OSF DCE SIG **M. (Mori) Romagna (OSF)**
Request for Comments: 41.2 **R. Mackey (OSF)**
November 1994

RPC RUNTIME SUPPORT FOR I18N CHARACTERS — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

RPC runtime library enhancements are required to support character conversion features being added to the IDL compiler. Together, the IDL improvements and runtime features provide applications the ability to communicate character data between clients and servers using different representations for their character data. This feature allows data to be passed to RPC stubs and automatically converted into an appropriate representation before it is received by the peer application. The runtime enhancements include functions that provide specific client/server code set management support such as conversion, evaluation of character and code set compatibility, and interfaces to local operating system code set management functions.

DCE 1.0.x only provides support for transparent conversion of character representation (**code sets**) within the intersection of characters that are supported in the ASCII and U.S. EBCDIC code sets (the DCE Portable Character Set (**PCS**)). Applications using this feature must explicitly identify character data as `char` or `idl_char` in the interface definitions. The RPC runtime is compiled with the code set employed at a given host, just as the information about endianess and floating point format is. The application supplies character data through this type and the stub at the receiving machine inspects the encoding information and converts to the local format, if necessary.

The limit of support to ASCII and U.S. EBCDIC is adequate for some applications, but falls short of applications that need to be able to transmit and process other **character sets** (e.g., European, Chinese, Korean, Japanese, etc.). An added complication is that character sets may be represented by a variety of code sets (e.g., ISO 8859-1, pc850, ROMAN8, SJIS, and eucJP, etc.). For example, *a-acute* is represented `0xe1` in ISO 8859-1, `0xa0` in IBM's pc850, and `0xc4` in HP's ROMAN8. ISO 8859-1, IBM's pc850, and HP's ROMAN8 all encode the Latin-1 character set. Another example is the Japanese character for *one* (in kanji), which is represented as `0xB0EC` in eucJP, and can be represented as `0x88EA` in SJIS. So even though the character set is same, there is a need to convert from one code set to another. In fact, it is very common that one code set is represented differently from host to host in a **heterogeneous environments**, where several vendors' machines are connected together.

Prior to DCE 1.1, applications operating with data other than ASCII and U.S. EBCDIC were either restricted to **homogeneous code set environments** where all the character data representations were the same or were forced to implement their own code set recognition and conversion mechanism.

To aid programmers in developing applications that will behave correctly in both homogeneous and heterogeneous environment, DCE release 1.1 will include RPC runtime library enhancements and IDL compiler improvements to allow automatic conversion between various representations of characters.

The IDL compiler will be improved to have a new ACF feature. This feature, allows application developers to specify automatic conversion between different code sets.

RPC runtime library will be enhanced to include:

(a) Code set advertisement and retrieval routines.

(b) Character and code set compatibility evaluation routines.

(c) Code set conversion routines.

(d) Local operating system and OSF Character and Code set Registry access routines.

See Functional Definition about the OSF Character and Code set Registry.

(e) Buffer sizing routines for marshalling and unmarshalling.

Using these improvements and enhancements, distributed applications can be written with code set transparency. Code set transparency means that application programs do not need to be aware of how their character data are encoded. Once an application sets a **locale**, the code set information will be retrieved internally. Code set compatibility evaluation will determine the binding between client and server. Then the code set is automatically converted to make a connection possible (between client and server), when necessary.

OSF's goal was to design these new features to be as flexible as possible. We didn't want to hard code features like a character and code set evaluation logic or a **conversion model** between client and server. Rather, emphasis was put upon a framework which allows these features to be more easily tailored to the needs of each application. For example, customized evaluation logic, or the selection of a particular conversion model can be easily implemented.

RPC runtime library enhancement will include sample code set compatibility evaluation routines. When these sample routines fall short of the needs of particular application, they can be replaced by application developers.

See FUNCTIONAL DEFINITION for the details of these enhancements.

1.1. Changes Since Last Publication

This is the last change for this RFC.

(a) The OSF code set registry is used to gather the information about the supported code sets rather than from the local operating system.

(b) Added the idea and implementation of `intermediate` code set conversion method.

(c) Modified signatures of some of the APIs. These changes were necessary to deal with the slight design modifications as well as for the implementation needs.

(d) Added new APIs (`rpc_cs_binding_set_tags` and `rpc_rgy_get_max_bytes`) which were not defined, and identified as necessary.

(e) The interface to the local operating system section was modified. This is because the OSF character and code set registry is used to determine the available local code sets. The OSF character and code set registry is required to be installed in each host within the internationalized cell.

(f) Updated the customized routines development section to include more sample codes.

(g) Updated the restrictions and limitations section to include the problem relating to `wchar_t` data type usage.

(h) Updated open issues section to indicate `dcecp` interface to import/export the supported code sets information to NSI is not available for the 1.1 release.

2. TERMINOLOGY

(a) **Character Set**

A group of characters without any associated encoding. Examples of character sets are the English alphabet, Japanese Kanji, and the characters needed to write European languages. [RFC 27.0]

(b) **Coded Character Set, or Code set**

A mapping of the members of a character set to specific numeric code values. Examples include ASCII, ISO 8859-1 (Latin-1), JIS X0208 (Japanese Kanji). [RFC 27.0]

(c) **Conversion Method**

Conversion method is the way that one code set is converted to another code set. There are four methods of conversion.

(i) Receiver makes it right (RMIR). The receiver of the data is responsible for converting the data from the sender's representation to its own. This is the method that current ASCII and U.S. EBCDIC automatic conversion uses.

(ii) Client makes it right (CMIR). 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.

(iii) Server makes it right (SMIR). Like CMIR, only the server is responsible for all conversions.

(iv) Universal. Both client and server convert to a common or universal format. OSF will use ISO 10646 (UCS-2 Level 2) as a universal encoding format.

These methods are combined into an order of preferences to make up a conversion model.

(d) **Conversion Model**

Conversion model is an ordering of preferences of methods, such as ‘‘CMIR first, then SMIR, and if these two methods don’t work, use Universal’’. A conversion model consists of one or more conversion methods. When multiple conversion methods are used, the conversion model is considered a dynamic model, since the actual conversion method is selected at application execution time.

(e) **DCE Portable Character Set (PCS)**

A minimum group of characters guaranteed to be supported on all compliant systems. The DCE PCS includes all the graphic characters in ASCII - that is, letters A-Z and a-z, digits 0-9, and common punctuation. The PCS may be encoded in multiple ways. The DCE PCS is defined as idl_char data type.

The list of DCE PCS is:

0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] ^ _ ‘ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ! " # $ % & ’ ( ) * + - . / <space>

(f) **Encoding Method**

In order to use computers in some countries, it may be necessary to combine multiple code sets in a data stream. An encoding method provides the rules for combining sets and recognizing the set to which a given character belongs. An implementation of an encoding method is a specific instantiation of an encoding method’s rules. Examples of encoding method implementations are eucJP (Japanese EUC Code) or a specific version of Japanese SJIS.

Although code sets and encoding methods are different, they are treated nearly the same with respect to interoperability issues. Unless otherwise stated, the term code sets refers to both concepts. The distinction between character and code sets, however, is important. [RFC 27.0]

(g) **Heterogeneous Environment**

In the Heterogeneous Environment, all hosts support the DCE Portable Character Set. But, there is no single homogeneous character set which is shared across all hosts in a network. Each host may support its own character set(s) so long as it is a superset of the DCE Portable Character Set, and, may use its own code set which may be different from the code set that other hosts are using. Like the Homogeneous character set Environment, if the code set used by a server is different than the client code set, then the code set is converted to make the communication possible. The code set conversion can take place either on the server side, on the client side, or both. Since different code sets support different character sets, a potential exists for massive data loss when code set compatibility is not evaluated correctly.

(h) **Homogeneous character set Environment**

In the Homogeneous character set Environment, all DCE hosts within a network use the same Character Set. Each host may use a different code set to encode the character set locally. If the code set, used by a given server, is different than the code set of a client that wishes to bind with the server, then the code set is converted to make communication possible. This conversion can take place either on the server side, on the client side, or both. Even though only one character set is supported in the network, different code sets include different range of characters, so a minor data loss can be resulted by the code set conversion. For example, Japanese SJIS does not include JIS X0212 characters, while eucJP can encode them.

(i) **Homogeneous code set Environment**

This environment consists of a single code set which is supported by all hosts in a network. It assumes that a single code set is used to transfer text strings between hosts. No code set conversion is necessary, and there will be no data loss in a communication. This environment does not specify the code set to be used. The only restriction is that the code set used must be a superset of (contain at least the characters in) the DCE Portable Character Set.

(j) **I14Y**

Interoperability

(k) **I18N**

Internationalization

(l) **Locale**

A locale is the definition of the subset of a user's environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system. Categories are defined for character classification and case conversion, collation order, date and time formats, numeric, non-monetary formatting, monetary formatting, and formats of informative and diagnostic messages and interactive response. [XOPEN]

3. TARGET

The I18N enhancements for RPC application development is aimed to be used by application developers who wish to exchange data outside of the DCE Portable Character Set.

4. GOALS AND NON-GOALS

The goals of the I18N enhancements for RPC application development are:

(a) Provide a way to exchange a variety of character and code set values. Automatic code set conversion can be done by using a new ACF feature. But automatic conversion outside of PCS is optional and under the application programmer's control.

(b) Provide a mechanism for a client to evaluate a server before it decides to establish a connection with it. Also provide a mechanism to give a client enough information for this evaluation to be done in appropriate fashion.

(c) Provide a mechanism for code set transparency during communication between client and server. Application programs do not need to be aware of which code set they are operating on.

(d) Provide necessary tools for global heterogeneous I14Y (e.g., character and code set registry, and routines to access it).

(e) Provide new management routines to handle attributes in CDS namespace. This new functionality is used for registering, retrieving and removing server's supported code set information.

(f) Tags are used to make global heterogeneous I14Y possible. These tags are exposed in the interface definition to be consistent with our philosophy of showing all data that is passed between a client and a server.

(g) Allow application interfaces to be written now that could be used with the new ACF feature when it becomes available.

In other words, this functional specification describes the procedure to exchange strings outside of the DCE PCS. Even though DCE 1.1 is not available yet, as long as application programs follow this procedure, the future interoperability between DCE 1.1 application programs is guaranteed. To be more concrete, if current application programs set tags appropriately, they can interoperate with DCE 1.1 programs. The current application programs cannot take advantage of automatic code set conversion and other new features like evaluation of character and code set interoperability, however, these features can be added to the application programs when DCE 1.1 is available.

(h) No protocol will be changed. The `idl_char` definition will not be changed in anyway.

(i) Keep backward compatibility between the previous DCE versions and DCE 1.1.

The non-goal is:

(a) Code set tags are part of user data, and they cannot be hidden in internal RPC mechanism. There will be no attempt to hide tags from interface definition.

5. REQUIREMENTS

This enhancement should enable exchanging characters outside of the DCE PCS in DCE applications. It includes the code set conversion mechanism, which converts from one code set to the other, for cases when server and client both understand the same character set (e.g., the same language), but use different code sets (e.g., the way each character is represented).

In the case where a server only understands character sets which are incompatible with a client's character sets (e.g., Korean and French), and client and server are processing the characters, establishing a connection can be meaningless. In that case, a client can decide what to do.

There are various conversion method to establish a way that how code set conversion is performed. The current automatic character conversion between ASCII and U.S. EBCDIC within the DCE PCS domain only supports the RMIR (Receiver Make It Right) method.

For the new feature to be flexible enough for world wide requirements, RMIR method might fall short of the needs of variety of applications. Thus, the architecture of this new feature should be designed so an application can decide what is the right method by constructing a conversion model. Conversion model is an ordering of preferences of methods. Each application developer can put a bias toward which conversion method should have a priority over the others.

Automatic character data conversion outside of the DCE PCS is optional. As long as tag(s) are correctly set for data which flow over the wire, world wide inter-operability is guaranteed.

OSF provides a mechanism to give enough information to application developers to decide the inter-operability between client and server. New NSI management routines can be used to export code set information to CDS namespace.

OSF will provide several sample character and code set compatibility evaluation routines and code set conversion routines. When these sample routines fall short of an application program's needs, application developers can write the customized routines. OSF develops the framework which takes these customized routines.

OSF provides an API to advertise the server's supported code sets information to CDS entry.

6. RPC RUNTIME I18N FEATURES IN DCE 1.1

Since there are many new concepts involved in the RPC runtime support for I18N characters, this section is devoted to explaining the various models and features for DCE I18N applications.

There are two major sub-sections. The first sub-section describes an I18N application as a whole. Both execution model and development model are explained.

