OSF DCE SIG **R. Mackey (OSF)**
Request For Comments: 23.0 **January 1993**

DCE 1.1 INTERNATIONALIZATION GUIDE

1. INTRODUCTION

This document is a brief outline of the modifications to DCE code that are necessary to meet the DCE 1.1 internationalization requirements. There are many references to [Workbook] included here. The Workbook is more comprehensive in scope and shows examples of possible I18N problems in various DCE components. This document is meant to point out the work areas which should be the focus of 1.1 development and which areas should not. All work items mentioned in this document are to be completed by technology providers unless explicitly assigned to OSF.

In short, the major goals of the work are:

(a) Separate all user visible messages to message catalogs.

  (i) Process all messages in the same manner across all DCE components.

  (ii) Use the DCE message facility APIs for all message handling.

  (iii) Follow proper I18N rules for good message text.

(b) Display time in DTS or locale-dependent format.

(c) Better code set independence.

  (i) Handle a wide variety of character/code sets where appropriate.

  (ii) Remove unnecessary limitations on code sets.

  (iii) Remove code set dependencies (e.g., references to binary codes).

  (iv) Enhance protocols where use of multibyte data does not affect connectivity.

(d) Prepare the DCE to allow multibyte data in composite namespaces (e.g., DFS).

(e) Provide enhancements to tools to facilitate the design and implementation of internationalized distributed applications.

(f) Promote and demonstrate design methods that will allow application programmers to build internationalized applications using DCE tools.

2. MESSAGING

2.1. Use Message Catalogs For User Visible Messages

Most DCE programs currently use message catalogs for some portion of the error messages that are displayed to the user. For DCE 1.1, all existing DCE programs must be modified to use message catalogs for all user-visible message text. All new programs must be designed to use message catalogs. Note that text which is part of a DEBUG message is not required to be isolated in a message catalog.

2.2. Use XPG4-based DCE Message APIs

All DCE programs will use DCE message APIs shown below to display messages.

The set of functions in this API is provided to allow all DCE services to display messages in a consistent manner while hiding the details of default message retrieval and XPG4 API usage. Each message is identified by a DCE global message identifier. A DCE message ID, represented as a 32-bit unsigned integer, includes information identifying the message catalog and the index of a message in the catalog. The integer is kept in the local format for the machine and is assumed to be transmitted via RPC to provide automatic conversion to the local integer representation. The message facility also programmatically relates default messages (compiled in arrays) to the messages in catalogs to allow messages to be available when the message catalogs are not.

The message catalogs and the default message arrays are automatically produced by a new tool called the Symbolic Message Source (SMS) compiler from a source file describing the messages. The SMS compiler, the format for the SMS file, and the resulting message catalogs and C source files are described in more detail below. In addition, the Appendix contains an annotated example of a symbolic message source file.

The definitions below are provided for completeness of this document and should not be considered manual pages for this facility. Man pages will be made available.

2.2.1. dce_msg_get_msg()

Synopsis:

```
unsigned char *dce_msg_get_msg (
                unsigned32 message_id,
                unsigned32 *status );
```

`dce_msg_get_msg()` is the message routine that is expected to be used most often by DCE programs. It opens a message catalog, extracts a message identified by a global message ID from the catalog, and returns a pointer to `malloc()`'ed space containing the message. If the message catalog is inaccessible, and there is a default message in memory, the default message is returned in the allocated space. If neither the catalog nor the default message is available, a status code string is placed in the return value.

2.2.2. dce_msg_define_msg_table()

Synopsis:

```
unsigned char *dce_msg_define_msg_table (
                dce_msg_table_t *table;
                unsigned32      count,
                unsigned32      *status );
```

This routine installs a default message table accessible by the message facility. The `count` parameter specifies the number of messages in the table. This routine is designed to be used by programs which load all messages from a catalog into memory to avoid file access overhead on message retrieval (e.g., GDS).

2.2.3. dce_msg_get_default_msg()

Synopsis:

```
unsigned char *dce_msg_get_default_msg (
                unsigned32 message_id,
                unsigned32 *status );
```

This routine takes a global message ID, and returns a pointer to static space containing a message retrieved from the default message array. If the default message is not available, it is an error.

2.2.4. dce_msg_get_cat_msg()

Synopsis:

```
unsigned char *dce_msg_get_cat_msg (
                unsigned32 message_id,
                unsigned32 *status );
```

This routine opens a message catalog, extracts a message identified by a global message ID, and returns a pointer to `malloc()`'ed space containing the message. If the message catalog is inaccessible, it is an error.

2.2.5. dce_error_inq_text()

Synopsis:

```
unsigned char *dce_error_inq_text (
                unsigned32    message_id,
                unsigend char *text;
                unsigned32    *status );
```

This routine opens a message catalog, extracts a message identified by a global message ID, and places the message in the space pointed to by `text`. If the message catalog is inaccessible, and there is a default message in memory, the default message is copied into the space passed. If neither the catalog nor the default message is available, a status code is placed in `text`. This routine existed in prior releases of DCE and has been modified to used the default message arrays. Existing programs using this facility need not be modified.

