OSF DCE SIG **R. Annicchiarico (DEC)**
Request For Comments: 48.0 **August 1993**

C++ SUPPORT IN DCE RPC — FUNCTIONAL OVERVIEW

1. INTRODUCTION

1.1. Existing DCE Support

The OSF Distributed Computed Environment (DCE) provides the capabilities necessary for a C++ programmer to build an object-oriented distributed application. Utilizing these features is not a simple task, however, due to the in-depth knowledge of DCE programming required. Without additional support, the C++ programmer would be required to write a large number of simple mapping routines to jacket the stubs and map from the network to the local object representation. This document describes a set of extensions to the DCE RPC IDL stub compiler attribute configuration file (ACF) to support generation of C++ bindings to IDL interfaces. This support enables the C++ programmer to create and utilize objects in DCE applications in a natural manner utilizing existing IDL interface definitions and without modification to the existing IDL network protocol.

1.2. Network Interfaces

The DCE model of accessing remote services is via the network interface of the service. The network interface of a service is defined by an IDL definition that exactly describes the signatures of the operations provided by the service. An IDL interface definition is, in an abstract sense, equivalent to a C++ class. It describes an entity that provides a set of operations and (potentially) some encapsulated data on which those operations act. In order for C++ applications to make use of this equivalence, it must be mapped into a C++ class declaration. This document describes the necessary support to enable the C++ programmer to control the mapping of an IDL interface definition into a C++ class declaration. Conceptually, the network interface of a class can also be viewed as a refinement of the existing three access levels provided by C++ (public, private, and protected). The network interface allows remote access to the public member functions of the class. The primary restriction imposed by the network access level is that direct access to member fields is prohibited.

1.3. Overview

This section provides a high level description of functionality necessary to support using RPC to build distributed applications using C++. Trying to make C++ itself distributed is a tremendously complicated task beyond our abilities. Specifically, we would have to modify the C++ compiler on each platform to support our definition of distributed objects. Obviously, this is not acceptable. Thus the scope of this effort has been limited to allowing location-independent, and transparent access to C++ objects (either local or remote). This is accomplished by a layered approach. RPC limits its interaction with the C++ compiler to only generating standard C++ code. This does limit the capability somewhat, while still providing a valuable paradigm for building distributed applications. Some of the areas we intend to address (in order of decreasing priority) are:

(a) Generate class definitions that provide location-independent invocation of the public methods of the class.

(b) Provide a mechanism for creating/deleting objects.

(c) Provide a mechanism for representing/finding long-lived objects.

(d) Enable object references to be passed as parameters.

(e) Provide a mechanism for moving an object between address spaces.

The notation used for describing the client/server communication protocol for DCE RPC is the Interface Definition Language (IDL). The IDL language consists primarily of the declarative statements of ANSI C (constants, typedefs, and function prototypes) which is enhanced by attributes to provide the extended information necessary to allow distribution. The goal of C++ support is to allow the mapping of constructs described in an interface definition to be mapped into C++ in a natural way. It is an explicit goal not to extend the interface definition

language. By retaining the current language notation, we enable interoperability between client/server pairs regardless of their implementation language.

2. DESCRIPTION OF CAPABILITIES

This section describes the capabilities, in detail, which this proposal claims to support. These descriptions are intentionally abstract to defer discussion of implementation issues described in later sections. Hopefully, this will aid in the analysis of the value of these features. The descriptions of these capabilities, given in the following list, is immediately followed by an illustrative example called `Memo`, which should be referenced while reading this list.

(a) Given the signatures of the set of public member functions (the `Memo` IDL interface), a C++ class will be generated which enables remote access to the class.

(i) The generated C++ class `Memo` is placed into a header file that the application then includes.

(ii) In addition to a header file, RPC stub files are generated that allow transparent access to instances of the class, if they are remote.

(iii) A client may utilize the `Memo` class via the RPC stubs without having a local implemention of the class (`SimpleMemo`) linked into the application.

(iv) The application-provided implementation of the class `SimpleMemo` must be derived from the generated class and be available on some server.