The second sub-section describes functional models and implementation models. In this sub-section, a conceptual model of each feature is discussed without referring to any implementation details first. This functional models discussed can be implemented in other technologies outside of DCE.

Then implementation model is described for the functional model. Here we investigate how the conceptual model can be implemented in the DCE architecture.

6.1. I18N Applications in the Distributed Environment

6.1.1. Overview of the application execution model

In the new application execution model,

(a) A server advertises its supported character and code sets information into the namespace.

(b) A client imports this information and evaluates the server. If the client and server are using the same code set, there will be no data loss. If they are using different code sets, the client can check to determine whether the server's code set is compatible with the client's. If it is, data loss (if any) will be minimal. Code sets are considered compatible if they encode the same character set, or language. For example, a client and server that both support Japanese, but use different code sets, are considered compatible. However, a client that supports French and a server that supports Chinese are considered incompatible because conversion between the two would almost certainly result in massive data loss.

(c) A compatible server is returned to the client application in the process of binding.

(d) A client stub is called with data represented in the local form.

(e) These data are converted to an appropriate code set representation if necessary, and sent to the server with code set identifying information.

(f) A server stub receives the data and converts it into its local representation based on the code set identifying information, if necessary.

(g) The manager routine returns the data to the server stub. This data may be converted from local representation to network representation, based on a format from the client about its code set support. Server stub sends data together with code set identifier for network representation of the data.

(h) A client stub receives the data, converts it to a local representation, if necessary, based on the code set identifying information from the server.

(i) Client receives the data from a client stub. At this point, data is represented in client's local code set.

There is no conversion involved when both server and client use the same code set.

6.1.2. Overview of the application development model

(a) The application programmer writes an IDL file specifying code set identifiers and character data as parameters to operations.

(Implementation note: The interface definition identifies character data that may be automatically converted by creating a particular typedef that may be flagged in the associated attribute configuration file.)

(b) The application programmer writes a matching ACF file which requests that automatic conversion takes place on character data in parameters having a certain data type. This ACF file attribute allows the programmer to name the local data type. Character data will be stored in this data type. ACF also name a routine that will be called from within the stubs that will set the code set identifiers properly.

All names will have appropriate defaults and all default routines will be provided in the DCE stub support library. If the programmer selects routine names other than the default, the programmer must supply these routines.

6.1.3. Server development

(a) The server application programmer calls routines that determine the local code set and code set converter support available in this process (DCE library routines are provided which request this information from the local operating system).

(b) The server application programmer writes application code which calls NSI routines which place local code set and converter info into the server entry[ies] for this process. This information is assumed to coexist with the binding info for this server. (This step can be considered to be an extension of the `rpc_ns_binding_export()` step.)

(c) The server application program listens for requests.

6.1.4. Client development

(a) Client developer writes code that calls `rpc_ns_import_begin()` in the normal manner to begin the process of selecting a compatible server.

(b) The developer invokes a new server evaluation mechanism which places additional checking on the acceptable bindings. In the original import algorithm, only protocol and interface information was used to determine client / server compatibility. This new capability issues a call to user-supplied routine(s) to judge whether the server is acceptable. The routine is passed the name of the current namespace entry, and all information from `rpc_ns_binding_import_begin()` to determine compatibility. In the context of code set compatibility, the evaluation routine will read code set attributes from the namespace and compare it to local code set support to determine whether there is a possibility of significant data loss when conversion is done, or if conversion is needed at all.

OSF will supply several sample evaluation routines and examples for demonstrating how to use these features. The sample evaluation routines will implement two conversion models (between client and server). One evaluation routine will use universal code set when there is no converter available for a particular pair of code sets, even though those code sets are compatible. The other routine will not fall back on the universal code set, because some applications do not want to use the universal code set when there is no converter available. Examples include how these models will be used based on the sample code set compatibility evaluation logic. See Functional Model and Implementation Model sections for the sample logic.

(We cannot guard against minor data loss, (except if an application requires the client and server to use the same code set) but we can prevent massive loss with our evaluation functionality.)

(c) The developer supplies evaluation routines if samples are not sufficient.

(d) The developer calls the import next routine which evaluates the various server entries (through issuing a call to evaluation routine(s)) and returns a compatible server binding.

(e) Client makes a call to the client stub passing data in the representation identified in the API in the ACF.

If the sample routines provided with the DCE are not sufficient to handle the application's needs,

The client can supply any of the followings:

(a) The routine to set tags.

(b) The routine to calculate buffer size.

(c) The routine to convert code sets.

(d) The routine to evaluate the code sets compatibility.

Appendix ''Customized Routines'' explains more about these routines.

6.2. Functional Model and Implementation Model of the New Features

This section describes, 1) the model of the functionalities for the new RPC runtime support, and 2) how each feature is actually implemented. The idea behind of each feature is explained in detail without referring to the implementation details. Then how each model can be implemented in DCE 1.1 is discussed next.

6.2.1. NSI management

6.2.1.1. Functional model

For the server to advertise supported character and code sets, and for a client to obtain this information, a new NSI management mechanism is required. This is because the current NSI management functionality in RPC runtime only accepts pre-architected attributes, like protocol tower, uuid, ns group, or ns profile.

The new NSI management mechanism takes character and code sets information and places them into the namespace. It also performs a retrieval and removal of the information from the namespace.

6.2.1.2. Implementation model

The current RPC runtime library only accepts pre-architected attributes, like NS profile, object uuid, NS tower, or NS group. In other words, only these attributes are assigned **OID**s (object identifiers) in a OID domain, which is allocated to OSF. (OSF currently does not use OIDs above `1.3.22.1.1.4`).

The new attribute `code sets` will be assigned OID `1.3.22.1.1.5`, and supporting routines will be added to the RPC runtime library. Several routines to handle NSI attributes in the RPC runtime library exist already. The new routines will be similar to the existing routines to add, read, and delete the attribute in/from the CDS namespace.

The attribute `code sets` is expressed as an array of integers, so a new feature for IDL compiler (IDL Encoding Services), will be used to encode the code sets values into an endian-safe format before it is placed in CDS namespace. The IDL Encoding Services is provided with DCE 1.1 IDL compiler.

6.2.2. Evaluation

6.2.2.1. Functional model

The process of selecting a compatible server from a client is called **evaluation**. In the evaluation process, client checks the server's supported character and code sets that are advertised, and determines if a binding to the server is reasonable, based on its own supported character and code sets.

In the process of selecting a binding, we need to make sure that client and server are inter-operable for character and code set compatibility. In other words, we need to make sure that data will be communicated between the client and the server without resulting in a lot of data loss. For example, sending Chinese data to a Greek server would not be reasonable, since much of the data can be lost. In addition, even if both client and server understand the same character set, e.g., Japanese, if a client uses SJIS code set and a server uses eucJP(EUC variant) code set for character processing, we want to know which code set converters are available in client and server, so we can perform the necessary code set conversion.

The problem that needs to be solved is really a two step process.

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

(b) The code set conversion problem (i.e., given that the characters 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).

A result of the evaluation is whether the client and server can communicate characters adequately. In addition, the client will determine how data will be communicated, i.e., where (client side, or server side, or both sides) the conversion will take place.

For a client to get the information about server's supported character and code sets, a new NSI management mechanism can be used. This new mechanism can be used by a server to advertise its supporting character and code sets, too.

Also to make its supported character and code sets available for both client and server, an interface to the local operating system is provided. This mechanism queries a local operating system about its supported character and code sets, since different OSes support different groups of code sets. Information about supported character and code sets are mapped to globally unique identifiers via the OSF character and code set registry interface.

The flow of the evaluation process is:

(a) Server gets its supported character and code sets information from the local OSF character and code set registry

(b) Server advertises that information

(c) Client gets this advertised information

(d) Client gets its supporting character and code sets information from the local OSF character and code set registry

(e) Client determines and selects the appropriate server. In a sample implementation, first acceptable server will be selected.

6.2.2.2. Implementation model

The evaluation process can be divided into two steps. One is to determine if a server is inter-operable, and the other is to select the way clients and servers inter-operate.

For step one, to determine if a server is inter-operable, the client will set a pointer to an evaluation routine into a context handle structure as a part of the binding selection process. Application developers can provide this evaluation routine, or they can use OSF-supplied routines.

When an evaluation routine is set, a function pointer to an evaluation routine will be a part of context handle to the NS binding selection routine (namely, `rpc_ns_binding_import_next()`). This mechanism can be used not only for character and code set compatibility evaluation, but also for other future functionality, therefore it is designed as list structure (that includes a list of elements) which hold function pointer(s).

When the NS binding selection routine is called, it examines a context handle to see if this list information is included. When the list contains a function pointer to an evaluation routine, it is dereferenced and the function is executed.

For step two, to select the way the client and server inter-operate, an evaluation routine will choose the model of client and server communication. Based on the model selected, a tag(s) set routine can decide the actual tag value for a data. This tag(s) set routine is called from rpc stubs before the marshalling of data.

6.2.2.2.1. Logic of evaluation

A client and a server compatibility is determined by three steps:

(a) The first is the code set compatibility evaluation.

(b) When code set is not compatible, character set(s) is evaluated for compatibility.

(c) The last is the selection of the conversion method between a client and a server, based on the code sets used.

The OSF supplied sample evaluation routines will check if the same code set is used by a client and a server. When it is, the evaluation is done. They are compatible.

Next, character sets used by a client and a server is checked. When both client and server are using the same character set, it will be safe to exchange coded character values regardless of the encodings which is used to represent the character set. When character sets are determined compatible but client and server are using different representations of the character set (code set), the conversion model is used to determine a conversion method.

The somewhat complicated issue here is that a code set can contain several character sets. For example, some Asian code sets contain the Latin-1 character set in their character domain. Does this mean an ISO 8859-1 client is compatible with an eucJP server if both of them have ISO 8859-1 to/from eucJP converters? They could be compatible as long as only the client sends data, or if the server either returns no data or returns only Latin-1 data to the client. However, in most cases, they are not compatible because the repertoire of characters in eucJP is so much larger than ISO 8859-1, and there is no simple way to ensure that clients and servers limit their data exchange to characters that are in both of their repertoires. Therefore, for a sample implementation, the OSF evaluation routines will consider ISO 8859-1 and eucJP to be incompatible.

When a universal code set is used in a server, there is no good way to determine character set compatibility. Since universal code set encodes all characters, simply examining server's supported code sets will not tell anything useful. In this case, a client really needs to know how that server processes character data. Since the evaluation will rely on a knowledge of a client, customized evaluation routine will be required.

Even though a server is not using a universal code set, application developers can still provide their own the evaluation logic to modify the sample behavior.

When client's and server's code sets only consist of one character set, the evaluation logic will be simple. If their character sets match, then select the way they can be connected.

But when their code sets consist of more than one character sets, then the character set evaluation will be done based on the logic that:

‘‘If more than two (≥ 2) character sets match between client and server, then their character sets are compatible, otherwise they are not.’’

In an environment where all machines have all available converters installed, just checking the converters for a server will not be enough to determine the client and server code sets compatibility. For example, when a server machine has Korean converters, the server process is not necessary configured to process Korean characters. It might recognize ASCII only. Character sets compatibility checking will be helpful to deal with that kind of configuration (except the case when a universal code set is used by a server).

The code set conversion can be done with several models, namely RMIR, SMIR, CMIR, INTERMEDIATE, or UNIVERSAL. In case of RMIR, SMIR, or CMIR, the conversion will take place once in a remote procedure call, either the receiving side, the server side, or the client side. In case of INTERMEDIATE or UNIVERSAL, the code set conversion takes place in both sides. The UNIVERSAL method always uses the universal code set (ISO 10646: UCS-2 Level 2) as the network code set. The INTERMEDIATE method uses the user defined code set(s) as the alternatives of the UNIVERSAL code set.

The INTERMEDIATE code set(s) can be set within the OSF character and code set registry by using `csrc` (code set registry compiler) option. `csrc` will read the source file of the OSF character and code set registry (`code_set_registry.txt`), and generates the registry (`code_set_registry.db`). When code set(s) is specified with the intermediate option (`-M`) to the `csrc` tool, the code set(s) is placed at the beginning of the code sets array, so it will have a priority over the other code sets when the compatibility evaluation takes place. The evaluation logic will search the supported code sets array from the beginning, so it is likely that the intermediate code set is used for the data exchange if it is supported by both client and server.

6.2.3. Interface to the local operating system

6.2.3.1. Functional model

For both server and client, the information about supported character and code set is required to determine interoperability. Since this information belongs to the local operating system domain, a new mechanism is needed to provide the interface to the local operating system.

In addition, since DCE 1.1 will rely on a local code set conversion mechanism to perform code set conversion, we need the following two interfaces to the local operating system resources.

