code sets, it does not cover all the existing code sets in the world. Also, there are a few code sets that are supported by ICU and Java, yet not recognizable by ORBs because they are not found in the OSF registry (and they therefore lack code set IDs). ISO 8859-15 (Latin-9; a modification of ISO 8859-1, with addition of the Euro currency symbol €, among other changes) is an example of such code sets.

In Asian market where ideographic characters are in use and the code set has a large code space, it is common to reserve an area of code set space for *private use* so that the users can add characters, symbols and logos that they use but are not found in the standard code set. Some large corporations create and maintain an intra-corporation standard for the private use characters. For the purpose of code set conversion, adding extra private characters to the existing code set means creation of a new code set, and the existing code set converter may not work. (Note some code sets, such as eucJP-open and SJIS-open, have taken the private characters into consideration in its design, and have a unified mapping between each other and to and from Unicode. In such case, the standard converters would work. If used this way, however, an additional character

[7] Newer Latin-7~9 (ISO 8859-13~15) are excluded because the OSF Code Set Registry does not have them.

will always map to a character in Unicode's Private Use Area, even if the character exists in the Unicode proper.)

To remedy the first situation and to fulfill the need to support corporate standard (extended) code sets, we have added a mechanism by which our professional support engineers can write new code set converters for the customers and plug them into the ORBs at the customer site.

The interface for custom code sets is specified in CORBA IDL. A custom code set converter must provide an implementation (written in C++, Java or both) of a converter factory that can create converter objects given the native and transmission code sets. The factory interface is shown below. It has two operations, one each for creating converter objects for narrow and wide char data:

```
// IDL (in module IT_CodeSet)
local interface ConverterFactory
{
    CharConverter create_char_converter(
        in CONV_FRAME::CodeSetId ncs,
        in CONV_FRAME::CodeSetId tcs
    ) raises (UnknownCodeSet);

    WCharConverter create_wchar_converter(
        in CONV_FRAME::CodeSetId ncs,
        in CONV_FRAME::CodeSetId tcs
    ) raises (UnknownCodeSet);
};
```

The custom converter factory object is registered with a code set registry provided by the ORB, which maintains a list of factory objects. Whenever the ORB needs to create a code set converter it uses the code set registry to iterate though its list of factory objects asking each factory in turn to create a converter given the native and transmission code set identifiers. If none of the custom converter factories returns a valid converter the default factory attempts to create the converter. The relevant interface for the code set registry is shown below:

```
// IDL (in module IT_CodeSet)
local interface Registry
{
  ...

  void register_converter_factory(
      in IT_CodeSet::ConverterFactory fact
  );

};
```

When the registry calls the `create_char_converter()` or `create_wchar_converter()` operations, the custom factory either returns a nil object reference indicating that it cannot provide a converter, or returns a valid converter object. For the `create_char_converter()` operation the factory returns an object supporting the `IT_CodeSet::CharConverter` interface shown below:

```
local interface CharConverter
{
  unsigned long max_bytes();

  boolean preserves_length();

  boolean is_byte_oriented();

  boolean is_null_conversion();

  void char_to_octet(
        in string str,
        in unsigned long srcOffset,
        in unsigned long nchars,
        inout IT_Buffer::RawData octets,
        inout unsigned long destOffset
  );

  void octet_to_char(
        in IT_Buffer::RawData octets,
        in unsigned long srcOffset,
        in unsigned long noctets,
        inout string str,
        inout unsigned long destOffset
  );
```

The operations of interest here are `char_to_octet()` and `octet_to_char()`, which do the actual conversion. The `char_to_octet()` operation is used to convert from the native code set to the transmission code set taking a segment of the input string `str` and placing the converted characters as octets into the buffer `octets`. The operation `octet_to_char()` performs the reverse operation. If a character cannot be converted, the operation must raise the standard system exception `DATA_CONVERSION`.

The `WCharConverter` provides a similar interface but for wide strings.

4. Other Issues

4.1. OSF Code Set Registry

4.1.1. Ambiguity, Inaccuracy, etc.

CORBA has been using the OSF Code Set Registry as the basis of identifying code sets. However, this registry has several issues. Because these issues are from the CORBA specification itself, these issues remain unsolved in our implementation.

