OSF DCE SIG
Request For Comments: 48.1

R. Annicchiarico (DEC)
J. Harrow (DEC)
R. Viveney (DEC)
April 1994

C++ SUPPORT IN DCE RPC — FUNCTIONAL OVERVIEW

1. INTRODUCTION

1.1. Existing DCE Support

The OSF Distributed Computing 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 Interface Definition Language (IDL) and run time library 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 with 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 and used as operation return values.

(e) Enable native C++ objects to be passed as parameters.

(f) Provide support for interface inheritance.

(g) Provide support for multiple object implementations of a common interface.

(h) Allow co-existence of local and remote objects derived from a common interface in the same application.

(i) Provide a mechanism to do memory reclamation in object servers.

(j) Provide support for the C++ reference operator in remote procedures.

(k) Provide support for C++ exception handling between address spaces.

The notation used for describing the client/server communication protocol for DCE RPC is the Interface Definition Language. 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. Extensions to the existing Interface Definition Language are described in this paper along with implementation details. By retaining the current language notation, we enable interoperability between client/server pairs regardless of their implementation language.

2. DESCRIPTION OF CAPABILITIES

This section presents an illustrative example called `Memo` followed by a detailed description of the capabilities which this proposal claims to support. The example should be referenced while reading the capability descriptions. 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.

2.1. Memo Example: Network Interface

```
[uuid(70ff8220-6e1a-11cc-89ee-08002b2a1bca)]
interface Memo {
        typedef [string, ptr] char * net_string;

        Memo *newMemo([in, string] char *title);

        void write( [in, string] char *text );

        [string, ptr] char *read();

        boolean spellCheck( [in] Memo *m);

        void append( [in, string] net_string new_text);
}
```

2.2. Memo Example: Attribute Configuration File

```
interface Memo {
        include "cxx/String";

        typedef [represent_as(String)] net_string;

        [cxx_new(AnotherMemo)] newMemo;
}
```

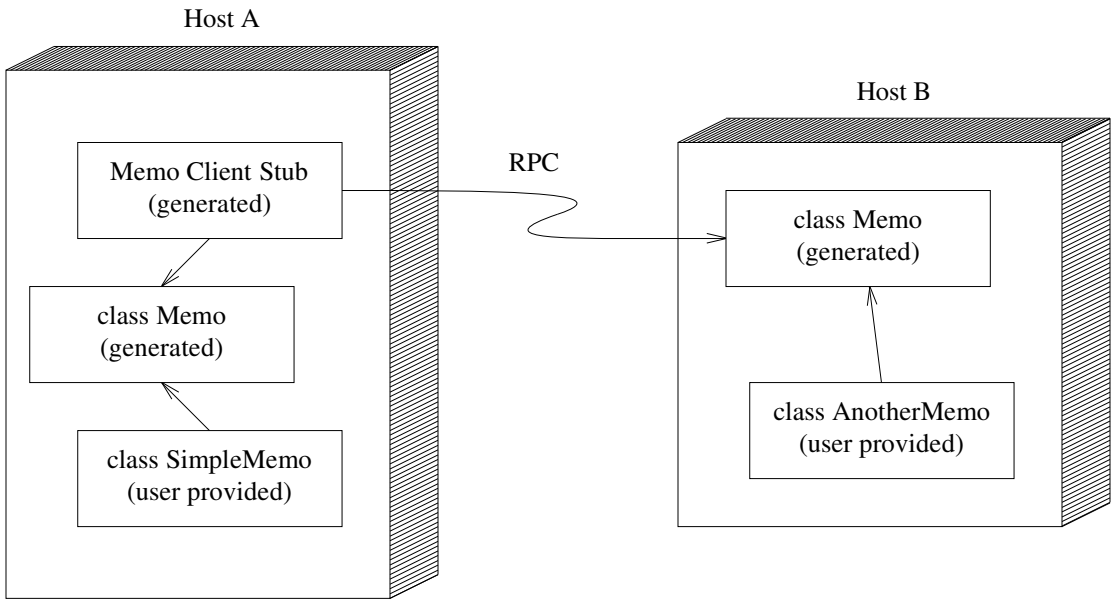
2.3. Memo Example: Generated Network Class

```
class Memo : public virtual rpc_object_reference {
public:
    static Memo *newMemo(char *title);
    void write(char *) = 0;
    char *read()       = 0;
    idl_boolean spellCheck(Memo *m) = 0;
    void append(String new_text) = 0;
};
```

2.4. SimpleMemo Example: A User Implementation of Memo

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

public:
    SimpleMemo() {text[0] = '\0';};
    SimpleMemo(char *title) {
        strcpy(text, title);
    }

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

    char *read() {
        return text;
    }

    idl_boolean spellCheck(Memo *m) {
        return idl_true;
    }

    void append(String new_text) {
        strcat(text, new_text);
    }
};
```

2.5. Memo Example: Diagram of Components



2.6. Generated C++ Class

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.

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

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

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

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

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

2.7. Location Independent Invocation

Public member functions of an interface are accessed the same, regardless of whether the object is implemented locally or remotely.

Given the following code fragment, it is determined at run time 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 `read()` function when it is executed remotely.

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

2.8. Object Creation

Objects may be created via standard C++ syntax. Local instances of the class may be instantiated directly using the name of the class implementation.

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

If a creator function is exported as one of the public member functions of the generated class then remote instances of the class are instantiated using the creator function and a remote constructor.