(v) The application-provided implementation of the class may vary from process to process.

(b) Location-independent invocation of the public member functions of an object whether it exists locally or remotely on a server.

(i) Given the following code fragment, it is determined at runtime whether or not `x` is local or remote and requests are processed accordingly. Remoteness is visible only by the increased potential for failure of the `update()` function when it is executed remotely.

```
Memo *x = GetfromInbox();
cout << x->read();
```

(c) Objects may be created via standard C++ syntax.

(i) Local instances of the class may be instantiated directly using the name of the class implementation.

```
Memo *x = new SimpleMemo;
Memo y;
```

(ii) If a constructor function is exported as one of the public member functions of the generated class and no local implementation is provided then remote instances of the class are instantiated using the remote constructor.

```
Memo *x = new Memo("Memo Value");
Memo y;
```

(d) Objects may be named and accessed by CDS name.

```
Memo *x = Memo::RPC_lookup("/.:/messages/my_messages");
cout << x->read();
```

(e) Servers register named objects by invoking the `RPC_object_register()` member function.

```
Memo *x = new SimpleMemo;
x->RPC_object_register(x, "/.:/messages/my_messages");
```

(f) Directed movement of mobile objects is requested via an invocation of the `RPC_migrate()` member function, specifying the RPC binding of the desired destination.

```
Memo x = GetfromInBox();
cout << x->read();
x->RPC_request_migrate(destination);
cout << x->read(); // Function on object at destination.
```

2.1. Memo Example: Network Interface

```
[uuid(70ff8220-6e1a-11cc-89ee-08002b2a1bca)] interface Memo {
void write(
    [in]         handle_t h,
    [in, string] char     *text);

[string, ptr] char *read(
    [in]         handle_t h);
}
```

2.2. Memo Example: Generated Network Class

```
class Memo : public RPC_object {
public:
    void write(char *);
    char *read();
};
```

2.3. SimpleMemo Example: A User Implementation of Memo

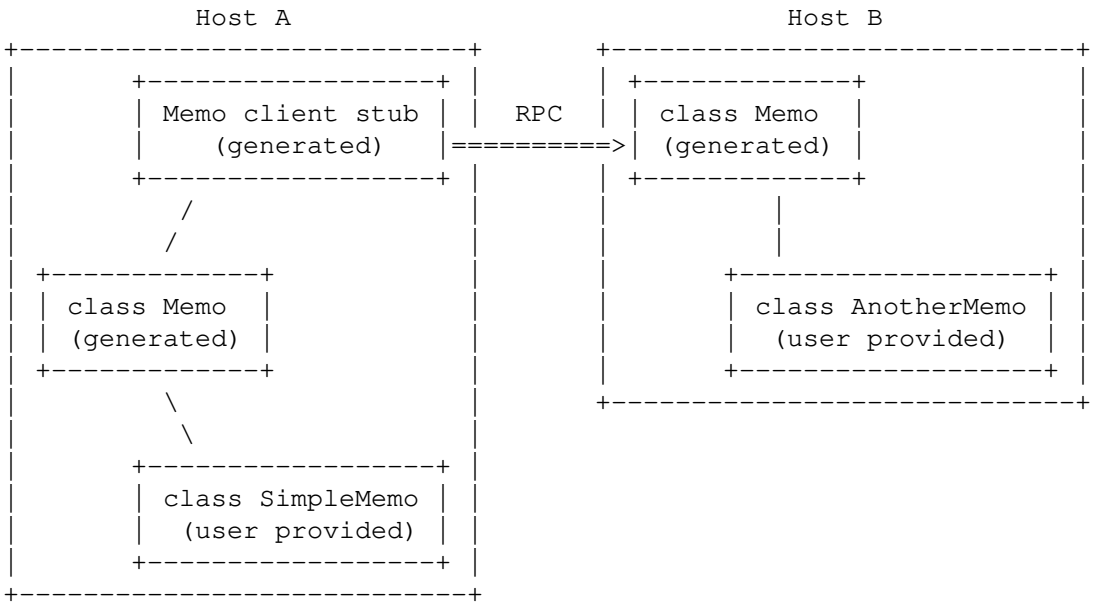
```
class SimpleMemo : public Memo {
private:
    char text[1000];

public:
    SimpleMemo() {text[0] = '\0';};

    void write(char *new_text) {
        strcopy(text,new_text);
    }

    char *read() {
        return text;
    }
};
```