2.2.6. dce_msg_cat_open()

Synopsis:

```
dce_msg_cat_handle_t dce_msg_cat_open (
                unsigned32 message_id,
                unsigned32 *status );
```

This routine opens a message catalog identified by a message ID. The routine returns a handle to the open catalog from which messages will be extracted. This routine is intended for use by applications (like user interface programs) which display many messages from a particular catalog.

2.2.7. dce_msg_cat_get_msg()

Synopsis:

```
unsigned char *dce_msg_cat_get_msg (
                dce_msg_cat_handle_t handle,
                unsigned32           message_id,
                unsigned32           *status );
```

This routine retrieves a message from an open catalog. If the message is not available it returns `NULL`.

2.2.8. dce_msg_cat_close()

Synopsis:

```
void dce_msg_cat_close (
                dce_msg_cat_handle_t handle,
                unsigned32           *status );
```

This routine closes the catalog specified by `handle`.

2.2.9. Example

As an example of typical uses of these routines, `printf()` calls with fixed strings, such as:

```
printf("Hello, world\n");
```

should be modified to be:

```
printf(dce_msg_get_msg(dce_hello_world_id, &status));
```

Where, `dce_hello_world_id` is a 32-bit unsigned integer constant in the DCE global ID format, identifying the message catalog and the message `"Hello World\n"` in that catalog. If the catalog is not available the message `"Hello World\n"` will be extracted from an array compiled into the program.

2.3. Supply Default Text

All DCE programs must supply default message text in English. This will be handled automatically for most if not all of the messages in DCE through use of the SMS compiler and the DCE message facility.

2.4. Consistent Message Specification and Processing

To accomplish the DCE messaging requirements in a consistent and easy-to-manage way, all DCE components will be modified to define their messages in the new SMS format, use the SMS compiler to produce message catalogs and related default message arrays, and use the message facility described above.

The SMS file for each component will include the seed number for status code generation (technology and component code), the symbol name for a default text array, the names of the symbolic constants representing the status values or message identifiers, the English message text, and a description that serves as a guide to translators preparing non-English message catalogs. The description also contains information required by the Serviceability facility such as recommended action.

The file will contain all messages for a component, both those associated with status codes and those associated with pure messages like prompts.

OSF has developed an SMS compiler which takes the SMS file as input and produces four files: a message table file containing all the default messages for the component (`.c`), a header file defining the constant message identifiers (`.h`), a message source file which is automatically passed to the gencat command (`.msg`), and a document intended to be used by administrators or translators.

Shown below is a diagram depicting the generation of files from the SMS file.

```
     Symbolic Message Source File (*.sms)
              |
              V
     ------------------------
     |                      |
     | SMS processor        |
     |                      |
     ------------------------
     |       |       |       |
     V       V       V       V
  *stat.h  *msgtbl.c *.msg <msg doc>
                     |
                     V
                     gencat
                     |
                     V
                     message catalog (.cat)
```

The `*msgtbl.c` file for a component defines an array of default messages for that component. The index into the array matches the message number and, if appropriate, the status value. The message number and matching position in the array is generated automatically based on the position in the SMS file. This means that additions to the file must be made at the end of the file or else status codes and message numbers will change (resulting in a breach of protocol). This also means that adding items to the message space is adding to the DCE protocol definition and can only be done under the auspices of OSF. In other words, licensees must not add messages or change the order of messages in the SMS or the message catalogs.

2.5. Use Standard Message Formats

All components will use the standard error, notification, and audit formats defined in [RFC 24.0] and [RFC 25.0]. The audit and serviceability APIs automatically add the required information.

2.6. Remove Message Fragmentation

Remove all message fragmentation in all components. See the DCE 1.1 I18N workbook for definition and examples of fragmentation.

3. USE LOCALIZED DATE/TIME FORMAT

All non-DTS-style time will be converted to internationalized format through the use of `strftime()` as shown in the DCE 1.1 I18N workbook. Both input and output must be processed in a locale-dependent time format.

4. USER RESPONSES IN DIALOGS

Yes/No responses will be converted to use `rpmatch` only in instances where the question being responded to has been translated to the native language. If `rpmatch` is not used, the prompt should indicate appropriate responses (e.g., y/n). OSF will ship a sample `rpmatch` function in DCE which can be replaced as part of a porting effort.

5. USE SETLOCALE

All DCE programs must use setlocale as described in [Workbook].

Setlocale affects different types of DCE programs in different ways. For interactive programs, the locale value determines how characters are interpreted, allowing multibyte characters to be processed correctly by programs designed to accept input outside the portable character set. The locale value also determines how binary data are mapped to displayable characters.

For both client and server programs using the new RPC automatic conversion software (see section on RPC Interoperability), the locale determines the local character set/code set. The local code set is the set of codes used to represent characters for all character data in a process. As explained in more detail below, the server and the client may run in different locales, with different local code sets, and employ the RPC automatic conversion feature to convert the character representation in one code set to to the corresponding value in another code set.