(a) A routine that calls local code set converters

(b) A routine that queries supported character and code sets

The first routine is used for code set conversion. The second routine is the new functionality.

6.2.3.2. Implementation model

New routines are provided to perform the following two functions.

(a) Routine which calls the local code set converters.

    This is a routine to call the appropriate iconv converters for code set conversions. This is a wrapper routine, and will be newly written.

(b) Routine which returns the supported character and code sets.

    This is a routine to access the local OSF character and code set registry, and to produce the information about the supported character and code sets. For example, this routine will access the `/usr/lib/nls/csr` directory as the default search path, and will read `code_set_registry.db` file to get the information. This routine will be newly written.

6.2.4. OSF character and code set registry

6.2.4.1. Functional model

OSF character and code set registry provides a mechanism to uniquely identify code sets. OSF is proposing OSF Character and Code Set Registry [RFC 40.1] to provide application programs the ability to uniquely identify every code set, including vendor specific ones. DCE 1.1 will include the routines to access the OSF character and code set registry, so application programs can map platform dependent code sets to the unique identifiers.

The OSF character and code set registry contains information about character set and code set with organization identification. Registered values are required to make inter-operability possible, since different organizations (different vendors) can have different encodings (vendor-specific encodings) as well as different names for the same character and code set (ISO 8859-1 can be named ISO8859-1, Latin-1, 88591, iso8859-1, etc.)

6.2.4.2. Implementation model

The OSF Character and Code Set Registry is a table that maintains

(a) **character set**

and

(b) **code set**.

Each registry entry has this information along with an integer value that represent the entry. This integer value is assigned by OSF. This is a 32-bit integer; the upper 16 bits identify the organization and the lower 16 bits identify the code set. In addition, each entry has one or more character set ID, which is 16-bit integer. [RFC 40.1]

Two new functions are implemented: One takes a string code set name, and returns the integer code set value. The other takes an integer code set value and returns the string code set name.

Both functions can return the number and supported character sets array as options.

6.2.5. Code set conversion

6.2.5.1. Functional model

When a client sends a data to a server, this data should be understood by both the client and the server. This understanding should be achieved in two levels.

(a) Both client and server understand a character set

(b) Both client and server understand the way the character set is expressed

Code set conversion provides a way for a character set to be represented in a form both client and server can understand.

The examples of a character set are DCE PCS, German, Chinese, or Japanese. Code set conversion cannot take care of character set conversion, since mapping German characters to Chinese characters will probably not make sense in most cases.

However, if both server and client use the same character set, say Japanese, it will make sense to exchange data even though their code set is different, like SJIS and eucJP(EUC variant).

The IDL enhancement can generate a hook to code set conversion mechanism that is invoked from stubs, so from either or both client and server stubs, the code set conversion mechanism can be invoked to perform an automatic conversion of different code sets, which translates a language from one representation to the other.

6.2.5.2. Implementation model

DCE 1.1 will provide a wrapper routine, which is explained in ‘‘Interface to the local operation system’’ section, to invoke the XPG4 code set conversion facility.

In a reference implementation, code set conversion is done by the XPG4 code set conversion facility, namely iconv. iconv is provided by a local operating system. This is a set of library routines to take one code set and map the value into another code set. (XPG4 also defines a command called `iconv`, to perform code set conversions. But what we will use for DCE 1.1 are the library functions.)

iconv facility consists of the following routines:

(a) `iconv_open ()`

Code conversion allocation function.

(b) `iconv()`

Code conversion function.

(c) `iconv_close ()`

Code conversion deallocation function.

`iconv_open()` returns a conversion descriptor that describes a conversion from the code set specified by the string pointed to by one argument to the code set specified by the string pointed to by the other argument.

`iconv()` converts the sequence of characters from one code set into a sequence of corresponding characters in another code set.

`iconv_close()` deallocates the conversion descriptor and all associated resources allocated by the iconv_open() function.

Since `iconv_open()` only takes the *code set specified by the string* and there are no standard which defines code set names so far, we need to have another mechanism to uniquely identify code sets to be used. Also in some cases for DCE internal operation, using an integer value to distinguish each code set is more appropriate than using string value.

For this purpose, the OSF Code Set Registry values are used to access the uniquely identified code sets in a sample implementation.

6.2.6. Buffer sizing

6.2.6.1. Functional model

When converting data from one code set to another, there is always a possibility that the converted string can be larger than the original string. This is because one character can be encoded in a variable number of bytes in different code sets.

To support an automatic conversion operation, a buffer sizing mechanism needs to be supplied to calculate the actual size of a buffer, so stubs can allocate a buffer of sufficient size to hold the converted string.

This mechanism is not meant to be used by application developers, but it is designed to be used by the RPC stub code for marshalling and unmarshalling of characters outside of DCE PCS range.

6.2.6.2. Implementation model

Since there can be a need to calculate a buffer size for code set conversion, a routine to determine the storage requirements for converting the data to transmissible form will be provided.

This new routine(s) will be a part of RPC runtime library, and be called from RPC stubs before marshalling and unmarshalling of data.

The logic for determining the new buffer size will be:

```
number-of-bytes * MB_CUR_MAX
```

`number-of-bytes` is the value that stubs receive from the network. (marshalling always requires the number of bytes be passed over the network.)

`MB_CUR_MAX` is a platform dependent value that is defined as the maximum number of bytes for a character, for the current locale.

6.2.7. Tags usage

6.2.7.1. Functional model

A code set identifier is called *tag*. Tags are necessary to correctly identify data that flows on the wire in case the data is character data outside of the DCE PCS range.

For the data sent from a client to a server, the client must send character data in an encoding that is understandable by the server. The choices of encoding are found in exported code sets in either NSI or in a local repository. Universal encoding can be used, when it is included in choices. One tag is used to indicate the encoding.

For the data sent from a server to a client, the client must indicate to the server a code set which server will use on reply. One tag is used to indicate the code set.

When a server send a reply to a client, it must indicate which encoding is used to represent the data. The encoding can be the code set a client indicates to the server, or the local encoding the server uses. In the latter case, the client should be able to support RMIR method of code set conversion. Universal encoding can be used, when it is included as one of the encoding choices.

6.2.7.2. Implementation model

A maximum of three tags will be used when sending any character data outside of the PCS over the wire. They are:

The sending tag, indicates the code set the client sends over the wire.

The desired reception tag, indicates the code set the client prefers to receive.

The reception tag, indicates the code set which server is actually sending.

When a client sends the data over the wire, the reception tag is not used. The reception tag is set by a server when a data is marshalled in the server stub.

When a client only needs to send a data and no output is expected, only the sending tag is used. When a client needs to receive data from server, both the desired reception tag and the reception tag are used. When client wants to send and receive data, all three tags are used.

These tags can be set directly by an application, or a new ACF feature can be used to set the tags automatically. These tags need to be listed in the IDL file explicitly. This is because:

(a) When data flows on the wire, it needs to be explicitly defined in an interface definition. This is the philosophy OSF is following.

(b) Tags are part of application data. They do not relate to DCE/RPC internal operation in any way.

The IDL description of an operation with a parameter which is a conformant varying of characters from a non-PCS set could be written as:

```
typedef byte my_byte;

void a_op(
        [in] unsigned long stag,
        [in] unsigned long drtag,
        [out] unsigned long *p_rtag,
        [in] long s,
        [in, out] long *p_l,
        [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);
```

7. FUNCTIONAL DEFINITION

7.1. NSI Management of New Attribute in CDS Namespace

New attribute `code sets` is to be assigned a new object OID within OSF OID range.

Code sets is defined as a structure to hold a list of integer values. Each integer value is one of the values from the OSF Code Set Registry, and at top of the list is the current code set which is used by a process (either client or server).

This code sets is converted into an endian-safe format before it is exported to RPC entry within the CDS namespace. The IDL encoding services for the encoding and the decoding operations. Appropriate interface definition and ACF definition will be written to define these methods.

A routine is written to take this code sets and to export it into the CDS namespace. It takes a structure code sets and converts it into an endian-safe format before the attribute is exported into CDS namespace. This routine is a part of RPC runtime library.

A set of routines will be written to read this exported attribute from the CDS namespace. The first routine sets up an inquiry context, similar to the `rpc_ns_entry_object_inq_begin()` routine. The second routine reads an attribute based on the inquiry context, similar to the `rpc_ns_entry_object_inq_next()` routine. This routine reads a string from CDS, and converts it into a `code set` structure. The last routine finishes the operation, similar to the `rpc_ns_entry_object_inq_done()` routine. These routines are a part of the RPC runtime library.

For the convenience of reading exported code set value, OSF provides a routine to be tailored for the code set attribute retrieval. This routine will call the set of routines internally.

A routine will be written to remove a code sets attribute from the CDS namespace. This routine operates in a similar way to the `rpc_ns_mgmt_binding_unexport()` routine. This routine is a part of the RPC runtime library.

New RPC NSI attributes, rpc_c_ns_codesets is used to indicate the new attribute.

7.2. Interface to the Local Operating System

A routine will be written to access the local OSF character and code set registry to get the information about supported code sets. This routine will check which code set is supported, and fills the values into a code sets structure. The first entry of a list of supported code sets should be the current code set under which a process is operating on.

7.3. Compatibility Evaluation Between Client and Server

A list structure which holds function pointer(s) to a routine is defined. An evaluation routine will be pointed by the function pointer as one case, but other kind of routines can also pointed by the function pointer, too.

The contents of a list will be examined and the function pointer is dereferenced at the binding selection process in a client. The routine can be written so when it is called from rpc_ns_binding_import_done() routine, it can perform a clean-up operation. In case of code set evaluation routine, this feature will not be used.

This list structure has fields indicating:

(a) Function type.

(b) Function pointer.

(c) Preserved import context.

(d) Pointer to a next element in the list.

A routine to add a function pointer into this list structure will be provided. This routine takes:

(a) Import context, which is allocated by `rpc_ns_binding_import_begin()` routine.

(b) Function pointer to an evaluation routine.

(c) Function type.

(d) Status.

When this routine is called, the list structure is allocated and appended to the import context. One of the fields defined in the current import context, which is not used until the routine `rpc_ns_binding_import_next()` performs the actual binding, will be used as a flag to indicate whether this list structure is appended to the import context or not.

This flag field is `inq_cntx`, and it is a part of the `rpc_lkup_rep_t` structure (which is defined in nsp.h in the RPC runtime library).

The functional pointer will be dereferenced in `rpc_ns_binding_import_next()` routine. This existing routine will be modified to check if the list structure is appended to a import context (by looking at the flag field), and execute the evaluation routine. The signature of `rpc_ns_binding_import_next()` routine will not be changed. This is possible since the argument, `import_context`, is `rpc_ns_handle_t` data type which is an opaque pointer. Within the routine, this pointer is cast to the actual data type, so depending on the value of the flag field, this opaque pointer will be cast to the right structure, and the dereference of an evaluation routine becomes possible.

The evaluation routine can evaluate anything, and the use of it is not restricted to code set evaluation. However, the main use of this evaluation routine for DCE 1.1 is code set evaluation. A sample code set evaluation routine returns the model of code set conversion, which is one of the No Conversion, RMIR, SMIR, CMIR, INTERMEDIATE, or UNIVERSAL. When this evaluation routine cannot find an inter-operable model between client and server, it returns IMPOSSIBLE (`rpc_s_ss_no_compat_codeset` for OSF implementation). If the model value is anything except IMPOSSIBLE, this model value is appended to RPC binding handle. This RPC binding handle will be passed to stubs, through client RPC call.

7.4. Logic of Compatibility Evaluation

The sample OSF evaluation routines will use the following algorithm to determine the compatibility between a client and a server.

Even though character set compatibility checking is helpful for the environment where all nodes have all available code set converters, avoiding character set compatibility might be necessary depending on the needs of an application. Walking around the checking of character sets compatibility is indicated by ‘‘no character set comparison is defined’’.

```
IF (client and server are using the same code set) {
    Great.  They are compatible;
    return (NO_CONVERSION_NECESSARY);
}

IF ("no character set comparison" is defined) {
        Apply "conversion model" evaluation;
        return (the result of "conversion model" evaluation);
}

IF (client's code set has only 1 character set &&
            server's code set has only 1 character set) {

    IF (their character sets match) {
            Apply "conversion model" evaluation;
            return (the result of "conversion model"
                    evaluation);
    }
    ELSE
            return (IMPOSSIBLE);

    IF (more than 2 character sets match between
        client & server) {
            Apply "conversion model" evaluation;
            return (the result of "conversion model"
                    evaluation);
    }
    ELSE
            return (IMPOSSIBLE);
```

```
}
```

7.5. Code Set Conversion

The actual code set conversion is done by the XPG4 iconv routines, so this code set conversion routine is a jacket routine to call iconv routine internally based on the tags value.