2.4. Memo Example: Diagram of Components



3. THE RPC OBJECT MODEL

In the RPC object model, objects exist (are directly accessible) only on a single server (at a time). Although clients may access remote objects (via an RPC) as if they were local, they are, in actuality, working with object references. An object reference is a type of surrogate object (also known as a proxy object) which exists in the client address space. It is an instance of a C++ class generated by the IDL stub compiler to describe the public interface of the remote class (i.e., the public member functions). It provides transparent forwarding of member function invocations to the remote object for which it is a surrogate. If a client wishes to pass a local object to a server as a parameter, only a reference to the object is passed, and therefore the object must have a network interface.

The RPC mechanism provided to identify an object is the object UUID. This generated identifier uniquely identifies an object from all others across the entire network. When combined with a binding handle, the RPC runtime can transparently route a method invocation to the correct object on a remote server. Thus the network representation of an RPC object reference is simply a UUID combined with a binding handle. This identifier is generated automatically for all network objects. Additionally, RPC objects may be associated with entries in the DCE name service which provides symbolic access to an object by name. This supports finding long-lived objects that may exist beyond the lifetime of any particular server process (e.g., a printer named `/.:/printers/ps_5`).

4. UTILIZING RPC WITH C++

This section describes the application interface to RPC objects, and explains how a C++ programmer can make use of them. The following are the conceptual steps needed to utilize RPC from C++:

(a) Obtain or create an interface definition using IDL.

(b) Customize the mapping of the IDL operations via attributes in the ACF file.

(c) Build the client:

(i) Invoke the IDL compiler to generate a header file containing the class declaration, and client stubs which invoke the remote operations.

(ii) Write the client code that utilizes the class.

(iii) Compile the client application code which utilizes the class.

(iv) Link the client application code, and client stubs to form the complete application.

(d) Build the server:

(i) Invoke the IDL compiler to generate a header file containing the types and constants of the interface, and server stubs which access the remote operations. In the server the class definition is provided by the application.

(ii) Provide a class implementation that provides the entire interface defined in the IDL definition.

(iii) Compile the server application code which implements the class, and the server initialization code with performs the necessary RPC setup and optional object registration for named objects.

(iv) Link the server application code, and server stubs to form the complete application.

The only significant differences between using RPC from C and C++ is that in step (b) there are additional attributes which control the mapping into C++ member functions, and there are potentially additional steps in the server initialization in step (d) to support named objects.

4.1. Modeling an IDL Interface as a C++ Class: interface-name and implementation-name

An IDL interface is, in an abstract sense, equivalent to a C++ class. It describes an entity that provides a set of operations and (potentially) some encapsulated data on which those operations act. In order for a C++ application to make use of this equivalence, it must be somehow mapped into a C++ class definition. The IDL stub compiler will be extended to support the `-lang cxx` option. When present, this requests that the interface be generated as a C++ class.

In actuality, each IDL interface is represented by a C++ class which enables location independent invocation of object methods whether the object is local or remote. By default, the name of this class is taken from the interface name. It is derived from an RPC-provided class declaration which encapsulates any necessary RPC support. Each operation in the interface is present in the class declaration as a virtual public member function. This generated class encapsulates all of the public behavior of the IDL interface. Since there is no single mapping of an interface definition into a C++ class definition, the mapping can be controlled via ACF attributes.

Thus the following set of classes are involved for each IDL interface which utilize distributed RPC objects:

(a) IDL-generated *`interface-name`* class:

```
class interface-name : public RPC_object
```

This class represents the behavior of the IDL interface definition. It provides the surrogate (proxy) object when the object is not local. Method invocations are transparently forwarded to the actual remote object instance. The application implementation class *`implementation-name`* is derived from this class.