Different application models are affected differently by value of the locale. Applications treating character data as uninterpreted byte strings rely on the part of the locale indicating the client's character representation matching that of the data recieved from the server. Otherwise the data will be displayed and processed improperly. An example of an application which deals with character data as bytes is the OSF/1 file system. The file system simply stores the file name as bytes and pays no attention to the character representation or locale of the process creating the file. On display of the file name, the process requesting the name (e.g., the `ls` command) must have its locale set such that the bytes of the file name are interpreted correctly. Otherwise, if the characters in the file name have no representation in the process's code set, the bytes are displayed as `'?'`s. This situation would result if the process creating the file used Kanji characters represented in the SJIS code set, and the listing process had a locale with ASCII as the code set.

6. CHARACTER MANIPULATION

There is code in every DCE component to handle and manipulate character data. Unfortunately, most of the character handling code was written with a bias toward character sets which are represented by single-byte codes. In fact, much of the code is written with a bias toward a single character representation (the ASCII code set). One of the goals of internationalized software is to allow the same code to process character data regardless of whether the code set representing the characters is ASCII or a multibyte code set such as SJIS. SJIS, for example, includes the Roman alphabet but also provides representations for thousands of Japanese characters. What makes handling these character data more complicated is the fact that the data is encoded in a variable number of bytes, sometimes one, sometimes more.

In order to make DCE acceptable to a worldwide market, DCE must be modified to allow users and application programmers to use the character/code set representing their native language. In some cases, there are designed limitations on the character sets which can be used. In many others, the limitations are due to bad coding practices and must be cleaned up.

The character manipulation problems in DCE code can be broken down into three general areas:

(a) Limitations on the code sets which are due to bad internationalization coding practices. Examples include hard-coded octal or hex constants for character values, arithmetic on characters, and use of code set dependent character evaluation functions and macros like `isascii()`.

(b) Hidden dependencies on a single code set such as opaque data structures in CDS attributes or other internal data structures not converted to native representation by RPC protocols.

(c) Limitations due to protocol definitions. Unnecessary restrictions are placed on data items due to the choice of data representation in protocols. (See Appendix for definition of `idl_char` and its limitations.)

6.1. Use Good Internationalized Coding Practices

[Workbook] includes an in-depth analysis of the DCE components in the area of coding practices. The code points called out for each item must be analyzed further to determine whether they cause I18N problems in the context of the application. If so, the code must be modified to use the recommended method of character handling. Work will include but will not be limited to removal of hard-coded character constants (where the value is used locally and does not represent a protocol item), dependencies on code set construction (e.g., relative position of lower to upper case characters), arithmetic on character data, and use of deprecated or code-set dependent macros. Use of `isascii()` must be replaced by a code set independent macro called `isdcepcs()`.

In general, character validation (to constrain the characters accepted on input to a command or operation) will not be removed for DCE 1.1 nor will new validation be added. This policy reflects OSF's desire to expand the character sets allowed while recognizing the limitations caused by protocols and application design.

6.2. Document or Remove Hidden Code Set Dependencies

There are instances in DCE where data is being passed across the network in data structures which are opaque or hidden from RPC automatic character representation conversion. This has both good and bad side effects. One good side effect is that the data is not limited to the PCS by the protocol used to handle the data. A bad side effect is that the representation must be the same in all instances of the data in order for it to be suitable for manipulation and comparison. One example of such data is the CDS name stored as part of the binding information in a CDS attribute. The representation of the data in the attribute (stored as bytes) is determined by the local representation of the client. OSF must document the fact that this representation issue is actually a statement of a canonical representation for name data in towers.

More problems like this exist in other DCE components. Examples are:

(a) Security code encrypts some data in its local representation, relying on the same representation to be used at the site the encrypted data is received.

(b) CDS stores attributes in opaque byte strings and makes no allowances for conversion of representation. These strings are not defined to have a canonical representation.

6.3. Character Manipulation and Protocols

DCE services exchange character data representing a wide variety of information. Prominent examples of DCE character data are CDS directory names, principal names, and file names. Other character data in DCE are less well known, and more hidden. Examples of these data include CDS names in RPC protocol towers, and fields within registry objects like `User_Full_Name`. All these types of data have a range of expected characters and representations. Some are constrained to what is called the Portable Character Set (the intersection of the characters represented in ISO646IRV and U.S. EBCDIC). Some have no limitations.

The primary reason there are character set constraints designed into certain data types in DCE is to guarantee that anyone in any part of the world can express the data contained in those data types. The ability for a user to express a name (actually get the character codes to be emitted from a keyboard), directly affects the user's ability to contact a server in DCE. Examples of data that are required for connectivity are CDS pathnames to entries used to store server binding information, principal names, and group names. Other data objects like file names, and attributes of principals do not affect the establishment of an RPC session therefore are not required to be constrained in this way.

One of the problems in DCE is that many data items that are not required for connectivity are constrained to the portable character set due to limitations in the `idl_char` data type in RPC. Application protocols defined in RPC using the `idl_char` data type have automatic conversion of character representation between ASCII and EBCDIC. This conversion is only guaranteed to be lossless when the characters being transmitted are in the PCS. This limitation in the wire protocol affects the handling of such data in a DCE application from command line input at a client through the server storage routines.