The tag value is expressed as an integer value, and since the XPG4 iconv routines are defined to take a string code set name, this routine calls the OSF Character and Code Set Registry routine to convert an integer code set to a string code set name.

There are four OSF supplied code set conversion (jacket) routines. For each data type, like `byte` or `wchar_t`, two routines are required to perform code set conversion. One routine converts the local data type for network representation, and is used for marshalling. The other routine converts the network representation of the data to the local data type, and is used for unmarshalling.

The signature for these routines are (*local_data_type* and *network data type* are place holders):

```
void local_data_type_to_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
        /* in */   local data type             *ldata,
        /* in */   unsigned32                   l_data_len,
        /* out */  network data type           *wdata,
        /* out */  unsigned32                   *p_w_data_len,
        /* out */  error_status_t               *p_st
);

void local_data_type_from_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
        /* in */   network data type           *wdata,
        /* in */   unsigned32                   w_data_len,
        /* in */   unsigned32                   l_storage_len,
        /* out */  local data type             *ldata,
        /* out */  unsigned32                   *p_l_data_len,
        /* out */  error_status_t               *p_st
);
```

Network data type is always resolved into `idl_byte`. In the OSF sample implementation, local_data_type will be either `byte` or `wchar_t` data type.

These signature were defined by DEC. [DEC 1]

(a) `byte_to_netcs()`

This routine takes a (multi-)`byte` value, and converts it to the code set which is defined by a tag, when necessary. The result string is in byte form, and is sent over the wire. This routine is used for marshalling.

(b) `byte_from_netcs()`

This routine receives a `byte` stream from the wire, and converts it to the code set which is used by a server, when necessary. The result string is in `byte` form, and is passed to a manager routine. This routine is used for unmarshalling.

(c) `wchar_t_to_netcs()`

This routine takes a process code (`wchar_t`) value, and first converts it to (multi-)`byte` string, then converts the string to the code set which is defined by a tag, when necessary. The result string is in `byte` form, and is sent over the wire. This routine is used for marshalling.

(d) `wchar_t_from_netcs()`

This routine receives a `byte` stream from the wire, and converts it to the code set which is used by a server, when necessary. Then this string is converted to process code (`wchar_t`) values of server process. This process code is passed to a manager routine. This routine is used for unmarshalling.

These routines convert an entire array buffer pointed by `ldata` or `wdata` all at once (as `iconv` does).

7.6. Character and Code Set Registry Access

The OSF Character and Code Set Registry includes a table in which each entry has a 32-bit integer value, a code set name and other supplementary information.

This table is a `code_set_registry.txt` file. Each Operating System provides its specific version of a `code_set_registry.txt` file, since supported code sets will vary from platform to platform.

In the case of OSF/1, this file is located in

```
/usr/lib/nls/csr/code_set_registry.txt
```

The OSF/1 version of a `code_set_registry.txt` would look like the following. The items listed (from the left) are: code set name, 32-bit integer value, and character set ID(s) (# identifies a comment line).

```
# ISO 8859-1:1987; Latin Alphabet No. 1
start
description ISO 8859-1:1987; Latin Alphabet No. 1
loc_name    ISO8859-1
rgy_value   0x001001
char_values 0x0011
max_bytes   1
end
  ........
```

This input source file is converted to the binary form by `csrc` tool. The binary form is `code_set_registry.db` and is found in the same directory where `code_set_registry.txt` is stored.

Two routines are provided to access the binary file. One takes a string code set name and returns an integer value which represents its corresponding entry in the OSF Character and Code Set Registry. The other takes an integer value and returns a string code set name.

These code set registry routines basically perform a table search, the key being either an integer value or a code set name.

7.7. Setting Tags

The routine `rpc_cs_get_tags()` is called from rpc stubs before the code set conversion routine is called. This routine will take a binding handle, which contains the model information of how the client and server communicate, and sets the tags. When the binding model information is not set, the `rpc_cs_get_tags()` routine will determine a binding model, and will select the tags. This is possible when previously called evaluation routine gets server's and client's supported code sets, and sets that information as a part of binding handle. `rpc_binding_eval_t` data structure contains server's and client's supported code sets information. See appendix ‘‘Customized Routines’’ for more information.

When this routine is called from a client stub, the `sending` tag and the `desired reception` tag are set, and the `reception` tag is ignored. When this routine is called from a server stub, the reception tag is set, and the sending tag is ignored. Desired reception tag is used as input to determine the most appropriate reception tag value.

To indicate from which stub it is called, the boolean value `server_side` is used in the signature of this routine. This signature was defined by DEC. [DEC 1]

```
void
rpc_cs_get_tags (h, server_side, p_stag, p_drtag, p_rtag, p_st)
        /* in */      rpc_binding_handle_t       h;
```

```
/* in */     boolean                    server_side;
/* out */    unsigned32                 *p_stag;
/* in/out */ unsigned32                 *p_drtag;
/* out */    unsigned32                 *p_rtag;
/* out */    error_status_t             *p_st;
```

7.8. Buffer Sizing for Marshalling and Unmarshalling

A buffer sizing routines are used within stubs before marshalling and unmarshalling of data, since when code set conversion is performed, there is always a chance that the conversion buffer needs to be enlarged to hold converted data.

The role of this buffer sizing routine is to determine the `conversion type`, and calculate the size of necessary buffer (when a new buffer is required) so the rpc stub can allocate the necessary buffer before it calls a conversion routine.

Conversion type will be either:

(a) **no conversion required**

(b) **in place conversion**

This means the code set conversion can be done in a single storage area. No buffer allocation is required.

(c) **new buffer conversion**

This means the converted data must be written to a new storage area. The new buffer needs to be allocated.

These types are defined in `idl_cs_convert_t` enumeration type.

When `new buffer conversion` is selected as the conversion model, the following rule is used to calculate the new buffer size.

```
number-of-bytes * MB_CUR_MAX
```

`number-of-bytes` is passed to the stub from the network. `MB_CUR_MAX` is defined as a system dependent global variable, and can be accessed by including `<stdlib.h>` header file.

There are four OSF supplied memory management routines. For each data type, like `byte` or `wchar_t`, two routines are required to select the conversion type and calculate the buffer size. One routine is used for marshalling, and the other routine is used for unmarshalling.

Only when `new buffer conversion` type is selected, the buffer size will be different from the `storage length` of a local data.

The signature of these routines are:

```
void local_data_type_net_size (
        /* in */   rpc_binding_handle_t       h,
        /* in */   unsigned32                 tag,
        /* in */   unsigned32                 l_storage_len,
        /* out */  idl_cs_convert_t           *p_convert_type,
        /* out */  unsigned32                 *p_w_storage_len,
        /* out */  error_status_t             *p_st
);

void local_data_type_local_size (
        /* in */   rpc_binding_handle_t       h,
        /* in */   unsigned32                 tag,
        /* in */   unsigned32                 w_storage_len,
        /* out */  idl_cs_convert_t           *p_convert_type,
        /* out */  unsigned32                 *p_l_storage_len,
        /* out */  error_status_t             *p_st
```

```
);
```

These signature were defined by DEC. [DEC 1]

(a) `byte_net_size()`

This routine is called before data marshalling.

It takes a binding handle, a tag that identifies the code set that will be used on the wire, and the local storage size in bytes.

In case of new buffer conversion type is required, it calculates the size of the new buffer in bytes.

(b) `byte_local_size()`

This routine is called before data unmarshalling.

It takes a binding handle, a tag that identifies the code set used on the wire for data being received, and the storage size of the on-the-wire data.

In case of new buffer conversion type is required, it calculates the size of the new buffer in bytes.

(c) `wchar_t_net_size()`

This routine is called before data marshalling.

It takes a binding handle, a tag that identifies the code set that will be used on the wire, and the local storage size in `wchar_t` data type.

In case of new buffer conversion type is required, it calculates the size of the new buffer in `wchar_t` data type.

(d) `wchar_t_local_size()`

This routine is called before data unmarshalling.

It takes a binding handle, a tag that identifies the code set used on the wire for data being received, and the storage size of the on-the-wire data.

In case of new buffer conversion type is required, it calculates the size of the new buffer in wchar_t data type.

8. DATA STRUCTURES

8.1. Data Structures Related to Code Set Conversion

```
/* This data enables a user routine to indicate whether I-char data
 * needs conversion from on-the-wire form to local form, and if so,
 * whether this conversion can be done in place
 *
 * This data type was defined by DEC.  [DEC 1]
 */

typedef enum {
     idl_cs_no_convert,          /* No code set conversion required */
     idl_cs_in_place_convert,    /* Code set conversion can be done
                                    in a single storage area */
     idl_cs_new_buffer_convert   /* The converted data must be
                                    written to a new storage area */
} idl_cs_convert_t;

/* These data contain supported code sets for either server or
 * client.
 *  - The first element in codeset[] is a local code set of the
```

```
 *    process.
 *  - Each code set has an attribute (max bytes) to indicate the
 *    maximum number of bytes needed to encode that code set.  This
 *    is used to calculate the size of a necessary buffer for code
 *    set conversion.
 *  - Other code sets are code sets supported by the process, and
 *    they are converted to the local code set by iconv converters.
 *  - Conformant array is used, since a number of code sets used in a
 *    host will vary.
 */
typedef struct rpc_cs_c_set_s_t {
        unsigned32      c_set;
        unsigned16      c_max_bytes;
} rpc_cs_c_set_t;

typedef struct rpc_codeset_mgmt_s_t {
        unsigned32      version;    /* version of this structure */
        unsigned32      count;      /* number of code sets defined */
        [size_is(count)] rpc_cs_c_set_t codesets[];
} rpc_codeset_mgmt_t, *rpc_codeset_mgmt_p_t;
```