(b) User-provided *`implementation-name`* class:

```
class implementation-name : public interface-name
```

This is the only actual implementation of the class. It provides the name used when the application wishes to create a new local object instance. It may contain any desired data and/or member functions necessary to implement the desired behaviors.

4.2. Identification of RPC Objects Via the ACF

Because C++ is a superset of C, many valuable C++ concepts cannot be expressed in the C-based IDL. The following are the capabilities which we want to support (in the ACF), but are not currently expressible in IDL:

(a) Object references.

(b) Constructors and destructors.

(c) Inheritance.

(d) Polymorphism.

The following set of ACF attributes control object interpretation and naming when mapping the interface into a C++ class definition:

(a) `cxx_name(`*`name`*`)` attribute.

Usage: type, parameter, or interface.

Description: On a parameter or type, this attribute specifies the class name used when representing the specified type or parameter. On an interface, it allows specification of the name of the generated class (the default is *`interface-name`*). On an operation, it provides the name used when declaring the operation as a class member function. Specifying *`interface-name`* indicates the operation is a constructor. Specifying `~`*`interface-name`* indicates that the operation is a destructor.

(b) `cxx_static` attribute.

Usage: operation.

Description: This attribute identifies the operation as a static member function; thus no `this` pointer is required. The resulting operation in the generated class is `static` and will utilize the standard C rules for specification of the server binding handle.

(c) `cxx_inherit(`*`name`*`)` attribute.

Usage: interface.

Description: This attribute identifies another network interface from which the generated class for this interface should be derived. This attribute may be repeated for each parent interface.

(d) `cxx_implementation(`*`implementation-name`*`)` attribute.

Usage: interface.

Description: This attribute identifies the name of the local class that implements the interface. If omitted, only remote instances of the class are supported.

As in C++, the object represented by `this` is normally implicit in all method invocations. This is due to the implementation technique of specifying object UUIDs on the binding handle. Thus the identification of the `this` pointer is an implicit input in all methods. Object parameters are specified in the interface definition as type `RPC_object`. The IDL definition of `RPC_object` is provided in file `dce/cxxbase.idl` which is accessed via the IDL `import` statement if this type definition is needed. In order to associate this `RPC_object` with more than one class within an interface, an intermediate type (e.g., `my_network_object`) should be introduced and be modified by the `[cxx_name(`*`name`*`)]` ACF attribute to specify the actual class name of objects of this type. The determination of the passing mechanism (by value or by pointer) of the object is made by how the `RPC_object` type is passed in the IDL.

For those operations that do not inherently require an object, IDL also supports static member functions. This is specified by applying the `[cxx_static]` attribute to the operation name.

Constructor functions are another special type of member function supported by IDL. These are identified, like in C++, by utilizing the same name as the class definition (e.g., normally the interface name). As this name overloading is not possible in C, it must be specified via the `[cxx_name(`*`name`*`)]` attribute in the ACF. The IDL operation, however, for a constructor function must include a return value of type `RPC_object` so the identity of the newly created object can be conveyed back to the client.

As with local C++ class declarations, a C++ network interface may inherit behavior from other network interfaces. This is specified via the `[cxx_inherit(`*`name`*`)]` interface attribute. The name specified must be the class name of another IDL-generated network interface. If the IDL interface does not import the interface from which behavior is being inherited, then an include statement should be specified in the ACF file to assure the necessary declarations are available for the C++ declarations are available in the generated headerfiles.

The final attribute is `[cxx_implementation(`*`name`*`)]`. This attribute specifies the class name of the local implementation of the network interface in the address space, if available. The class specified by *`name`* should be derived from the generated *`interface-name`* class. This attribute is omitted for those network interfaces which

do not have an available implementation in the current address space. If a local implementation is available, specification of the `[cxx_implementation(`*name*`)]` attribute enables location-independent invocation. Thus when executing a member function on a local object instance, it will invoke the corresponding operation in *implementation-name* class, otherwise it will invoke the method via an RPC call.