The effects of this limitation are manifold. It limits the kind of data allowed in certain data fields. It limits the processing that needs to be done on character data types with this limitations, eliminating the need for multibyte sensitivity. It relaxes the requirements on removal of code normally considered bad I18N practice, like the use of `isascii()` and similar character classification routines.

The limitation in `idl_char` and the resulting limitations in protocols in DCE require that OSF and DCE component providers reevaluate the protocols to determine whether the limitations are reasonable for a given data type. The main criteria for such a decision is whether the type is necessary for connectivity. If the type is not required for connectivity, use of `idl_char` is only recommended where the data is constrained by some other rule. For example, remote debugging interfaces which refer to internal variables or data structures are limited to PCS by ANSI C therefore the use of idl_char for these names is permitted.

While we reevaluate protocols and data types throughout DCE we must also take into consideration three factors:

(a) DCE 1.1 must be guaranteed interoperable with older versions of DCE protocols.

(b) Modifications in protocol must be made sparingly and only with a clear market justification. There is a large cost in development and maintenance as well as runtime complexity associated with any protocol change.

(c) The only way to accomplish the goal should be through a protocol change.

6.4. Name and Service Data Protocols

One of the primary goals of DCE 1.1 work in internationalization is to accommodate as many different character/code sets as possible in areas where a variety of characters are appropriate. We believe that this goal can be accomplished with no modifications to existing data types or operations in GDS, CDS, Security, Time, or RPC in DCE 1.1.

There will, however, be protocol additions in security, and conventions for using CDS attributes to adequately address limitations in the protocols for these services. In the security component, OSF will define a set of attributes using the Registry's Extended Attribute facility to handle names and other fields which should be specified in native character sets. OSF will also define standard CDS attributes to allow non-PCS characters to be used with CDS objects. These attributes will be made available to DCE applications in DCE 1.1 However, standard DCE commands and services are not planned to be modified to use the new attributes.

The DFS component will go through a major protocol change prior to the 1.0.2 release to allow it to handle names of files and directories in a wide variety character sets encoded in both single and multibyte codes. The new DFS protocol is being designed at the time of this writing. The file system is not expected to use the RPC I18N interoperability features planned for DCE 1.1. The protocol will be specified in such a way as to allow local policy (determined at porting time) to determine whether the local file system deals with a single encoding for all file system object names or whether the names are simply viewed as byte strings with no assumed encoding. The two approaches (single encoding vs. uninterpreted bytes) contrast the design choices made in the MVS file system to the OSF/1 file system. The DFS protocol will be designed to carry enough information to allow either of these methods to be chosen.

7. INTERNATIONALIZATION ENHANCEMENTS FOR RPC APPLICATION DEVELOPMENT

DCE RPC has supported automatic conversion of character representations since its inception. Unfortunately for those application programmers who use more than the Portable Character Set, RPC has only supported conversion between ASCII and EBCDIC encodings of the PCS.

Many application programmers have asked that RPC be enhanced to support conversions between other larger character/code sets. This request, however, is more complicated than it seems. The reason IDL restricts the conversion of character data to the PCS is that IDL makes a guarantee that all data on one side of RPC will be losslessly communicated to the other side. This guarantee can only be made when both sides support an encoding for the character being communicated. Since all clients and servers are required to support the PCS, IDL can make the guarantee that no data will be lost in the conversions. (This guarantee is exactly the same as that made for floating point and integer data exchanged between clients and servers using different representations of those data types.)

A natural progression of this idea would lead one to consider exchanging data from other, much larger character sets by supplying the same sort of automatic conversion. The problem here is that without specifying the character set being communicated, there is a chance large amounts of data might be lost. For example, relaxing the restrictions on character set might result in Japanese characters being sent to a process that had no representation for such characters in its local code set. Following the current model used in RPC conversion routines, the unrecognized data would be converted to a special character representing ‘‘the unknown character’’ and the data would be lost.

All is not lost, however. The problem can be broken into two parts:

(a) The character set compatibility issue (i.e., assessing whether the client and server both support the characters required to be communicated).

(b) The code set conversion problem (i.e., given that the characters transmitted can be represented at both client and server, how the data must be transmitted and converted to allow clients and servers to manipulate the data in their own local representation.)

The character set compatibility problem cannot be solved automatically. This is due to a number of complicating factors. Firstly, an application controls the range of characters it uses. (e.g., some applications are limited to the PCS no matter what codeset the application happens to be using). Secondly, this range may remain constant for all configurations of an application or may change depending on the installation or configuration. (e.g., other applications may be configured to speak Japanese or German) And lastly, the code set used by a particular instance of an application may or may not supply information about the range of characters acceptable to an application. (E.g., knowing that server is using ISO10646 doesn’t necessarily imply that the application is configured to process Chinese, Japanese, and Greek.)