8.2. Data Structures Related to Interoperability Evaluation

```
/*
 * Modifications to the import context.
 */
/*
 * Extension to an import context handle.  The new field
 * 'eval_routines' in 'rpc_lkup_rep_t' will be the following data
 * structure.
 */
typedef struct {
        unsigned32              num;
        rpc_cs_eval_list_p      list;
} rpc_cs_eval_func_t, *rpc_cs_eval_func_p_t;

/*
 * a lookup rep within RPC runtime
 */
typedef struct
{
    rpc_ns_handle_common_t  common; /* Data common to ns handles */
    rpc_if_rep_p_t          if_spec;
    rpc_obj_search_t        obj_uuid_search;
    uuid_t                  obj_uuid;
    uuid_t                  obj_for_binding;
    rpc_ns_handle_t         inq_cntx;
    unsigned32              max_vector_size;
    rpc_list_t              node_list;
    rpc_list_t              non_leaf_list;
    boolean                 first_entry_flag;
    rpc_ns_handle_t         eval_routines; /* rpc_cs_eval_func_p_t */
} rpc_lkup_rep_t, *rpc_lkup_rep_p_t;


/*
```

```
 * Modifications to the binding handle.
 */
/* Appending code set conversion information to a binding handle.
 *
 * -  This is a modification to the current binding handle which will
 *    be returned from rpc_ns_binding_import_next().  This conversion
 *    information is used by rpc_cs_get_tags() routine to set tags.
 *  - When rpc_ns_import_ctx_add_eval() is called by an application,
 *    rpc_ns_binding_import_next() will execute a new logic to
 *    determine code set compatibility.  Then
 *    rpc_ns_binding_import_next() routine will set
 *    extended_bind_flag to RPC_C_BH_EXTENDED_CODESETS, and set the
 *    value to cs_eval.
 *  - RPC runtime does not use RPC_C_BH_IN_STUB_EVALUATION value,
 *    since the evaluation within a stub is not recommended.  This is
 *    only provided for the future use.
 */

/*********************************************************************
 *
 * R P C _ C _ B H _ E X T E N D E D
 *
 * The values of the flag field within rpc_binding_rep_t
 * (extended_bind_flag).  If rpc_binding_rep_t is extended to include
 * a new information in the future, add the definition here, and
 * modify the affected routines accordingly.  These values are only
 * used within the RPC runtime.
 */
#define RPC_C_BH_EXTENDED_NONE          0x0000
#define RPC_C_BH_EXTENDED_CODESETS      0x0001
#define RPC_C_BH_IN_STUB_EVALUATION     0x0002

typedef struct rpc_handle_s_t
{
    /*
     * The following fields are meaningful all the time.
     */
    rpc_list_t                  link;       /* This must be first! */
    rpc_protocol_id_t           protocol_id;
    signed8                     refcnt;
    uuid_t                      obj;
    rpc_addr_p_t                rpc_addr;
    unsigned                    is_server: 1;
    unsigned                    addr_is_dynamic: 1;
    rpc_auth_info_p_t           auth_info;
    unsigned32                  fork_count;
    unsigned32                  extended_bind_flag; /*code set i14y*/
    /*
     * The following fields are not meaningful for binding reps
     * that are passed to server stubs.
     */
    unsigned                    bound_server_instance: 1;
    unsigned                    addr_has_endpoint: 1;
    unsigned32                  timeout;            /* com timeout */
    signed32                    calls_in_progress;
```

```
    pointer_t                   ns_specific;
    rpc_clock_t                 call_timeout_time; /*max exec time*/
    rpc_protocol_version_p_t    protocol_version;
    rpc_cs_evaluation_t         cs_eval;          /* code set i14y */
    /*
     *
     */
} rpc_binding_rep_t, *rpc_binding_rep_p_t;

/*
 * R P C _ C S _ E V A L U A T I O N _ T
 *
 * Data structure which is attached to a binding handle at a client
 * side, when automatic code set conversion is enabled.  The content
 * will be either 'rpc_cs_tags_eval_t' data or 'rpc_cs_method_eval_t'
 * data.
 */
typedef union switch(short key)
{
case 0:     rpc_cs_tags_eval_t      tags_key;
case 1:
default:    rpc_cs_method_eval_t    method_key;
} rpc_cs_evaluation_t;

/*
 * R P C _ C S _ T A G S _ E V A L _ T
 *
 * Data structure which is attached to a binding handle at client
 * side, when automatic code set conversion is enabled.  When 'fixed'
 * flag is not on, code set compatibility evaluation can be done
 * within a client stub.  Performing code set evaluation in a stub is
 * not a good idea for performance wise, however, some application
 * might need that functionality.  Usually, each item is set by an
 * evaluation routine within a client.
 *
 * stag, drtag  : sending tag and desired receiving tag
 * stag_max_bytes : maximum number of bytes required to encode 'stag'
 *                  code set
 * client_tag   : client current code set tag.
 * client_max_bytes : maximum number of bytes required to encode
 *                    client code set
 * fixed        : boolean flag indicating if in-stub evaluation is
 *                necessary
 * type_handle  : points to 'idl_cs_convert_t' data structure.  This
 *                is used within a client stub to calculate
 *                conversion buffer size.
 */
typedef struct {
        unsigned32              stag;
        unsigned32              drtag;
        unsigned16              stag_max_bytes;
        unsigned32              client_tag;
        unsigned16              client_max_bytes;
        rpc_ns_handle_t         type_handle;
} rpc_cs_tags_eval_t, *rpc_cs_tags_eval_p_t;
```

```
/*
 * R P C _ C S _ M E T H O D _ E V A L _ T
 *
 * Data structure which is attached to a binding handle at client
 * side, when automatic code set conversion is enabled.  This data
 * includes `rpc_cs_tags_ eval_t' data structure.  The main
 * difference is it includes server's and client's supported code
 * sets, which makes in-stub evaluation faster.
 *
 * method       : connection method between client and server, e.g,
 *                CMIR
 * tags         : rpc_cs_tags_eval_t.  See above.
 * server       : server's supported code sets.
 * client       : client's supported code sets.
 * cs_stub_eval_func  : When 'fixed' is not true, it
 *                      points to code set I14Y evaluation routine.
 */
typedef struct {
        unsigned32              method;
        rpc_cs_tags_eval_t      tags;
        rpc_codeset_mgmt_t      *server;
        rpc_codeset_mgmt_t      *client;
        boolean32               fixed;
        void                    (*cs_stub_eval_func)(unsigned32
                                *p_stag, unsigned32 *p_drtag,
                                error_status_t *status);
} rpc_cs_method_eval_t, *rpc_cs_method_eval_p_t;

/*
 * Available 'method': Code sets interoperability connection models
 * These values are used only within the RPC runtime.
 */
#define RPC_EVAL_NO_CONVERSION          0x0001
#define RPC_EVAL_RMIR_MODEL             0x0002
#define RPC_EVAL_CMIR_MODEL             0x0003
#define RPC_EVAL_SMIR_MODEL             0x0004
#define RPC_EVAL_INTERMEDIATE_MODEL     0x0005
#define RPC_EVAL_UNIVERSAL_MODEL        0x0006


/* Appending evaluation function pointer(s) to a import context
 * handle.
 *  - This is a modification to the current import context which is
 *    returned from rpc_ns_binding_import_begin().
 *  - New list structure will be allocated by new
 *    rpc_ns_import_ctx_add_eval() routine, and is appended to a
 *    context handle, which is passed to rpc_ns_binding_import_next()
 *    routine.
 *  - At the beginning to rpc_ns_binding_import_next() routine,
 *    evaluation function will be searched from the list, and be
 *    executed.
 *  - argument of a function is an input only void pointer.
 *    This is not currently used by the RPC runtime implementation.
 *  - cntx is an input/output void pointer.  Within the RPC runtime,
 *    this pointer points to rpc_cs_codeset_i14y_data.
```

```
 *  - cs_free_func is called from rpc_ns_binding_import_done(), if it
 *    is set.  This is not used by the RPC runtime implementation.
 */

typedef struct rpc_eval_lists   *rpc_cs_eval_list_p;

typedef struct rpc_eval_lists {
        unsigned32              type;
        void                    (*eval_func)(handle_t binding_h, void
                                *args, void **cntx);
        void                    (*cs_free_func)(void *cntx);
        void                    *args;
        void                    *cntx;
        rpc_cs_eval_list_p      next;
} rpc_cs_eval_list_t, *rpc_cs_eval_list_p_t;

/*
 * R P C _ C S _ C O D E S E T _ I 1 4 Y _ D A T A
 *
 * Argument to OSF code set evaluation routine.  This data will be
 * passed to the evaluation routine, and is used for figuring out the
 * compatible client and server code sets combination.  The
 * evaluation routine will be called from a client * in OSF
 * implementation, and it will not be called from a client stub.
 *
 * ns_name      : NSI entry name for a server
 * cleanup      : boolean flag indicating any clean-up action
 *                required.
 * method_p     : pointer to 'rpc_cs_method_eval_t' data.  See above.
 * status       : result of the code set evaluation.
 */
typedef struct codeset_i14y_data {
        unsigned_char_p_t       ns_name;
        void                    *args;
        boolean32               cleanup;
        rpc_cs_method_eval_p_t  method_p;
        error_status_t          status;
} rpc_cs_codeset_i14y_data, *rpc_cs_codeset_i14y_data_p;
```

8.3. Data Structures Related to NSI Attributes Management

```
/* NS Attribute string constants
 *  - There are currently five attributes defined.  So at the end of
 *    these attributes, rpc_c_ns_codesets attribute is appended.
 * (nsp.h)
 */

#define RPC_C_NS_DNA_TOWERS     ((unsigned_char_p_t) "1.3.22.1.3.30")
#define RPC_C_NS_CLASS_VERSION  ((unsigned_char_p_t) "1.3.22.1.1.1")
#define RPC_C_NS_OBJECT_UUIDS   ((unsigned_char_p_t) "1.3.22.1.1.2")
#define RPC_C_NS_GROUP          ((unsigned_char_p_t) "1.3.22.1.1.3")
#define RPC_C_NS_PROFILE        ((unsigned_char_p_t) "1.3.22.1.1.4")
#define RPC_C_NS_CODESETS       ((unsigned_char_p_t) "1.3.22.1.1.5")

#define RPC_C_ATTR_CLASS_VERSION    0
#define RPC_C_ATTR_DNA_TOWERS       1
```

```
#define RPC_C_ATTR_OBJECT_UUIDS     2
#define RPC_C_ATTR_GROUP            3
#define RPC_C_ATTR_PROFILE          4
#define RPC_C_ATTR_CODESETS         5
#define RPC_C_ATTR_MAX              6

/* Addition to 'Name Service Attributes String constants'
 * - these names are converted to opaque string internally.
 * (nsp.c)
 */
/*
 * Global multi-dimensioned array to hold the architected Name
 * Service attributes string constants and their equivalent Name
 * Service (opaque) format.
 */

GLOBAL rpc_ns_attributes_t
    rpc_g_attr_table[RPC_C_NS_MAX][RPC_C_ATTR_MAX] =
{
    {
        {RPC_C_NS_CLASS_VERSION},
        {RPC_C_NS_DNA_TOWERS},
        {RPC_C_NS_OBJECT_UUIDS},
        {RPC_C_NS_GROUP},
        {RPC_C_NS_PROFILE},
        {RPC_C_NS_CODESETS}
    }
};

/*
 * Well-known UUID for code set attribute
 *
 * From CR 11294
 *   For compatibility with the other DCE attribute stuff (that is,
 *   the xattrschema stuff from ERA and used by DCED), the attr_type
 *   parameter in the rpc_ns_mgmt_attr API calls should uuid_t *, not
 *   unsigned32.  The API should be changed and a new well-known uuid
 *   should be defined for the codeset attribute.
 *
 *   For now, the RPC code can internally map the uuid into an
 *   integer.  In 1.2 or later we should revisit this issue in a more
 *   global scale.
 */
#define rpc_c_uuid_codesets_string \
        "a1794860-a955-11cd-8443-08000925d3fe"

extern uuid_t rpc_c_attr_real_codesets;
extern uuid_t *rpc_c_attr_codesets;

/* Additional types of RPC memory objects
 * (rpcmem.h)
 */

#define RPC_C_MEM_AVAIL             0
/*......*/
```

```
#define RPC_C_MEM_FUNC              86       /* rpc_eval_func_t    */
#define RPC_C_MEM_EVAL              87       /* rpc_binding_eval_t */
#define RPC_C_MEM_LIST              88       /* rpc_eval_list_t    */

/* can only use up to "rpc_c_mem_maxtypes - 1" without upping it */
#define rpc_c_mem_max_types    90     /* i.e., 0...(max_types - 1) */
```

9. USER INTERFACES

DCE Shell's `nsi-entry` class will include a method to manipulate a code set attribute in CDS namespace. The details of the method TBD.

10. API'S

10.1. Interface to the Local Operating System

```
/* This function will allocate rpc_codeset_mgmt_p_t data, and fills
 * the supported code sets values.  It uses nl_langinfo() to
 * determine the current code set, then access the OSF code set
 * registry to determine which code sets are supported on the system.
 */

void rpc_rgy_get_codesets (
   /* [OUT] */  rpc_codeset_mgmt_p_t    codesets_p,
   /* [OUT] */  error_status_t          *status
);
```

10.2. NSI Management

```
/* Management routines to register (code sets) attributes to CDS.
 * rpc_ns_mgmt_set_attribute()
 *  - This routine takes attribute type and a pointer to the value,
 *    and add it to CDS entry.  pointer to the value is declared as
 *    'void', means this routine is designed not only for code sets
 *    attributes, but also for other opaque user supplied values.
 *
 * Reading these attributes:
 *  rpc_ns_mgmt_read_attr_begin()
 *   - This routine is called to setup inquiry context.
 *  rpc_ns_mgmt_read_attr_next()
 *   - This routine reads attributes based on the inquiry context.
 *  rpc_ns_mgmt_read_attr_done() routines are used.
 *   - This routine finishes the operation.
 *
 *  rpc_ns_mgmt_read_codesets()
 *   - A convenience function for reading 'code set' attribute.
 *     This routine calls the above three routines internally.
 *
 * rpc_ns_mgmt_remove_attribute()
 *  - This routine is similar to rpc_ns_mgmt_binding_unexport()
 *    It takes entry name & syntax, and attribute type, then
 *    remove the value of that attribute.
 */

void rpc_ns_mgmt_set_attribute (
   /* [IN] */   unsigned32              entry_name_syntax,
   /* [IN] */   unsigned_char_p_t       entry_name,
```

```
   /* [IN] */   unsigned32              attr_type,
   /* [IN] */   void                    *attr_val,
   /* [OUT] */  error_status_t          *status
);

/* This is very similar to rpc_ns_entry_object_inq_begin except it
 * takes attr_type as a signature.  attr_type is rpc_c_attr_codesets.
 * other types will result rpc_s_mgmt_op_disallowed error.
 */

void rpc_ns_mgmt_read_attr_begin (
   /* [IN] */   unsigned32              entry_name_syntax,
   /* [IN] */   unsigned_char_p_t       entry_name,
   /* [IN] */   uuid_p_t                attr_type,
   /* [OUT] */  rpc_ns_handle_t         *inquiry_context,
   /* [OUT] */  error_status_t          *status
);

/* This is very similar to rpc_ns_entry_object_inq_next but it
 * is designed to handle arbitrary string.
 * attr_bytes will be rpc_c_attr_codesets.
 * other types will result rpc_s_mgmt_op_disallowed error.
 * When value is returned, it is cast-ed to (void *), but it will be
 * rpc_codeset_mgmt_p_t
 */

void rpc_ns_mgmt_read_attr_next (
   /* [IN] */   rpc_ns_handle_t         inquiry_context,
   /* [IN] */   uuid_p_t                attr_type,
   /* [OUT] */  void                    **value,
   /* [OUT] */  unsigned32              *length,
   /* [OUT] */  error_status_t          *status
);

/* This is very similar to rpc_ns_entry_object_inq_done.
 * Only difference is that it will checkrpc_e_codesets_member
 * as context usage.
 */

void rpc_ns_mgmt_read_attr_done (
   /* [IN] */   rpc_ns_handle_p_t       inquiry_context,
   /* [OUT] */  error_status_t          *status
);

/* This is a convenience routine for reading the 'code set' attribute
 * value
 */

void rpc_ns_mgmt_read_codesets (
   /* [IN] */   unsigned32              entry_name_syntax,
   /* [IN] */   unsigned_char_p_t       entry_name,
   /* [OUT] */  rpc_codeset_mgmt_p_t    codeset_val,
   /* [OUT] */  error_status_t          *status
);
```

```
void rpc_ns_mgmt_remove_attribute (
   /* [IN] */   unsigned32              entry_name_syntax,
   /* [IN] */   unsigned_char_p_t       entry_name,
   /* [IN] */   uuid_p_t                attr_type,
   /* [OUT] */  error_status_t          *status
);
```