One issue is that the description of each code set tends not to give enough information to distinguish it from similar code sets. For example, take a look at two variants of EUC-JP, "JIS eucJP:1993" and "JVC_eucJP." The entries in OSF Registry for these code sets read as follows:

```
start
Short Description   JIS eucJP:1993; Japanese EUC
Registered Value    0x00030010
Character Set ID(s) 0x0011:0x0080:0x0081:0x0082
Max Bytes per Character    3
Ordering Information
       ?
Comments
       Implementation of the EUC (Extended UNIX Codes) encoding
       method, with ISO 646:1991 IRV assigned to CS0, JIS X0208:1990
       assigned to CS1, JIS X0201:1976 assigned to CS2, and
       JIS X0212:1990 assigned to CS3.
end

start
Short Description   JVC_eucJP
Registered Value    0x05020001
Character Set ID(s) 0x0001:0x0080:0x0081:0x0082
Max Bytes per Character    3
Ordering Information
       See information provided before first OSF JVC entry (above).
Comments
       ?
end
```

As seen, these two entries do not give enough information about how they are different. It is not common knowledge among the Japanese engineering community. Considerable amount of effort to carefully research each of them is needed to understand the difference. OSF Registry actually lists 13 Shift_JIS variants and 10 EUC-JP variants. This makes the OSF Registry very difficult to use. It is suspected that code sets for other languages have a similar problem, though the extent of the problem seems much less than those for Japanese.

Secondary, it is not accurate. Further research on the "JVC_eucJP" code set revealed this name was a temporary name used during the standardization process of this code set, and its official name is "eucJP-open," for example.

Thirdly, it lists deprecated code sets. For example, one of the Chinese code sets found in the Registry, IBM-936 is obsolete, according to a source at IBM. IBM now uses either IBM Code Page 1381 (IBM's implementation of the standard "GB" code set standard) or Code Page 1386 ("GBK") in place of Code Page 936. But Code Page 1386 cannot be found in the Registry. This is because OSF Registry is not actively maintained. The latest update was done in 1999.

Lastly, and most importantly, the Registry lacks some important code sets such as ISO 8859-15 (Latin-9) and Microsoft Code Page 932[8] (Shift_JIS extension). In fact, no Microsoft code sets can be found in the Registry[9]. We would need to resort to using identical or similar code set from IBM, or subset code sets, for the purpose of CNS.

[8] Note that IBM-932, which is listed in the OSF Registry, is different from Microsoft Code Page 932.
[9] Microsoft and IBM share some code pages. In that case, the code set id for the IBM code page can be used for the Microsoft code page. IBM-1252 is such an example. Care must be taken, however, because an IBM code page and an Microsoft code page of the same number are not always identical.

4.1.2. Planned Solution by OMG

In order to resolve some of these issues, OMG is planning to allow the use of numeric ids (*MIBenum*s) from the IANA Charset Registry, *in addition to* the OSF code set ids[10], in a later version of the CORBA spec. This would resolve the issue of lack of new code sets such as ISO 8859-15 and GBK in the OSF Registry. This would actually be the only realistic solution since there is no other publicly maintained code set registry.

This adoption of IANA Charset Registry, however, might introduce new issues, coming from the fact that these two registries are very different in terms of granularity. The OSF Registry considers variants of EUC-JP as separate code sets while the IANA Registry considers them as one code set, and only one identifier is given to the entire EUC-JP family; no distinction can be made among the variants. When two systems announce EUC-JP and therefore they agree to use EUC-JP as TCS, they might still be incompatible, for example because of small differences between implementations, which could have been detected if OSF Registry Code Set Ids were used. In addition, there would be a problem of making two code set naming schema co-exist in the transition period. These problems are, in essence, a problem of mapping a code set id of a code set naming scheme to a concrete ICU/Java converter. Because either code set naming scheme completely satisfies all needs, they must be configurable by the customers. We would probably solve this problem by providing a sensible default mapping with ability for the customer to override the default mapping.

4.2. Performance Issue

4.2.1. Poor Performance in Java ORB Code Conversion

Initially, Java ORB used UTF-8 as the NCS-C regardless of the file encoding. A performance test on this version showed worse-than-expected performance degradation compared with the previous ORB. Analyzing this performance problem further, we found two root causes.

CORBA allows a long string to be split into multiple *GIOP* messages at arbitrary positions of multiple of 8 bytes. For a single-byte code set, double-byte code sets (e.g. UCS-2), or quadruple-byte code sets (UCS-4), this does not pause a problem because the message boundaries are always at character boundaries, and the code set conversions can be performed for each message. For variable-length code sets such as UTF-8, Shift_JIS, EUC-JP and other Han-based Asian code sets, this is a problem because a character may appear across two messages; i.e. the first byte of a double-byte character may appear at the end of the first message and the second byte at the beginning of the second message. We would need to accumulate all string fragments into a large buffer, and then perform the code set conversion at once. Allocation of a large buffer and copying operations was lowering the performance.

[10] Fortunately, number ranges of OSF code set ids and IANA charset MIBenum values do not overlap.