An application designer develops programs with certain assumptions about constraints on the kinds of data that will be passed between clients and servers. In the DCE Security Service, for example, principal names are constrained to the PCS to guarantee worldwide connectivity. This will be true regardless of the size of the code set supported by either the client or the server; it is simply a characteristic of the design of the security service. Other applications, like an employee directory for a region, might deal with different character sets at different sites. The reason it is important to understand that there are different models for applications is that it might be attractive to make the simplifying assumption that if the RPC could know the code sets supported by both a client and a server, the runtime could make an automatic decision about the compatibility of the character sets (following the same model as import uses for protocol compatibility). The problem is that general statements about the character set cannot be made by simply knowing the codeset. The point here is that the only the application can determine whether whether the codesets at a client and server support the requisite characters for the application to function well.

Given that the character set compatibility problem is not automatically solved for all clients and servers, we need a mechanism that allows applications to provide enough information for a client to evaluate whether a server is compatible, and a mechanism that allows clients to use that information to choose servers.

7.1. Client/Server Character Set Compatibility Evaluation

DCE will provide an extension to the RPC NSI import facility to assess the compatibility of codesets at clients and servers. Servers will export code set information into the server entries in the CDS namespace. Clients will use that information coupled with the constraints of the application to judge whether the server supports the necessary range of characters for the desired session. Only binding handles for servers that were deemed compatible by an application-provided server evaluation function will be returned to the client. Applications which equate code set with character set can use a default evaluation function provided with the RPC runtime.

By isolating the compatibility issue in the server selection process (where interface and protocol compatibility issues are handled now), the code set conversion and data loss issues become very simple. In judging a particular server compatible, the client has accepted the losses associated with communications with that server. The only question that remains is whether the client (or server) receives any indication as to whether any data was lost in the conversion. The current thinking is that there would be no indication of data loss.

One benefit of the design is that the extension to NSI can be done in a general way so as to allow it to be used for other client/server compatibility tests as well. Various attributes and values would be associated with the server binding information and would be returned for evaluation in the extended import function. This approach would be useful in Security, to assess compatibility of encryption mechanisms or authentication protocols. Applications could also use this feature for their own compatibility issues.

7.2. Automatic Code Set Conversion and Communication

Once an application has accepted the extent of data losses which may occur in communication between a client and a server, all that is left is to determine what representation(s) should be used to communicate the characters across the wire. With such a wide variety of code sets in use in the world, there is no chance one program could support all possible conversions. Since different regions tend to use particular character sets and codesets, we need to support as many of the regionally popular codesets as efficiently as possible while still allowing interoperability on a universal basis. This points to supporting a configurable set of conversion routines that will differ from region to region as well as some universally available conversions.

One goal we have had in DCE is to only convert data representations where necessary. In other words, communications between processes using the same representation would require no conversion. In addition, we would like to minimize the number of conversions for cases where there is conversion necessary.

When client and server representations differ, there are four methods of communicating:

(a) Receiver makes right (RMR). The receiver of the data is responsible for converting the data from the sender's representation to its own.

(b) Client makes right (CMR). The client converts data bound for the server into the server's representation before the data is transmitted and converts from the server's representation to its local representation on receipt of data from the server.

(c) Server makes right (SMR). Like CMR only the server is responsible for all conversions.

(d) Both client and server convert to a common or universal format (U).

The RMR, CMR, and SMR approaches are equally fast because they all require only two conversions per round trip. The down side of RMR is that both client and server require conversion routines for every representation a peer process might use (although only from wire to local conversions are needed). The CMR puts the burden of conversion on the client which distributes the processing across the many client nodes instead of pushing the processing load to the server in the SMR approach. The client grows as a result of the additional conversion routines needed to allow interoperability with any server. In certain cases, the SMR approach may make more sense due to the fact that server machines are likely to be larger and faster on average and more able to take the additional burden of conversion logic and processing. The U approach (sometimes referred to as canonical) requires the least amount of code but often costs more in processing time than a direct conversion due to the fact that four conversions are required per round trip. A universal code set is apt to take more space to represent character data so transmission costs will be higher (proportional to amount of data transmitted).

As you can see, these various methods all have their pluses and minuses. For this reason we chose to adopt a method that allows policy to determine which of the above methods should be chosen in a given situation. The aim of this design is to allow DCE vendors to develop as simple or as sophisticated logic as they deem appropriate to make the tradeoff of speed versus size in their libraries and applications.

The thrust of the design is this: the server exports information about the code sets it supports to the namespace. The client uses the extended import function described above to evaluate the server. Once the server has been chosen, the binding handle for the server would have identifiers of code sets the server could accept. The first code set in the list exported to the namespace represents the ''local code set'' of the server. This means that it is the native code set of the server process and will require no conversion on receipt. This is the fastest path through the server. Other code sets listed represent the conversion logic supported by the server. When any of the supported code sets are received, the conversion logic converts from the wire code set to the local code set (the first code set in the list).

In addition to the list of conversions advertised in a server entry, all processes are required to support the conversion to and from their local code set from and to the Universal code set (some variant of ISO10646 in all likelihood). The universal code set is a very large code set that is capable of representing many characters of many different languages. This is not to say that conversion between a local code set and the universal code set is lossless. On the contrary, many proprietary code sets contain characters that have no representation in the universal code set.