10.3. Evaluation

```
/* This function will allocate rpc_cs_eval_func_t data and list data.
 * Then it sets code set compatibility evaluation function pointer in
 * a list structure (rpc_cs_eval_list_t).
 * Application programmer is responsible for providing desired
 * code set compatibility evaluation function, or routines which OSF
 * provided can be used.
 * This evaluation function is called from import context which is
 * allocated by rpc_ns_binding_import_begin().
 * func_type is either RPC_EVAL_TYPE_CODESETS or
 * RPC_CUSTOM_EVAL_TYPE_CODESETS for a user written evaluation
 * routine for now, but it can be extended in the future.
 */

void rpc_ns_import_ctx_add_eval
(
    /* [IN/OUT] */ rpc_ns_handle_t  *import_ctx,
    /* [IN] */ unsigned32           func_type,
    /* [IN] */ void                 *args,
    /* [IN] */ void                 (*eval_func)(handle_t binding_h,
                                    void *args, void **cntx),
    /* [IN] */ void                 (*cs_free_func)(void *cntx),
    /* [OUT] */ error_status_t      *status
);

/* Sample code set compatibility evaluation functions.
 * OSF provides these functions for examples.  Application developers
 * are free to write their own evaluation functions, as long as
 * they don't change the signature.
 *
 * Or application developers can use rpc_cs_binding_set_tags()
 * routine to set the tags directly into a binding handle to perform
 * a proper code set conversion as a part of marshalling and
 * unmarshalling.
 */

/*
 * R P C _ C S _ E V A L _ W I T H _ U N I V E R S A L
 *
 * Code set interoperability evaluation routine.  If none of the
 * client and server code sets match, Universal code set will be used
 * for communication.  'args' is not used, and 'cntx' points to
 * 'rpc_cs_codeset_i14y_data'.
 */
void rpc_cs_eval_with_universal
(
    /* [IN] */ handle_t       binding_h,
    /* [IN] */ void           *args,
```

```
    /* [IN/OUT] */ void       **cntx
);

/*
 * R P C _ C S _ E V A L _ W I T H O U T _ U N I V E R S A L
 *
 * Code set interoperability evaluation routine.  If none of the
 * client and server code sets match, evaluation will fail.  'args'
 * is not used, and 'cntx' points to 'rpc_cs_codeset_i14y_data'.
 */
void rpc_cs_eval_without_universal
(
    /* [IN] */ handle_t      binding_h,
    /* [IN] */ void          *args,
    /* [IN/OUT] */ void      **cntx
);

/* These is no signature change to rpc_ns_binding_import_next(),
 * however, new logic will be added within the routine to check if
 * import_context contains rpc_cs_eval_func_t structure.  If it is,
 * the new logic will dereference code set evaluation function
 * pointer to execute it.  This code set evaluation function will
 * return a model of client server communication as well as the tags
 * used for a code set conversion.  A model value and tags
 * information will be appended to binding handle to be referenced by
 * stubs.
 */

void rpc_ns_binding_import_next (
        /* [IN] */ rpc_ns_handle_t              import_ctxt,
        /* [OUT] */ rpc_binding_handle_t        *binding,
        /* [OUT] */ error_status_t              *status
);

/* These is no signature change to rpc_ns_binding_import_done(),
 * however, new logic will be added within the routine to check if
 * import_context is contains rpc_cs_eval_func_t structure.  If it
 * is, the new logic will dereference code set evaluation clean-up
 * function pointer to execute it.  The evaluation routine is only
 * executed if the clean-up function pointer is set.  In case of code
 * set evaluation, there is no clean-up function.  This feature is
 * added for future use.
 */

void rpc_ns_binding_import_done (
        /* [IN] */ rpc_ns_handle_t              import_ctxt,
        /* [OUT] */ error_status_t              *status
);

/*
 * Internal routine to attach the code set interoperability
 * attributes to a binding handle.  This routine is not intended to
 * be used by application developers.  Only runtime uses it.
 */
extern void rpc_cs_binding_set_method (
```

```
        /* [IN/OUT] */ rpc_binding_handle_t     *h,
        /* [IN] */ rpc_cs_method_eval_p_t       method_p,
        /* [OUT] */ error_status_t              *status
);

/*
 * Set the tags value into rpc binding handle.
 *
 * This routine is provided for the application developers who wish
 * to perform a character and code set compatibility evaluation by
 * themselves.  For code set conversion routine to be invoked from
 * stubs, tags information need to be attached to the rpc binding
 * handle.  This routine does not perform any compatibility
 * evaluation for the application.  The application developers need
 * to supply the values to the routine.
 *  stag:             sending tag (from a client)
 *  drtag:            desired receiving tag (from a client)
 *  stag_max_bytes:   maximum number of bytes to encode a code set
 *                    specified for 'stag'
 */

void rpc_cs_binding_set_tags (
        /* [IN/OUT] */ rpc_binding_handle_t     *h,
        /* [IN] */ unsigned32                   stag,
        /* [IN] */ unsigned32                   drtag,
        /* [IN] */ unsigned16                   stag_max_bytes,
        /* [OUT] */ error_status_t              *p_st
);
```

10.4. Tags Set Routine

```
/* This routine takes binding handle, and determines the tags value
 * depending of a model information appended to a binding handle.
 *
 * The signature of this routine is defined by DEC [DEC 1].
 */

void rpc_cs_get_tags (
        /* [IN] */ rpc_binding_handle_t       h,
        /* [IN] */ boolean                    server_side,
        /* [OUT] */ unsigned32                *p_stag,
        /* [IN/OUT] */ unsigned32             *p_drtag,
        /* [OUT] */ unsigned32                *p_rtag,
        /* [OUT] */ error_status_t            *p_st
);

/*
 * p_stag  -> [OUT] client side, ignored server side
 * p_drtag -> [OUT] client side, [IN] server side
 * p_rtag  -> [OUT] server side, ignored client side
 */
```

10.5. Buffer Sizing and Code Set Conversion

```
/* This routine takes binding handle, tag and local storage size, and
 * calculate the necessary buffer size.  This needs to be done before
 * marshalling of data.
```

```
 *
 * The signatures of these routines are defined by DEC.
 */

/*
 * cs_byte is used as I18N byte type from I18N applications.
 */
typedef ndr_byte cs_byte;

void cs_byte_net_size (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ unsigned32                   l_storage_len,
        /* [OUT] */ idl_cs_convert_t            *p_convert_type,
        /* [OUT] */ unsigned32                  *p_w_storage_len,
        /* [OUT] */ error_status_t              *p_st
);

void wchar_t_net_size (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ unsigned32                   l_storage_len,
        /* [OUT] */ idl_cs_convert_t            *p_convert_type,
        /* [OUT] */ unsigned32                  *p_w_storage_len,
        /* [OUT] */ error_status_t              *p_st
);

/* These routines convert data from local format to on-the-wire
 * format.  These routines are used from the stubs.
 */

void cs_byte_to_netcs (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ idl_byte                     *ldata,
        /* [IN] */ unsigned32                   l_data_len,
        /* [OUT] */ idl_byte                    *wdata,
        /* [OUT] */ unsigned32                  *p_w_data_len,
        /* [OUT] */ error_status_t              *p_st
);

void wchar_t_to_netcs (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ wchar_t                      *ldata,
        /* [IN] */ unsigned32                   l_data_len,
        /* [OUT] */ idl_byte                    *wdata,
        /* [OUT] */ unsigned32                  *p_w_data_len,
        /* [OUT] */ error_status_t              *p_st
);

/* These routine take binding handle, tag and wire storage size, and
 * calculate the necessary buffer size.  This needs to be done before
 * unmarshalling of data.
 */
```

```
void cs_byte_local_size (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ unsigned32                   w_storage_len,
        /* [OUT] */ idl_cs_convert_t            *p_convert_type,
        /* [OUT] */ unsigned32                  *p_l_storage_len,
        /* [OUT] */ error_status_t              *p_st
);

void wchar_t_local_size (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ unsigned32                   w_storage_len,
        /* [OUT] */ idl_cs_convert_t            *p_convert_type,
        /* [OUT] */ unsigned32                  *p_l_storage_len,
        /* [OUT] */ error_status_t              *p_st
);

/* These routines convert data from on-the-wire format to local
 * format.  This is called from stubs.
 */

void cs_byte_from_netcs (
        /* [IN] */ rpc_binding_handle_t         h,
        /* [IN] */ unsigned32                   tag,
        /* [IN] */ idl_byte                     *wdata,
        /* [IN] */ unsigned32                   w_data_len,
        /* [IN] */ unsigned32                   l_storage_len,
        /* [OUT] */ idl_byte                    *ldata,
        /* [OUT] */ unsigned32                  *p_l_data_len,
        /* [OUT] */ error_status_t              *p_st
);

void wchar_t_from_netcs (
        /* [IN] */ rpc_binding_handle_t         h;
        /* [IN] */ unsigned32                   tag;
        /* [IN] */ idl_byte                     *wdata;
        /* [IN] */ unsigned32                   w_data_len;
        /* [IN] */ unsigned32                   l_storage_len;
        /* [OUT] */ wchar_t                     *ldata;
        /* [OUT] */ unsigned32                  *p_l_data_len;
        /* [OUT] */ error_status_t              *p_st;
);
```

10.6. Character and Code Set Registry

```
/* These routines are for the OSF Character and Code Set Registry
 * access.  Since iconv only accepts string as code set name, while
 * code set are represented in integer for internal operations, we
 * need string <-> int value conversion routines.  Also this registry
 * ensures code set compatibility between vendors, and in the
 * heterogeneous network environments.
 *
 * Character sets are used to determine the compatibility for code
 * sets which encode multiple character sets.  For example, eucJP can
 * encode ASCII as well as Katakana(JIS0201) and Kanji(JIS0208).
```

```
 */

/*
 * Convert string code set name to integer code set value
 */

void dce_cs_loc_to_rgy (
        /* [IN] */  const char                *local_code_set_name,
        /* [OUT] */ unsigned32                *rgy_code_set_value,
        /* [OUT] */ unsigned16                *rgy_char_sets_number,
        /* [OUT] */ unsigned16                *rgy_char_sets_value[],
        /* [OUT] */ error_status_t            *p_st
);

/*
 * Convert integer code set value to string code set name
 */

void dce_cs_rgy_to_loc (
        /* [IN] */  unsigned32                rgy_code_set_value,
        /* [OUT] */ char                      **local_code_set_name,
        /* [OUT] */ unsigned16                *rgy_char_sets_number,
        /* [OUT] */ unsigned16                **rgy_char_sets_value,
        /* [OUT] */ error_status_t            *status
);

/*
 * Get the code set’s maximum number of bytes per a character.
 * This routine is called by the RPC runtime routine to calculate the
 * necessary buffer size, as well as used by the application
 * developers before rpc_cs_binding_set_tags() routine.
 */

void rpc_rgy_get_max_bytes (
        /* [IN] */  unsigned32                tag,
        /* [OUT] */ unsigned16                *max_bytes,
        /* [OUT] */ error_status_t            *status
);

/*
 * Free a codesets array allocated by the runtime.  Server should
 * call this routine after it exported code sets array to the name
 * space, to avoid a memory leak.
 */
void rpc_ns_mgmt_free_codesets (
        /* [IN/OUT] */ rpc_codeset_mgmt_p_t   *codesets,
        /* [OUT] */    error_status_t         *status
);
```

11. REMOTE INTERFACES

The followings are the interface definition and ACF for IDL Encoding Services routines used in NSI management to encode and decode code sets attribute.