4.3. Object Creation

For objects in which an implementation is available in the current process, the simplest method of object creation is to simply create an instance of the class *implementation-name*. As this is a standard C++ class, RPC is not even involved in local object creation. If the object is mobile across address spaces, then there is an additional requirement that it be allocated via the C++ `new` operator, and the resulting object may only be assigned to variables of type *interface-name*.

If there is not an implementation of the class *implementation-name* provided in the address space, it is still possible to create instances of the *interface-name* class on a remote server if the interface provides a constructor. Invocation of the constructor will cause an RPC to a server which provides an implementation of the class. On the server the actual constructor is invoked to create the object. On the client a surrogate is created to provide transparent access to the actual remote object. As mentioned above, if the object is mobile across address spaces, it must be allocated via the C++ `new` operator, and the resulting object must be assigned to a variable of type *interface-name*.

4.4. Object Access

The manner in which client applications access objects depends to some extent upon the type of processing being performed and whether or not the objects are mobile. These decisions are application driven, and thus the transparency of the resulting access is under control of the programmer.

Access to a remote object requires an instance of an RPC object reference. As described earlier, RPC object references are instantiated from a surrogate C++ class generated by the IDL compiler. An RPC object reference may be instantiated in the client process in the following ways:

(a) Utilizing the C++ `new` operation with a constructor function which originated in the IDL interface definition that was mapped into a constructor function via the ACF `[cxx_name(`*name*`)]` attribute.

(b) Returning an object reference as a parameter or function result from either a static member function or a member function of another RPC object reference.

(c) By providing an object name or binding handle when creating an *interface-name* object.

Using either method (a) or method (b) is syntactically identical to the instantiation of a local C++ object. If an object already exists (i.e., it is a named object), instantiation of an RPC object reference to access the object described in method (c) is somewhat less transparent.

4.5. Support for Named Objects

For the most part, the significant issues for supporting named objects deal with initialization and registration of the objects. Servers for named objects need to perform some of the following tasks:

(a) *Named object creation/registration.* — This is accomplished by first creating the C++ object and then invoking the `RPC_object_register()` member function with the desired name. If the named object does not yet exist in the name space then this invocation causes the object to be created, otherwise it simply registers the object with the RPC runtime for automatic lookup when requests are directed at the object. The `RPC_object_register()` member function has an optional second parameter which determines if the object UUID is registered with the endpoint mapper. If specified as `idl_true`, it enables multiple servers for the same interface, but a distinct set of named objects to be running on the same machine concurrently.

(b) *Named object lookup.* — For servers which do not elect to register all of their objects at server initialization time, they may choose to provide their own lookup function. By redefining the `RPC_object *RPC_get_object(uuid_t)` member function in their local class definition, the server may avoid registration of all named objects at initialization time, in favor of providing a different lookup mechanism that may actually create the in-memory object representation when a request is received on that object.

4.6. Matrix Example: Location Independent Objects

This client-side example illustrates the use of distributed objects to create and manipulate a matrix on a compute server elsewhere in the network. By virtue of automatic binding, and errors reported as exceptions, the *Matrix* object `m` is utilized in a completely natural manner. The example consists of the following files, whose contents are given below:

(a) `Matrix.idl`

The interface definition which provides the public declaration for the *Matrix* class.

(b) `Matrix.acf`

The ACF that enhances the description of the interface definition to support constructors/destructors and other C++ extensions not expressible in IDL.

(c) `Client.cxx`

The actual example program which creates and manipulates the remote *Matrix* objects.

4.6.1. Matrix.idl

```
[uuid(c664b260-a5db-11cb-832c-08002b2a1bca)] interface Matrix
{
    import "dce/cxxbase.idl";
    typedef RPC_object Matrix_object;

    /* Create a 2x2 matrix. */
    Matrix_object create_2x2(
        [in] double v11,
        [in] double v12,
        [in] double v21,
        [in] double v22);

    /* Set a new value in the matrix. */
    void set(
        [in] long row,
        [in] long col,
        [in] double value);

    /* Get a value from the matrix. */
    double get(
        [in] long row,
        [in] long col);

    /* Return the inversion of the specified matrix. */
    Matrix_object invert();

    /* Return the product. */
    Matrix_object multiply(
        [in] Matrix_object m1);

    /* Return the sum. */
    Matrix_object add(
        [in] Matrix_object m1);
```

```
    /* Delete a matrix. */
    void delete_matrix();
}
```