After importing a binding, the client has knowledge of the local code set and supported conversion routines at the server. The client can determine what conversions it supports through a local mechanism. Based on this information, the client can decide (based on policy built into the stub code) which method (CMR, RMR, SMR, or U) described above is best.

The simplest policy a vendor could build into the stub is to choose to convert to the universal representation every time. Another policy could be to only convert to the universal code set when the client and server representations differ. Yet another could be that the client recognizes that it supports conversions for the server's local code set and the server supports conversions for the client's local code set. This is the required configuration for RMR. The client would send its local code set and request that the server reply in its local code set. It is also possible for stub code to favor SMR or CMR.

So far, we have limited our discussion of the information and logic needed to carry out these decisions to the client side. Furthermore, we have restricted the information discussed to that available from the binding handle and the local environment. It is true that the client is responsible for evaluating the server for compatibility and initiating the communication, but RPC often has character data flowing both to and from the server. Some information needs to be transmitted with the bytes representing characters to allow the client and server to process the data correctly. Therefore, all character data is transmitted with a tag identifying the code set in which the character data is encoded. To make the policy decision symmetric at both client and server, it is also necessary for the client to identify (at least) its local code set and possibly supported conversion routines. Once the server stub code has this information, it is free to choose any of the above methods for conversion. Bear in mind that the server can depend on the client supporting conversion to and from the universal code set.

7.3. Client/Server Character Exchange Protocol Rules

The rules of the character exchange protocol that guarantee interoperability are:

(a) All clients and servers support from their local code set to the Universal code set and from the Universal code set to their local code set.

(b) For input data, the client must send character data in an encoding that is understandable by the server (the local code set or a code set for which the server has a converter). The choices of encoding are listed in the exported service data, plus the universal encoding.

(c) For output data (from server to client), the client must indicate to the server a code set which should be used by the server on reply. Policy at the client dictates the identity of this code set. It may be the local code set of the client, the local code set of the server, the universal code set, or any other code set identifier that the client is able to handle (see rules on server reply). Alternatively, the client could request that the server reply in its local code set even in the absence of knowledge of the server's local code set identity by using a special RMR tag.

(d) For output operations, upon receiving the requested reply code set identifier, the server is free to choose to reply in that code set or the universal code set (as local policy dictates).

7.4. The Application Development Model

Now that we have discussed the information flow between clients and servers, questions that remain are how application programs are designed and implemented to deal with the evaluation of servers and how code set information is supplied to the RPC for transmission and use in the conversion logic.

The first step in developing an internationalized DCE application is to design the client server interface. The designer uses the DCE interface definition language to specify what information will pass ‘‘on the wire’’ between the client and the server. We are making a distinction here between the specification of the interface, which is done in the interface definition (a file with a `.idl` suffix), and specification of the stub programming interface and behavior which is done in the attribute configuration file (a file with `.acf` suffix).

The interface definition for a simple operation with characters going both in and out would look like this:

```
typedef byte my_byte;

void op_foo (
    [in]      handle_t h,
    [in]      unsigned long stag,   /* sending tag */
    [in]      unsigned long drtag,  /* desired reception tag */
    [out]     unsigned long rtag,   /* received tag */
    [in, out] unsigned long *length,
    [in]      unsigned long size,
    [in, out, size_is(size), length_is(*length)] byte data[] );
```

The interface definition has a explicit references to the tags identifying the input code set (client → server), the client's desired reply code set, and the actual reply code set (server → client). The `stag` parameter is supplied by the client and tells the server the encoding of data in the array of bytes. The `drtag` parameter, also supplied by the client, tells the server at least one encoding supported by the client so the server can (if it decides) reply in that code set. The `rtag` parameter is supplied by the server and tells the client the encoding of data in the byte array parameter from the server at the completion of the operation.

The code set identifiers are 32-bit unsigned identifiers assigned and registered by OSF. Companies having proprietary code sets will be granted ranges of values. OSF will reserve some code set identifiers and define standard code sets for use by licensees. OSF's set will include but will not be limited to UNKNOWN, RMR, ISO646IRV, EBCDIC (code page 500), T61, Universal, SJIS, and AJEC. The wire representation of the array of character data (regardless of the character set, code set, or local representation of character data at the client or the server) will be `byte`.

Thus far in the development process we have been using only functionality that is currently supported in the IDL compiler. This means that application programmers are free to define protocols using IDL with confidence that the internationalization features can be taken advantage of later. The functional enhancements to aid I18N application programmers are isolated to the attribute configuration file processing part of the IDL compiler.