11.1. IDL File

```
interface rpc_nsi_management
{
        /* Encoding routine */
        void rpc__codesets_to_nscodesets (
                [in] handle_t                  h,
                [in] long                      num,
                [in, size_is(num)] byte        codesets[]
        );

        /* Decoding routine */
        void rpc__nscodesets_to_codesets (
                [in] handle_t                h,
                [in, out] long               *num,
                [out, size_is(*num)] byte    codesets[]
        );
}
```

11.2. ACF File

```
interface rpc_nsi_management
{
        [encode] rpc__codesets_to_nscodesets();
        [decode] rpc__nscodesets_to_codesets();
}
```

12. MANAGEMENT INTERFACES

Not applicable

13. RESTRICTIONS AND LIMITATIONS

(a) `auto_handle` and Customized handles cannot be used with the automatic code set conversion feature.

(b) `pipe` type cannot be used with the automatic code set conversion feature.

(c) Automatic code set conversion feature cannot be used when an array has more than one dimension, or any of the attribute `min_is`, `max_is`, `first_is`, `last_is` or `string` has been applied to an array.

(d) Automatic code set conversion feature cannot be used with `transmit_as` or `represent_as` attributes.

(e) Automatic code set conversion feature cannot be applied to a type which is the target of a pointer with the `[size_is]` or `[max_is]` attribute. For example, following IDL fragment is not acceptable.

```
typedef byte my_byte;

typedef struct {
      long s;
      [size_is(s)] my_byte *arrayfield;
} problem_struct;
```

But the same information can be carried by the following.

```
typedef struct {
      long s;
      [size_is(s)] my_byte arrayfield[];
} conf_struct;
```

(f) If an array parameter has a `[cs_char]` base type, it cannot have `[ptr]` or `[unique]` as parameter attributes. This forbids the cases illustrated by

```
void wrong_proc (
        [in] long s1,
        [in, unique, size_is(s1)] my_byte ua[],
        [in] long s2,
        [in, ptr, size_is(s2)] my_byte pa[]
);
```

in which the arrays are being treated as pointer targets. It does not forbid

```
void right_proc (
        [in] long s1,
        [in, size_is(s1)] my_byte ra[]
);
```

(g) The use of `wchar_t` type has a potential problem regarding the conversion between `byte` and `wchar_t` data types.

Current implementation for `wchar_t_to_netcs` has the following signature.

```
PUBLIC void wchar_t_to_netcs
(
        rpc_binding_handle_t     h,
        unsigned32               tag,
        wchar_t                  *ldata,
        unsigned32               l_data_len,
        idl_byte                 *wdata,
        unsigned32               *p_w_data_len,
        error_status_t           *status
)
```

`l_data_len` is the number of `wchar_t` elements in `ldata`. Since we will not send `wchar_t` type data over the network (`wchar_t` is a process code, which is not compatible with various operating systems), we have to convert wchar_t data type to multibyte data type.

When this conversion takes place, the resulting multibyte length is different from `l_data_len`. In case of fixed length array and varying length array, `p_w_data_len` is NULL. IDL library assumes `l_data_len` will be the length of network data. (And that's the specification of international character support in IDL). The length of multibyte string is usually larger than the length of `wchar_t` string.

So when server receives multibyte string, it is usually shorter than the entire multibyte string. Consequently, at the server side, when this multibyte string is converted to `wchar_t` string, `invalid character error` occurs.

14. OTHER COMPONENT DEPENDENCIES

(a) I18N IDL complier extension will be done at DEC Nashua, so the completion of this work depends on their work.

(b) IDL Encoding Services is a part of 1.1 deliverables from DEC. Encoding / decoding a code set attribute into / from an endian-safe format depends on this new feature.

(c) DCE Shell is a new facility for 1.1 release. A manipulation of code set attributes by DCE Shell command depends on its availability.

15. COMPATIBILITY

(a) DCE 1.0 IDL compilers will not understand new syntax for I18N characters.

(b) DCE 1.0 RPC library will not support the new data structures and new APIs.

16. STANDARDS

(a) POSIX 1003.2 / XPG4 I18N facilities and functionalities. (locale, `iconv`, and other library routines).

17. OPEN ISSUES

(a) The DCE Shell code set manipulation interface (method) is not yet decided. Even though `dcecp` is available today, OSF does not have time and resources to implement a method to perform, add, show, export, and delete operations for the code set attribute. This can be done at later releases.

APPENDIX A. SAMPLE IDL AND ACF USING NEW FEATURES

A.1. Fixed Array Example

A very simple example, which exchanges fixed arrays of characters as both input and output, will have IDL and ACF files like the following.

Even though the tags are exposed in the RPC signature, actual tags values will be set by `[cs_tag_rtn(set_tags)]` in ACF definition. So application developers do not need to be aware of the actual tag values.

IDL:

```
[
uuid(001bb502-2190-1be8-b507-08002b0f59bb),
version(1.0)
]
interface fixed_string_example
{
   const long SIZE = 100;
   typedef byte my_byte;

   void fixed_string(
       [in] unsigned long stag,
       [in] unsigned long drtag,
       [out] unsigned long *p_rtag,
       [in,out] my_byte my_string[SIZE]
   );
}
```

ACF:

```
[
implicit_handle(handle_t global_binding_h)
]
interface fixed_string_example
{
       typedef [cs_char(cs_byte)] my_byte;

       [cs_tag_rtn(rpc_cs_get_tags)] fixed_string(
                                          [cs_stag] stag,
                                          [cs_drtag] drtag,
                                          [cs_rtag] p_rtag );
}
```

A.2. Conformant Varying Array Example

DEC Design Note [DEC 1] explains the detail of the IDL features. So this is the excerpt from the document. This example is for conformant varying array as both `[in]` and `[out]` argument.

Suppose an IDL file contains:

```
typedef byte my_byte;

void a_op(
        [in] unsigned long stag,
        [in] unsigned long drtag,
        [out] unsigned long *p_rtag,
        [in] long s,
        [in, out] long *p_l,
        [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);

void b_op(
        [in] unsigned long stag,
        [in] unsigned long drtag,
        [out] unsigned long *p_rtag,
        [in] long s,
        [in, out] long *p_l,
        [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);
```

and the associated ACF file contains:

```
/* ltype can be wchar_t, byte, or application defined type */
typedef [cs_char(cs_byte)] my_byte;

[cs_tag_rtn(rpc_cs_get_tags)] a_op( [cs_stag] stag,
                            [cs_drtag] drtag,
                            [cs_rtag] p_rtag );

                      b_op( [cs_stag] stag,
                            [cs_drtag] drtag,
                            [cs_rtag] p_rtag );
```

Then the generated header file will contain:

```
typedef idl_byte my_byte;

void a_op(
       /* [in] */      idl_long_int     s,
       /* [in, out] */ idl_long_int     *p_l,
       /* [in, out, size_is(s), length_is(*p_l)] */ cs_byte a[]
);

void b_op(
       /* [in] */      idl_ulong_int    stag,
       /* [in] */      idl_ulong_int    drtag,
       /* [out] */     idl_ulong_int    *p_rtag,
       /* [in] */      idl_long_int     s,
       /* [in, out] */ idl_long_int     *p_l,
       /* [in, out, size_is(s), length_is(*p_l)] */ cs_byte a[]
);
```

The stubs for a_op will call rpc_cs_get_tags before marshalling data. For b_op it is the application code's responsibility to make sure that the tags are set correctly before data is marshalled. [DEC 1]

If application developers decide to set tags by themselves, they can use

(a) new routine (rpc_rgy_get_codesets()) to get supported code sets information,

(b) new routine (rpc_ns_mgmt_read_codesets()) to read the code set information; and

(c) set tags based on supported code set information and CDS code set attribute value by using (rpc_cs_binding_set_tags()). It might be necessary to write code set compatibility evaluation function to set tags value properly.

APPENDIX B. SAMPLE SERVER AND CLIENT CODE

B.1. Server Side

Using this modification to RPC application development, the new server flow will be like the following (> indicates the new steps):

(a) Register the interface

(b) Create binding information

(c) Obtain the binding information

(d) Advertise the server location in the name service database

\> (e) Get server's supported code sets

\> (f) Register the code sets attributes to CDS

(g) Register the endpoints in the local endpoint map

(h) Free the set of binding handles

(i) Listen for remote call

\> (j) Remove the code sets attributes from CDS

The simple server code is shown in the following subsection. Note, however, to show the steps clearly, these codes omit necessary error checkings on purpose.

B.1.1. Sample server code

```
#include <dce/rpc.h>
#include <dce/nsattrid.h>       /* defines well-known uuid for
                                   rpc_c_attr_codesets */
#include <locale.h>

void main (void)
{
        unsigned32              status;
        rpc_binding_vector_t    *binding_vector;
        unsigned_char_t         *entry_name = "/.:/my_app_entry";

        rpc_codeset_mgmt_p_t    codesets;

        setlocale(LC_ALL, "");     /* <-- NOTE */

        rpc_server_register_if (
                myif_v1_0_s_ifspec,             /* if_handle */
                NULL,                           /* mgr_type_uuid */
                NULL,                           /* mgr_epv */
```

```
            &status );

    rpc_server_use_all_protseqs (
            rpc_c_protseq_max_reqs_default, /*max_calls_request*/
            &status );

    rpc_server_inq_bindings (
            &binding_vector,
            &status );

    rpc_ns_binding_export (
            rpc_c_ns_syntax_default,       /* entry_name_syntax*/
            entry_name,
            myif_v1_0_s_ifspec,            /* if_handle */
            binding_vector,
            NULL,                          /* object_uuid_vec */
            &status );

    rpc_reg_get_codesets (          /* <-- NOTE */
            &codesets,                     /* [OUT] codesets */
            &status );

    rpc_ns_mgmt_set_attribute (     /* <-- NOTE */
            rpc_c_ns_syntax_default,       /* name entry syntax*/
            entry_name,                    /* name entry */
            rpc_c_attr_codesets,           /* attribute type */
            (void *)codesets,             /* attribute_value */
            &status );

    rpc_ns_mgmt_free_codesets (     /* <- NOTE */
            &codesets,
            &status );

    rpc_ep_register (
            myif_v1_0_s_ifspec,            /* if_handle */
            binding_vector,
            NULL,                          /* object_uuid_vec */
            NULL,                          /* annotation */
            &status );

    rpc_binding_vector_free (
            &binding_vector,
            &status );

    rpc_server_listen (
            rpc_c_listen_max_calls_default, /* max_calls_exec */
            &status );

    rpc_ns_mgmt_remove_attribute (     /* <-- NOTE */
            rpc_c_ns_syntax_default,
            entry_name,
            rpc_c_attr_codesets,
            &status );
}
```

B.2. Client Side

The following example uses *implicit binding* method, but an application can also use *explicit binding* method when it is appropriate.

The simple client code is shown in the following subsection. Note, however, to show the steps clearly, these codes omit necessary error checkings on purpose.

B.2.1. Client side code

```
#include <dce/rpc.h>
#include <locale.h>
#include "cs_test.h"

main ( )
 {
        cs_byte                 in_out_string[100];
        unsigned32              status;
        rpc_ns_handle_t         import_context;


        setlocale(LC_ALL, "");          /* <-- NOTE */

        rpc_ns_binding_import_begin (
                rpc_c_ns_syntax_default,        /* defined in nsp.h*/
                "/.:/my_app_entry",
                myif_v1_0_c_ifspec,             /* if_handle */
                NULL,
                &import_context,
                &status );

        rpc_ns_import_ctx_add_eval (     /* <-- NOTE */
                &import_context,
                rpc_c_eval_type_codesets,
                (void *)"/.:/my_app_entry",
                rpc_cs_eval_with_universal,
                NULL,
                &status );

        while (1) {

                rpc_ns_binding_import_next (
                        import_context,
                        &global_binding_h,      /* declared in ACF */
                        &status );

                if (status == rpc_s_ok)
                        break;

                if (status == rpc_s_no_more_bindings)
                        exit(1);
        }

        rpc_ns_binding_import_done (
                        &import_context,
                        &status );
```

```
        my_rpc (in_out_string);
}
```

APPENDIX C. CUSTOMIZED ROUTINES

Section for "client development" mentioned four types of routines which are possibly written by application developers. This section explains each of them in more detail.