4.6.2. Matrix.acf

```
interface Matrix
{
    /* Map type Matrix_object into a C++ Matrix object. */
    typedef [cxx_name(Matrix)] Matrix_object;

    /* create_2x2() should be mapped as a constructor. */
    [cxx_name(Matrix)] create_2x2();

    /* delete_matrix() should be mapped as a destructor. */
    [cxx_name(~Matrix)] delete_matrix();
}
```

4.6.3. Client.cxx

```
#include "Matrix.h"

void main ()
{
    // Utilizing automatic binding, a server is found and a 2-
    // dimensional matrix is created.
    Matrix *m = new Matrix(1,0,0,1);

    // Invert (executes on the compute server).
    Matrix *inverted_m = m->invert();

    // Local Matrix object.
    Matrix *l = new matriximpl(2,7,3,2);

    // Add inverted_m to l and return sum (executes locally)
    // thus sum is a local object.
    Matrix *sum = l->add(inverted_m);

    // Perform the multiplication of sum and m (executes on server)
    // thus result is a reference to the remote object.
    Matrix *result = m->multiply(sum);

    // Display result.
    print(result);
}
```

5. THE RPC_object CLASS

The RPC_object class definition provides a framework for identifying and distributing objects. It is the base class for all IDL-generated surrogate classes. This is primarily a conceptual implementation based upon the types of support that we wish to make available. Whether or not all of these particular members will be actually implemented in the `RPC_object` class is TBD.

Currently, we do not foresee instances of `RPC_object` being created. We intend that this be a base type for the common support required by all RPC objects.

5.1. Details of the RPC_object Class

```
class RPC_object {
public:
    // RPC object, with default ("optional") UUID and name.
    RPC_object(
        uuid_t u = {0,0,0,0,0,{0,0,0,0,0,0}},
        char *object_name = NULL);

    // Return object UUID.
    uuid_t RPC_object_get_identity();

    // Set the object UUID.
    RPC_object_set_identity(uuid_t);

    // Request object to move.
    void RPC_request_migrate(handle_t h);

    // Default implementation of object migration causes nothing to
    // occur.  This operation is redefined in derived classes to
    // actually provide a meaningful action in repsonse to this
    // request.
    idl_boolean RPC_migrate(handle_t h) {return idl_false;};

    // Return the location of the object, or null if it is local.
    // This method is redefined by subclasses that enable remote
    // access to the object to return non-null values.
    handle_t RPC_object_get_location();

    // Register the object in the name service with the specified
    // name.  This enables access to the object from remote hosts by
    // specifying only the name.
    void RPC_object_register(void *, char *);

    // Return the address of the actual object (used from the server
    // stub).  This function should be provided by the server if
    // other than the default lookup method is to be utilized.
    static void *RPC_object_lookup(uuid_t);

    // ...
};
```

6. THE interface-name CLASS

As outlined above, this class defines the publicly accessible interface of the class (i.e., the public member functions). The resulting declaration is provided in the generated header file. The name of this generated class is that supplied via ACF attribute `cxx_name` on the interface. If the `cxx_name` attribute is not specified the class is named *`interface-name`*.

The generated class is derived from the `RPC_object` class, and has a public, virtual member function for each operation in the IDL interface. The default implementation of these member functions provide access to the remote implementations of the class. If the user provides an implementation of the class described by the IDL file, it would be derived from the *`interface-name`* class and identified via the ACF attribute `[cxx_implementation(`*`implementation-name`*`)]`. The presence of the `cxx_implementation` attribute causes the generated *`interface-name`* class to contain a pointer to an *`implementation-name`* object. If the object is local, then invocations of *`interface-name`* member functions are transparently

forwarded to the local instance of *implementation-name* otherwise, the invocation is forwarded via an RPC to remote instance.