The second cause of the performance degradation was due to the way some classes within the JDK use the built-in converters for `String`-to-`byte[]` conversion:

1. For the Java `String` of *n* characters, it allocate *n*m* bytes where *m* is the max number of bytes per character as the converter reports.
2. It asks the converter to convert the String and leave the result in the allocated buffer. The convert returns *k*, the actual byte length of the result.
3. It allocates a new byte array of *k* bytes.
4. It copies bytes from the first buffer of *n*m* bytes to the second one of *k* bytes.
5. (The garbage collector eventually reclaims the first buffer.)

This is most problematic for UTF-16 whose converter returns 4 as the maximum bytes per character. At the first glance the maximum bytes per character[11] of UTF-16 should be 2, not 4. But the converter reports 4 because it could add a Byte-Order-Mark, which takes additional 2 bytes. As a result, conversion of a String of 100 characters means allocating 400 bytes, convert the string, allocating 200 bytes, copying 200 bytes and returning the 200 bytes.

4.2.2. An Ad Hoc Solution

Observing the fact that the performance degradation is minimum for the single-byte fixed code set, we changed the default NCS-C to ISO 8859-1 when the file encoding is US-ASCII, ISO 8859-1, or Windows Code Page 1252, as we mentioned above as a cure for our customers in Americas and Europe. We still need to find a general solution.

For the performance degradation rooted in Java built-in converters, we implemented a special handling code in the ORB for UTF-16, UCS-2 and ISO 8859-1. The special handling code would detect Byte-Order-Mark by itself. For UTF-16 and UCS-2, the "conversion" is done by just copying and potentially byte-swapping from the input data in `byte[]` to `char[]` and vice versa. For ISO 8859-1, we use the `java.lang.String`'s constructor `String(byte[] ascii, int hibyte, int offset, int count)` and its deprecated method `getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`, which we find yields rather high performance.

5. Future Plans & Hopes

This chapter describes features and enhancements that we would like to see in the future versions of our product. (Note that these should not be taken as our commitment as a company, but just as current thoughts of the author.)

5.1. Features for C++ ORB

5.1.1. High-Performance Direct Conversion

The ICU converters ASP uses is a Unicode-based converter. In order to convert from a code set A to code set B, two conversion steps must take place:

Code Set A → Unicode → Code Set B

Obviously, this is not very efficient.

[11] In this context, a surrogate character is considered a character.

Conversion between Unicode and a legacy code set, except ISO 8859-1, requires a mapping table, which can be quite large for double byte code sets. Two such mapping tables would need to be loaded into memory to facilitate conversions between the two double-byte code sets.

Some pairs of code sets are designed so that they can be converted to each other algorithmically without need for table lookup. Two commonly used Japanese code sets, EUC-JP and Shift_JIS, are such an example. Nevertheless, the current ASP implementation performs code conversion between them via Unicode, in two steps.

We would like to find a way to shortcut this two-step conversion process into a single step, in general. For commonly used pairs of code sets between which algorithmic conversion is possible, we would like to develop plug-ins for those specific pairs of code sets.

5.1.2. Default NCS/CCS Configuration

We would like to revisit the issue of providing a default mechanism for NCS and CCS configurations in a similar fashion as for Java. As noted earlier, however, this would require creating and maintaining the locale-to-encoding mapping tables, one for each platform we support. Alternatively, we might develop a Java-based configuration tool that would generate static configuration for a specific locale. Although this has an obvious limitation that the ORB must run in the fixed locale, this may be an acceptable limitation in actual uses.

5.2. Features for Java ORB

5.2.1. NIO Converters

The current implementation uses the built-in converters made for the traditional `java.io.*` and `java.lang.*` packages. The converters belong to Sun's private `sun.io` package. Information on adding new code set support is not public.

Java has another code conversion facility in its `java.nio.charset` package, starting with Sun's JDK 1.4 release, which allows addition of custom code converters through its public `java.nio.charset.spi` package,

We considered basing our implementation on the `java.nio.charset.*` converter framework, because it provides a Java standard way of adding custom converters, and it can improve performance. But we decided to base our implementation on the older and inflexible `java.io.*` framework, because (1) if we used the `java.nio.charset.*` converter framework it would narrowed our supported platform to JDK 1.4 and higher, (2) code sets converter supplied for this package were very limited in the JDK 1.4.0 release, the latest release of JDK at the time we were making this decision, and (3) our evaluation showed that performance implementation was limited to programs that make use of the NIO Socket API.

The situation may have changed since we made this decision. Code sets are equally well supported on `java.nio.charset.*` as on `java.io.*`. JDK 1.4.x install base has grown. Our ORB implementation itself may use the high performance IO framework of NIO. We are considering using the `java.nio.charset.*` conversion framework again in future releases.