With the interface definition above, the application programmer needs to specify the stub API and the stub behavior. The attribute configuration file for the interface above would look like:

```
typedef [codeset_type(<ltype>)] my_byte;

[codeset_tag_rtn(set_code_tags)] op_foo ([codeset_stag]  stag,
                                         [codeset_drtag] drtag,
                                         [codeset_rtag]  rtag);
```

Here we see the first use of the new codeset conversion features in the ACF. The first entry in the ACF identifies the local type that will be used to represent character data at site where this ACF file is used (client or server). Typical values of `ltype` would be `wchar_t` or `char`. The next attribute which can be applied to an operation or to the interface is `codeset_tag_rtn` which identifies the a function which provides the RPC stub the codeset tags to be used on the wire between clients and servers. At the client, this routine uses information associated with the binding handle, and information from the environment to determine the codesets that will be used on the wire and the conversion policy. The `codeset_stag` attribute marks the parameter that will carry the tag used to identify the codeset of `codeset_type` data going from the client to the server. This attribute is only required in operations which have input `codeset_type` arguments. The `codeset_drtag` attribute marks the parameter that will carry the tag used to identify the codeset that the client has requested the server return `codeset_type` data in output parameters. This attribute is required in all operations which have output `codeset_type` arguments. This means that even operations which would customarily be output only would be required to have at least one input parameter. The `codeset_rtag` attribute marks the parameter that will carry the tag used to identify the codeset of `codeset_type` data going from the server to the client. This attribute is only required in operations which have output `codeset_type` arguments.

The ACF feature is designed such that the same tag will be used for all codeset__type data items in the RPC. It is also permissible to have the same interface parameter serve as more than one of the `i18n_char` parameters (e.g., letting `stag` serve as both the sending tag and the desired reply tag). Practices such as these can lead to application limitations and will be documented in the *DCE Application Programming Guide*.

After the ACF is processed, the compiler generates a stub with the following API:

```
op_foo (handle_t h, unsigned long size, unsigned long *length,
        <ltype> *data);
```

The type of the data is some user-chosen data type like `wchar_t`. For input parameters, the client application programmer is required to supply a routine which when passed a parameter of `ltype` can convert it to the wire representation (an array of bytes and a tag). For output parameters, the client must supply a routine to convert from the wire to the local representation. OSF will provide standard APIs for use within these routines to accomplish the conversion of one code set to another.

At the client side, the server code set information is available through accessor functions on the handle. The local code set is available through a standard runtime API. Type conversion, character mapping, and codeset conversion will all be handled for typical applications in supplied library routines.

ACFs may be processed independently or not at all on clients or servers. As long as the interoperability rules listed above are obeyed, interoperability is guaranteed. It should be noted, however, that when the ACF functionality is not employed, the application program is responsible for all conversions outside the RPC stubs.

Servers making use of the codeset conversion feature must call an extended export function or export individual CDS code set and/or character set attributes. Clients must call an extended `rpc_ns_import_begin` function which accepts a list of attributes to be evaluated and a application-supplied evaluation function in addition to the interface spec, protocol information, and entry name parameters.

7.5. Sample Codeset Conversion Feature Application

OSF will provide an example of a DCE application using the codeset conversion feature described above.

APPENDIX A. SYMBOLIC MESSAGE FILE PROCESSING EXAMPLES

Shown here is an example of a symbolic message source file that is processed to create a header file defining message ID constants, `.c` files defining a default message array, a message catalog, and documentation for administrators:

```
#
#  @OSF_COPYRIGHT@
#
#
#  Message table for SVC routines.
#
#  HISTORY
#  $Log$
#  $EndLog$
#
component        svc
variable         svc__table
facility         dce

start
code             svc_s_ok = 0
text             "Successful completion"
explanation      "Operation performed."
action           "None required."
end
```

```
start
code            svc_s_no_memory
text            "Out of memory"
explanation     "Could not allocate memory for message table,
                string copy or other internal requirement."
action          "Buy more memory, increase swap, etc."
end

start
code            svc_s_unknown_component
text            "Unknown component"
explanation     "Attempted to find the service handle for a
                component and could not do so."
action          "Verify that component name is known or correct
                programming error."
end
```

A new sms processing has been written by OSF which processes the file shown above. The syntax of the file is shown below annotated with descriptions of each of the fields.

```
component       svc <the three letter component seed number,
                part of msg ID>
variable        svc__table <the name of the default message
                array>
facility        dce <the facility ID, part of the msg ID,
                dce | dfs>

# for each message, application programmers specify a record
start
code            svc_s_ok = 0 <the symbol used in the code for
                status val>
text            "Successful completion" <the default msg text>
explanation     "Operation performed." <Further explanation
                (admin doc)>
action          "None required." <Administrative action
                required>
end
```

APPENDIX B. DESCRIPTION OF IDL_CHAR

Any protocol defined with `idl_char` is limited to the characters common to ASCII and U.S. EBCDIC. Currently, names of files, principals, groups, CDS directories and entries, and many attributes use `idl_char` as their data type. The reason for this is that `idl_char` was designed to communicate the ‘‘concept’’ of a character across the wire to a machine which might use a different representation of that character. The two representations that are supported in NDR are ASCII and U.S. EBCDIC. When an ASCII machine communicates the character `'a'` to an EBCDIC machine, the rpc stub inspects the data representation used at the source (ASCII), recognizes that the source and destination representations don’t match and converts the ASCII `'a'` to the destination to the EBCDIC code for `'a'`. The key point to remember is that the conversion is assumed to be lossless. In other words, there is a mapping from every character/code being transmitted to a code representing the same character on the destination machine. The translation is done in the rpc stub one byte at a time since both supported code sets represent characters in a single byte.