Another attribute of the *interface-name* class is that it may provide serialized access to the actual user-provided class. This allows for the potential migration of objects between hosts. This would consist primarily of tracking access to the object so that the RPC runtime can determine at which instances the object is able to be moved.

7. REPRESENTATION OF BINDING INFORMATION

Another issue for client-side object representation is that of determining the RPC binding to the server. An RPC binding represents the location of the RPC server. Binding handles (which in IDL terms is supplied per-operation) is not particularly useful when attempting to act upon an object. Presumably, the program will be acting upon an object which exists on a remote server. Therefore the binding information is inherent in the object. That being the case, binding (other than that provided implicitly by the existence of the object) is only relevant when no object is specified. Thus only the following actions require binding information:

(a) Static function members.

(b) Constructors.

(c) Object lookup routines.

All other operations have binding implicitly supplied by the existence of the object. Binding handles (either implicit, automatic, or explicit in the signature) will be visible (for explicit) or accessed (for automatic or implicit) by the member function only in the following cases:

(a) An operation that is mapped into a constructor via the ACF.

(b) An operation that is mapped into a static member function.

In all other cases, the binding information is implicitly contained in the *interface-name* class. Another binding issue is that of accessing an object by name. This is done by supplying the name when creating the *interface-name* class for an object, or by invoking the static member function `RPC_object_lookup` supplied in the *interface-name* class. The object name is converted to a binding handle via the RPC NSI import operation.

8. NAMED OBJECT REGISTRATION AND ACCESS

The primary issue with respect to named objects is how they are registered. This consists of the following abstract steps:

(a) If not already contained by the object, a UUID is generated and associated with the object.

(b) An entry in the DCE namespace is created for this object name, and the object UUID is exported into it.

(c) The set of servers that support the object must be, somehow, associated with the object name entry in the namespace.

(d) If multiple servers of the same interface are available on the same node (this is determined by an optional argument to `RPC_object_register()`), then the server must also register each of the object UUIDs that it supports with the endpoint mapper.

(e) The object instances must be registered within the server such that requests to specific objects can be mapped into member function invocations.

Currently, all of these issues are encompassed with the single `RPC_object_register()` member function provided in the `RPC_object` class.

9. OBJECT MIGRATION BETWEEN HOSTS

Network objects typically encapsulate data of interest to many clients in the network. RPC provides significant value in that the data now becomes accessible, however there is often a significant performance penalty when

accessing a remote object as compared to the cost of a local object. As such this design proposes to support a framework for assisting in the support of mobile objects that can migrate from one host to another. When dealing with mobile objects, it is required that they be accessed only via an *interface-name* object. This enables location-independent invocation as well as ability to track and control access to the object.

The API to request an object change its location is the `RPC_request_migrate()` routine. It accepts a single binding handle parameter which identifies the desired destination. Object migration is always under control of the object instance itself, thus calls to this routine are effective only if the object involved is local. If a particular class wishes to support client-directed migration, then it must export a member function which accepts requests for migration and mediates them. As the migration is self-directed, intelligent objects can detect patterns of access (if so inclined) and dynamically adjust their location accordingly. The general scenario for movement of objects is as follows:

(a) The `RPC_request_migrate()` member function is invoked with a binding handle indicating the desire destination. As noted above this must occur on the host on which the object actually resides.

(b) If the object is currently being accessed the request is noted for future processing, and the `RPC_request_migrate()` returns to its caller.

(c) When the object becomes available, it is locked for exclusive access until the migration is complete. This is accomplished by locking primitives included in the *interface-name* class. The destination binding is then fully resolved to identify the particular server instance to which the transfer will occur.

(d) The user-supplied *implementation-name*`::RPC_migrate()` routine on the local host is then invoked with the fully resolved binding handle. The object must bring itself into a fully quiescent state and then copy that state to the remote host. This can be accomplished by whatever means desired such as: invoking an operation in the interface such as a constructor, or static member function which will reinstantiate the object on the remote host using the specified state information. The signature of the support operation that actually transfers the data is necessarily defined by the application programmer as it needs to describe a network encoding which captures all information required to recreate the object with an equivalent state at the remote host.