C.1. Set Tags

The signature of this routine in the RPC runtime library is:

```
void rpc_cs_get_tags (
        /* [IN] */ rpc_binding_handle_t       h,
        /* [IN] */ boolean                    server_side,
        /* [OUT] */ unsigned32                *p_stag,
        /* [IN/OUT] */ unsigned32             *p_drtag,
        /* [OUT] */ unsigned32                *p_rtag,
        /* [OUT] */ error_status_t            *p_st
);
```

Since this routine is specified in ACF like:

```
[cs_tag_rtn(rpc_cs_get_tags)] a_op ( [cs_stag] stag,
                                     [cs_drtag] drtag,
                                     [cs_rtag] p_rtag );
```

application developers can write private routine to replace rpc_cs_get_tags() routine. In this case, the ACF needs to be modified accordingly.

C.2. Calculate Buffer Size

The signature of routines in the RPC runtime library are ("local_data_type" is either `cs_byte` or `wchar_t`):

```
void local_data_type_net_size (
        /* [IN] */ rpc_binding_handle_t       h,
        /* [IN] */ unsigned32                 tag,
        /* [IN] */ unsigned32                 l_storage_len,
        /* [OUT] */ idl_cs_convert_t          *p_convert_type,
        /* [OUT] */ unsigned32                *p_w_storage_len,
        /* [OUT] */ error_status_t            *p_st
);

void local_data_type_local_size (
        /* [IN] */ rpc_binding_handle_t       h,
        /* [IN] */ unsigned32                 tag,
        /* [IN] */ unsigned32                 w_storage_len,
        /* [OUT] */ idl_cs_convert_t          *p_convert_type,
        /* [OUT] */ unsigned32                *p_l_storage_len,
        /* [OUT] */ error_status_t            *p_st
);
```

Application developers can define unique data type in the ACF as follows:

```
/* kanji is defined as
 *        typedef byte kanji;
 * in IDL.
 * 'kanji' is used as network encoding, which actually travels
 * over the wire.
 */
```

```
typedef [cs_char(my_byte)] kanji;
```

In this case, `my_byte_net_size()` and `my_byte_local_size()` routines need to be written. These routines will be called from stubs. The output should include a convert type and a required storage length. Convert types are defined in <dce/idlbase.h> in the OSF sample implementation, and are defined as the following data structure:

```
typedef enum {
     idl_cs_no_convert,         /* No code set conversion
                                   required */
     idl_cs_in_place_convert,   /* Code set conversion can be
                                   done in a single storage
                                   area */
     idl_cs_new_buffer_convert  /* The converted data must be
                                   written to a new storage
                                   area */
} idl_cs_convert_t;
```

C.3. Convert Code Sets

The signature of routines in the RPC runtime library are (*`local_data_type`* is either `cs_byte` or `wchar_t`):

```
void local_data_type_to_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
        /* in */   local data type             *ldata,
        /* in */   unsigned32                   l_data_len,
        /* out */  network data type           *wdata,
        /* out */  unsigned32                   *p_w_data_len,
        /* out */  error_status_t               *p_st
);
void local_data_type_from_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
        /* in */   network data type           *wdata,
        /* in */   unsigned32                   w_data_len,
        /* in */   unsigned32                   l_storage_len,
        /* out */  local data type             *ldata,
        /* out */  unsigned32                   *p_l_data_len,
        /* out */  error_status_t               *p_st
);
```

When `my_byte` is defined as locale data type, and `kanji` is defined as network data type, the following two routines are required:

```
void my_byte_to_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
        /* in */   my_byte                      *ldata,
        /* in */   unsigned32                   l_data_len,
        /* out */  kanji                        *wdata,
        /* out */  unsigned32                   *p_w_data_len,
        /* out */  error_status_t               *p_st
);
void my_byte_from_netcs (
        /* in */   rpc_binding_handle_t         h,
        /* in */   unsigned32                   tag,
```

```
            /* in */   kanji                          *wdata,
            /* in */   unsigned32                     w_data_len,
            /* in */   unsigned32                     l_storage_len,
            /* out */  my_byte                        *ldata,
            /* out */  unsigned32                     *p_l_data_len,
            /* out */  error_status_t                 *p_st
    );
```

`my_byte_to_netcs()` takes character string encoded in `my_byte` form, and converts it into `kanji` form based on `tag` value. `kanji` is always resolved into `idl_byte`, so undesirable automatic conversion by DCE (namely ASCII to/from U.S. EBCDIC) will not take place.

Application developers can use any type of conversion procedure available for them, as long as `out` values are set correctly. In the OSF sample implementation, `iconv` routines which are defined by XPG4 are used.

C.4. Evaluate Code Sets Compatibility

The RPC runtime library provides a routine, which allows application developers to set the tags value directly into a binding handle. The signature of the routine is:

```
    void rpc_cs_binding_set_tags (
            /* [IN/OUT] */ rpc_binding_handle_t     *h,
            /* [IN] */ unsigned32                   stag,
            /* [IN] */ unsigned32                   drtag,
            /* [IN] */ unsigned16                   stag_max_bytes,
            /* [OUT] */ error_status_t              *p_st
    );
```

Application developers can implement the appropriate character and/or code set compatibility logic within their application, then use this routine to set the tags value. Following code shows a client which performs character and code set evaluation. Client has to get the binding vector from the RPC runtime, and performs the evaluation on each binding handle within the vector, until it finds the compatible server.

This code also omits the necessary error checking to make the code shorter.

```
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <dce/rpcsts.h>
#include <dce/dce_error.h>

#include "cs_test.h"            /* IDL generated include file */

void
main(void)
{
        rpc_binding_handle_t    bind_handle;
        rpc_ns_handle_t         lookup_context;
        rpc_binding_vector_p_t  bind_vec_p;
        unsigned_char_t         *entry_name;
        unsigned32              binding_count;
        cs_byte                 net_string[SIZE];
        cs_byte                 loc_string[SIZE];
        int                     model_found, smir_true, cmir_true;
        rpc_codeset_mgmt_p_t    client, server;
        unsigned32              stag;
        unsigned32              drtag;
        unsigned16              stag_max_bytes;
        error_status_t          status;
```

```
char                    *nsi_entry_name;
char                    *client_locale_name;

nsi_entry_name = getenv("I18N_SERVER_ENTRY");

setlocale(LC_ALL, "");

rpc_ns_binding_lookup_begin (
        rpc_c_ns_syntax_default,
        (unsigned_char_p_t)nsi_entry_name,
        cs_test_v1_0_c_ifspec,
        NULL,
        rpc_c_binding_max_count_default,
        &lookup_context,
        &status );

rpc_ns_binding_lookup_next (
        lookup_context,
        &bind_vec_p,
        &status );

rpc_ns_binding_lookup_done (
        &lookup_context,
        &status );

/*
 *  Get the client's supported code sets
 */
rpc_rgy_get_codesets (
        &client,
        &status );

binding_count = (bind_vec_p)->count;
for (i=0; i < binding_count; i++)
{
        if ((bind_vec_p)->binding_h[i] == NULL)
               continue;

        rpc_ns_binding_select (
                bind_vec_p,
                &bind_handle,
                &status );

        if (status != rpc_s_ok)
        {
                rpc_ns_mgmt_free_codesets(&client, &status);
        }

        rpc_ns_binding_inq_entry_name (
                bind_handle,
                rpc_c_ns_syntax_default,
                &entry_name,
                &status );

        if (status != rpc_s_ok)
```

```
{
        rpc_ns_mgmt_free_codesets(&client, &status);
}

/*
 *  Get the server’s supported code sets from NSI
 */
 rpc_ns_mgmt_read_codesets (
        rpc_c_ns_syntax_default,
        entry_name,
        &server,
        &status );

if (status != rpc_s_ok)
{
        rpc_ns_mgmt_free_codesets(&client, &status);
}

/*
 *  Start evaluation
 */
if (client->codesets[0].c_set ==
    server->codesets[0].c_set)
{
        /*
         *  client and server are using same code set
         */
        stag = client->codesets[0].c_set;
        drtag = server->codesets[0].c_set;
        break;
}

/*
 *  check character set compatibility first
 */
 rpc_cs_char_set_compat_check (
        client->codesets[0].c_set,
        server->codesets[0].c_set,
        &status );

if (status != rpc_s_ok)
{
        rpc_ns_mgmt_free_codesets(&server, &status);
}

smir_true = cmir_true = model_found = 0;

for (k = 1; k <= server->count; k++)
{
        if (model_found)
                break;

        if (client->codesets[0].c_set
                   == server->codesets[k].c_set)
        {
```

```
                        smir_true = 1;
                        model_found = 1;
                }

                if (server->codesets[0].c_set
                           == client->codesets[k].c_set)
                {
                        cmir_true = 1;
                        model_found = 1;
                }
        }

        if (model_found)
        {
                if (smir_true && cmir_true)
                {
                        /* RMIR model works */
                        stag = client->codesets[0].c_set;
                        drtag = server->codesets[0].c_set;
                        stag_max_bytes =
                             client->codesets[0].c_max_bytes;
                }
                else if (smir_true)
                {
                        /* SMIR model */
                        stag = client->codesets[0].c_set;
                        drtag = client->codesets[0].c_set;
                        stag_max_bytes =
                             client->codesets[0].c_max_bytes;
                }
                else
                {
                        /* CMIR model */
                        stag = server->codesets[0].c_set;
                        drtag = server->codesets[0].c_set;
                        stag_max_bytes =
                             server->codesets[0].c_max_bytes;
                }

                /*
                 *  set tags value to the binding
                 */
                rpc_cs_binding_set_tags (
                        &bind_handle,
                        stag,
                        drtag,
                        stag_max_bytes,
                        &status );

                if (status != rpc_s_ok)
                {
                        rpc_ns_mgmt_free_codesets(&server,
                                                  &status);
                        rpc_ns_mgmt_free_codesets(&client,
                                                  &status);
```

```
                        }
                }
                else
                {
                        /*
                         *  try another binding
                         */
                        rpc_binding_free (
                                &bind_handle,
                                &status );

                        if (status != rpc_s_ok)
                        {
                                rpc_ns_mgmt_free_codesets(&server,
                                                          &status);
                                rpc_ns_mgmt_free_codesets(&client,
                                                          &status);
                        }
                }
        }

        rpc_ns_mgmt_free_codesets(&server, &status);

        rpc_ns_mgmt_free_codesets(&client, &status);

        if (!model_found)
        {
                printf("FAILED No compabile server found0);
                exit(1);
        }

        cs_fixed_trans(bind_handle, net_string, loc_string);
}
```

APPENDIX D. IDL ENCODING SERVICES

IDL Encoding Services are a feature of the IDL compiler which separates out the data marshalling and unmarshalling functionality from the interaction with the RPC runtime.

This extension to the IDL stub compiler will enable instances of one or more data types to be encoded into and decoded from a byte stream format suitable for persistent storage.

For more information of IDL encoding services, refer to [RFC 2.1] and [DEC 2].

REFERENCES

[RFC 42.3] H. Melman ‘‘DCE Shell Functional Specification’’, October, 1994.

[DEC 1] T. Hinxman, ‘‘Design Note RPCLang 015 International Character Support in IDL’’, February 1993.

[DEC 2] T. Hinxman, ‘‘IDL Encoding Services’’.

[OGURA] T. Ogura, ‘‘DCE 1.1 I18N Workbook’’, October, 1992.

[RFC 2.1] J. Harrow, ‘‘Proposed Enhancements for DCE 1.1 IDL’’, July 1992.

[RFC 13.0] A. Thormodsen, ‘‘DCE 1.1 Internationalization Requirements’’, January, 1993.

[RFC 23.0] R. Mackey, ‘‘DCE 1.1 Internationalization Guide’’, January, 1993.

[RFC 27.0] S. Martin, ‘‘Coded Character Set Conversions and Data Loss: Providing Interoperability While Preventing Loss’’, December 1992.

[RFC 34.0] H. Melman, ‘‘DCE 1.1 Coding Style Guide’’, April 1993.

[RFC 39.0] HP/IBM, ‘‘An Internationalized DCE Character Handling Proposal’’, March 1993.

[RFC 40.1] S. O’Donnell, ‘‘OSF Character and Code Set Registry’’, January 1994.

[SHIRLEY] J. Shirley, ‘‘Guide to Writing DCE Applications’’, O’Reilly publishers, June 1992.

[XOPEN] X. Open, ‘‘Internationalisation Guide’’.

[XPG4] X/Open Portability Guide (Version 4) — 1993, System Interfaces and Headers.

AUTHOR’S ADDRESS

Mariko (Mori) Romagna
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
Internet email: mori_m@osf.org
Telephone: +1-617-621-8981

Dick Mackey
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA
Internet email: dmackey@osf.org
Telephone: +1-617-621-8924