What this all means is that for character data to be passed transparently (without an application’s knowledge of machine architecture) and correctly (if mapping occurs, it is indeed lossless) the application must limit itself to the set of characters which exist in both ASCII and EBCDIC, the Portable Character Set (PCS).

APPENDIX C. DISCARDED ALTERNATIVES IN CHARACTER HANDLING

C.1. Use of idl_char as byte

One suggestion for a solution to the PCS restriction in protocols defined with `idl_char` stems from the fact that the conversion or mapping from one code set to another only occurs when the machines have been designated to have different representations (ASCII and EBCDIC). In the case where two machines of the same type are communicating, there is no conversion done. Therefore there is no loss of data. It is because of this behavior that there is a opportunity for users of homogeneous (all ASCII or all EBCDIC) cells to transmit data outside the PCS in all protocols and data structures which are defined in `idl_char`. This means that people willing to sacrifice interoperability across ASCII and EBCDIC platforms could use any character set (even multibyte) without the RPC stubs corrupting the data.

Of course, changes would need to be made to existing implementations of DCE services to enable the greatest possible range of code sets to be handled. For example, all DCE code which processed data communicated through an `idl_char` data type would need to be free of single-byte ASCII dependencies. There would be exceptions to this rule but they would need to be minimized. The `'/'` character is one notable exception because it is used as a delimiting character in the file system, registry, and directory services. Certain services have other special characters which must be recognizable in single-byte form. For example, the security service reserves the use of `'@'` for a separator in the Kerberos V5 protocol.

Provision for special characters has the unfortunate effect of limiting the characters that can be used from various code sets. For example, the ASCII code for the `'@'` character appears as the second byte in valid multibyte codes in the SJIS code set. Therefore all characters encoded thusly would be illegal to use in the security component.

This option was abandoned for two reasons:

(a) It is architecturally opposed to the philosophy of the `idl_char` data type and the concept of transparent heterogeneous interoperability.

(b) It encourages use of characters that will lead to interoperability problems across cells with different locales. Protocols defined with `idl_char` are restricted to the portable character set to allow worldwide interoperability. Use of characters that are not universal invites interoperability problems.

C.2. Alteration of the idl_char Data Type

A fairly popular suggestion for the solution to the PCS restriction on `idl_char`-based protocols was to extend the definition of `idl_char` to allow representations other than the ASCII and EBCDIC sets currently allowed.

In this approach the `ndr_char_rep` field in the RPC header would be used to store a tag that would identify the source code set. The comparison determining whether to convert would work in the same manner as the comparison works in stubs today except the combinations of source and destination encoding would be more numerous and the results more complicated. Since the code sets would be representing different character sets, there is a possibility that there could be no mapping possible. You should note that this is the case with any multiple character set solution and is basically equivalent to the method of choice described in the main text of this document.

There are three problems that led us to discard this approach. One is that it is a basic protocol change that breaks interoperability with DCE 1.0 and DCE 1.0.1 code. Another is that the protocol difference is hidden such that clients or servers wishing to use or avoid this behavior are not able perceive any difference in servers supporting or not supporting the enhanced functionality. The third problem is that it may not be appropriate for all protocols to handle international characters.

The real interoperability problem stems from a ‘‘bug’’ in a macro defined in `rpc/sys_idl/stubbase.h`. The code is shown here:

```
#define rpc_convert_char(src_drep, dst_drep, mp, dst)\
    if (src_drep.char_rep == dst_drep.char_rep)\
        rpc_unmarshall_char(mp, dst);\
    else if (dst_drep.char_rep == ndr_c_char_ascii)\
        *((ndr_char *) &dst) = (*ndr_g_ebcdic_to_ascii)\
        [*(ndr_char *)mp];\
    else\
        *((ndr_char *) &dst) = (*ndr_g_ascii_to_ebcdic)\
        [*(ndr_char *)mp]
```

This means that currently available clients and servers would blindly map what could be multibyte data one-byte-at-a-time as if it were EBCDIC to the ASCII codes if the local character representation were ASCII. There would be no interoperability between old and new servers and clients if the new components happened to use a code set other than ASCII or EBCDIC.

The interoperability problem caused by the introduction of hidden changes in an IDL primitive type is unacceptable.

REFERENCES

[IBMI18N] ‘‘I18N of AIX Software — A Programmers Guide’’, IBM Order #SC-23-2431-00.

[RFC 24.0] R. Salz, ‘‘DCE 1.1 Serviceability Proposal’’, November 1992.

[RFC 25.0] E. McDermott ‘‘DCE Auditing Design and Strategy’’, December 1992.

[Workbook] T. Ogura, ‘‘DCE 1.1 I18N Workbook’’, DCE Document, September 1992.

AUTHOR’S ADDRESS

Dick Mackey
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: dmackey@osf.org
Telephone: +1-617-621-8924