(e) After creating the remote instance of the object, the support operation must set the object identity to that of the source object (via `RPC_object_set_identity()`). This implies that he object identity is somehow communicated to the remote host (typically by including the object's UUID (obtained by `RPC_object_get_identity()`) in the operation signature).

(f) Upon returning to the RPC runtime support from the *implementation-name*`::RPC_migrate()`, the local instance of the *implementation-name* object is destroyed with the C++ `delete` operation, and the *interface-name* surrogate object is modified to forward requests to the new location as specified by the binding handle.

(g) The *interface-name* is then unlocked, and forwards all accesses to the object to the instance at the new location.

9.1. Migration Support Examples

All network classes inherit the `RPC_request_migrate()` and `RPC_migrate()` methods from the `RPC_object` class. The system-provided implementations of these routines cause no actual migration. True object migration requires support in the user-provided *implementation-name* class. This is done by replacing the default implementation of the `RPC_migrate()` member function with an implementation that actually transfers the object to another host. The example below shows such a routine, for an object with only two private member fields that constitute the object's state. First is a structure called `data_context`, and second is a string `last_name_used`. When processing a migrate request, the RPC runtime invokes this function. If it successfully moves the object to the remote host (in this case via the static member function `my_migrate_support()`) it returns `idl_true`.

```
/* Sample user-provided RPC_migrate() routine. */
idl_boolean implementation-name::RPC_migrate(handle_t h)
{
    uuid_t object_identity = this->RPC_object_get_identity();
    return interface-name::my_migrate_support(
        h, object_identity, data_context, last_used_name);
}
```

The corresponding implementation of the `my_migrate_support()` routine can be seen in the following example. The routine creates a new instance of the *implementation-name* class via a constructor (not necessarily available in the network interface). It then uses the RPC support routine to replicate the original identity of the object. Finally, it returns an indicator of success which is reported back the `RPC_migrate()` function shown above.

```
/* Sample user-provided migrate support routine. */
static int implementation-name::my_migrate_support(
    handle_t h, uuid_t object_identity, my_state_t s,
    char *last_name_used)
{
    interface-name *new_instance =
       new implementation-name(s, last_name_used);
    new_instance->RPC_object_set_identity(object_identity);
    return idl_true;
}
```

9.2. Location-Independent Support for RPC_request_migrate()

If an application chooses to allow client-directed object migration in a completely transparent manner, it needs to support remote access to the `RPC_request_migrate()` routine. Without supporting client-directed object migration, performing a migrate operation on a surrogate object has no effect. Client support for controlling object migration is enabled by providing an operation in the IDL interface that transparently forwards the request to the actual object instance for mitigation. This operation has the following signature.

```
void my_migrate_routine([in] handle_t object);
```

For the location-independent migration support to be effective, the application provides a migration routine in the IDL interface as described above, and then replaces the default implementation of the `RPC_request_migrate()` operation by applying the `[cxx_name(RPC_request_migrate)]` attribute to the operation in the ACF file. Thus if invoked for a surrogate, the `RPC_request_migrate()` operation is forwarded transparently to the remote object implementation for processing. As noted above, the object instance can then choose to honor or ignore client migration requests as it deems proper for the most efficient optimization.

10. OPEN ISSUES

We have not yet addressed the following open issues:

(a) Exceptions.

(b) Garbage collection for mobile objects.

(c) Support for pass-by-copy C++ objects in addition to the current support for object references.

(d) Supporting IDL `[string]` types with C++ `string` class.

(e) Syntactic issues (e.g., capitalization style).

11. ACKNOWLEDGEMENTS

The original work represented by this document, as well as the first version of the document itself, is due to Jerry Harrow (DEC).

AUTHOR'S ADDRESS

Rico Annicchiarico
Digital Equipment Corporation
110 Spit Brook Road ZK2-1/Q18
Nashua, NH 03062-2698
USA

Internet email: Rico@clt.enet.dec.com
Telephone: +1-603-881-2685