5.2.2. Fundamental Performance Solution

We need to find a fundamental solution to the performance problem. This might require a close co-operation with Sun in two areas. We would propose a way to improve overall performance of JDK methods that uses the converters. We would propose additional methods to the `java.lang.String` class so that conversion of incomplete string is possible.

5.3. Features Common to Both ORBs

5.3.1. Support of IANA Charset Registry

When the next revision of CORBA is published, we will support the numerical "MIBenum" ids of IANA Charset Registry. Some kind of configuration mechanism needs to be added because a MIBenum doesn't always resolve to one converter, due to coarse granularity of the IANA Charset Registry.

5.3.2. More Code Sets

ASP 6.0 is not supporting all the code sets registered with the OSF Code Set Registry (and IANA Charset Registry). In particular, we chose to support only the plain vanilla version of Shift_JIS, named "OSF Japanese SJIS-1," among other variants, and "JIS eucJP" among many EUC-JP variants. It has come to our attention that supporting the other variants named SJIS-open (wrongly referred as JVC_SJIS in the OSF Registry) and eucJP-open (JVC_eucJP) is important due to compatibility with Windows Code Page 932. We would add these and others important code sets.

5.3.3. Symbolic Code Set Ids

Currently, NCS and CCS are configured using the OSF code set ids, which are 8 digits hexadecimal numbers excluding the hexadecimal prefix 0x. These are error-prone and difficult to read once set. Using symbolic ids, rather than numerical ids, via some kind of macro mechanism would improve the ease-of-use of the product.

6. Acknowledgement

This paper summarizes the work done by Paul Taylor, Gregor Heine and the author. Paul and Gregor educated the author on CORBA basics and internationalization features. They gave me vital information about design and implementation, reviewed draft versions of this paper, and gave suggestions.
Steve Vinoski, Chief Engineer of Product Innovation for IONA, helped me to write the overview section of CORBA and also reviewed this paper. The author would also like to mention that he learned CORBA basics from the book Steve co-authored[9].

I also thank my former manager, Ken Schwarz, and current manager, Larry Lumsden, for their support and encouragement.

Reference

[1] CORBA 3.0 (latest version), `http://www.omg.org/docs/formal/02-06-33.pdf`

[2] CORBA 2.1 (first version with internationalization features),
`http://www.omg.org/docs/formal/97-09-01.pdf`

[3] CORBA 2.4 (version that ASP 5.1 and 6.0 conform to),
`http://www.omg.org/docs/formal/99-10-07.pdf`

[4] ICU Project,
`http://oss.software.ibm.com/icu/`

[5] OSF Code Set Registry,
`ftp://ftp.opengroup.org/pub/code_set_registry/code_set_registry1.2g.txt`

[6] IANA Charset Registry, `http://www.iana.org/assignments/character-sets`

[7] Supported Encodings of Sun JDK 1.4.2,
`http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html`

[8] Standard ICU converters listed by ICU Converter Explorer,
`http://oss.software.ibm.com/cgi-bin/icu/convexp?`

[9] M. Henning and S. Vinoski, *Advanced Corba Programming with C++*, Addison Wesley Longman, Reading, Mass., 1999.

About The Author

KUROSAKA Teruhiko, or Kuro, started his quest for internationalization when he was a graduate student. His first project was to localize the SNOBOL4 language compiler. Since then, he has over 20 years of experience in internationalization and localization.

He worked for Sun Microsystems for 12 years where he led the Solaris internationalization and later managed the localization activities of Sun's software tools.

Most recently, he has joined IONA Technologies at its Santa Clara office as Internationalization Architect. He has helped developers in implementing CORBA and J2EE internationalization features for Orbix Application Server Platform 6.0. He has also been involved in internationalization effort of other products.

Kuro holds a Master of Science degree in Computer Science from Kyushu University, Japan.

Appendix. Comparison of CORBA and Web Services

This is a summary of an article titled "Object Interconnections: CORBA and XML — Part 3: SOAP and Web Services" by Douglas C. Schmidt and Steve Vinoski from C/C++ Journal. The article is available on World Wide Web at:
`http://www.cuj.com/documents/s=7990/cujcexp1910vinoski/`

	CORBA	**SOAP/Web Services**
Model	Remote object	Stateless service
Interface defined by	IDL	WSDL
Message type	Binary	XML
Message size	Small	Large
Marshalling cost	Small	Large
Message readability	Not readable	Somewhat readable
Language dependency	Independent	Independent
Platform dependency	Independent	Independent
Firewall	Usually blocked	Usually unblocked
Granularity	Finer	More coarse
Industry acceptance	Middle	High
Good for	Intra-Enterprise integration	Business-to-business integration, Intra-Enterprise integration, Service for general public