```
Memo *x = Memo::newMemo("Memo Title");
```

2.9. Named Objects

IDL will support named objects. A named object is an object that is registered with a name service. Objects may be named and accessed by CDS name.

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

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

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

2.10. Objects as Parameters

Local and remote objects may be passed as parameters.

Given the following code fragment, the local implementation `x` can be accessed by the remote object `y`.

```
Memo *x = new SimpleMemo;
Memo *y = Memo::newMemo("Memo Title");
y->spellCheck(x);
```

Given the following code fragment, the remote object `y` can be accessed by the local implementation `x`.

```
Memo *x = new SimpleMemo;
Memo *y = Memo::newMemo("Memo Title");
idl_boolean ok = x->spellCheck(y);
```

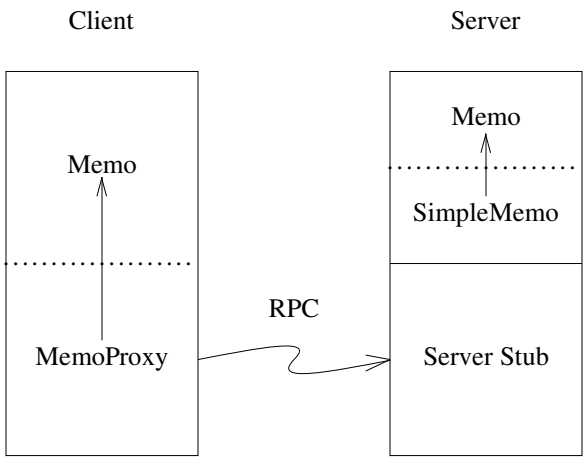
Native C++ objects may be passed as parameters.

```
String s("Hello World");
Memo *y = Memo::newMemo("Memo Title");
y->append(s);
```

3. THE RPC OBJECT MODEL

In the RPC object model, objects exist and 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 proxy object (also known as a surrogate 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 proxy. 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 run time 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`). The diagram below outlines the client/server interaction with remote objects involved.



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 object creator operations in the IDL file with attributes in the ACF file. (This step is only necessary for dynamically created remote objects.)

(c) Build the client:

(i) Invoke the IDL compiler with a `-lang cxx` option to generate a C++ header file containing the class declaration and client stub which invokes the remote operations.

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

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

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

(d) Build the server:

(i) Invoke the IDL compiler with a `-lang cxx` option to generate a C++ header file containing the types and constants of the interface, and server C++ stub which accesses the remote operations. In the server, the class definition is provided by the application.

(ii) Create 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 which 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 option directs the compiler to generate the interface as a C++ class.

In actuality, each IDL interface is represented by an abstract C++ class which enables location independent invocation of object methods via polymorphism whether the object is local or remote. 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 pure virtual public member function. This generated class encapsulates all of the public behavior of the IDL interface. A proxy class is also generated by the IDL compiler that is derived from the abstract base class. The operations in the proxy class carry out the RPC calls to the server.

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

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

```
class interface-name : public virtual
                                    rpc_object_reference
```

This abstract class represents the behavior of the IDL interface definition.

(b) IDL-generated proxy class:

```
class interface-nameProxy : public virtual
                                        interface-name
```

This class provides the proxy object when the object is not local. Method invocations are transparently forwarded to the actual remote object instance.

(c) User-provided *implementation-name* class:

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

This is the actual implementation of the interface. 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. It must, however, provide an implementation function for each of the member functions in the abstract base class from which it is derived.

4.2. Identification of RPC Object Creators Via the ACF

The following ACF attribute will be available to identify operations that are used to create new remote objects.

(a) `cxx_new ( name )` attribute.

Usage: operation

This attribute identifies the operation as a remote object creator. The remote creator function creates an object of type *name* and returns an object reference. The *name* class must be derived from the *interface-name* class. The operation must be defined as returning a pointer to the *interface-name* class. The generated operation is accessed as a normal C++ static member function in the *interface-name* class.

4.3. Object Reference

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 uuid on the binding handle. Thus, the identification of the `this` pointer is an implicit input in all non-static methods.

Object parameters are specified in the interface definition as type *interface-name* `*`. The wire representation of an *interface-name* `*` parameter is an `ObjectRef`, which is encapsulated in the `rpc_object_reference` base class inherited by the *interface-name* class. The IDL compiler will generate code to marshal an *interface-name* `*` parameter into an `ObjectRef` and to unmarshal an `ObjectRef` into an *interface-name* `*` type.

An `rpc_object_reference` class has the following public interface.

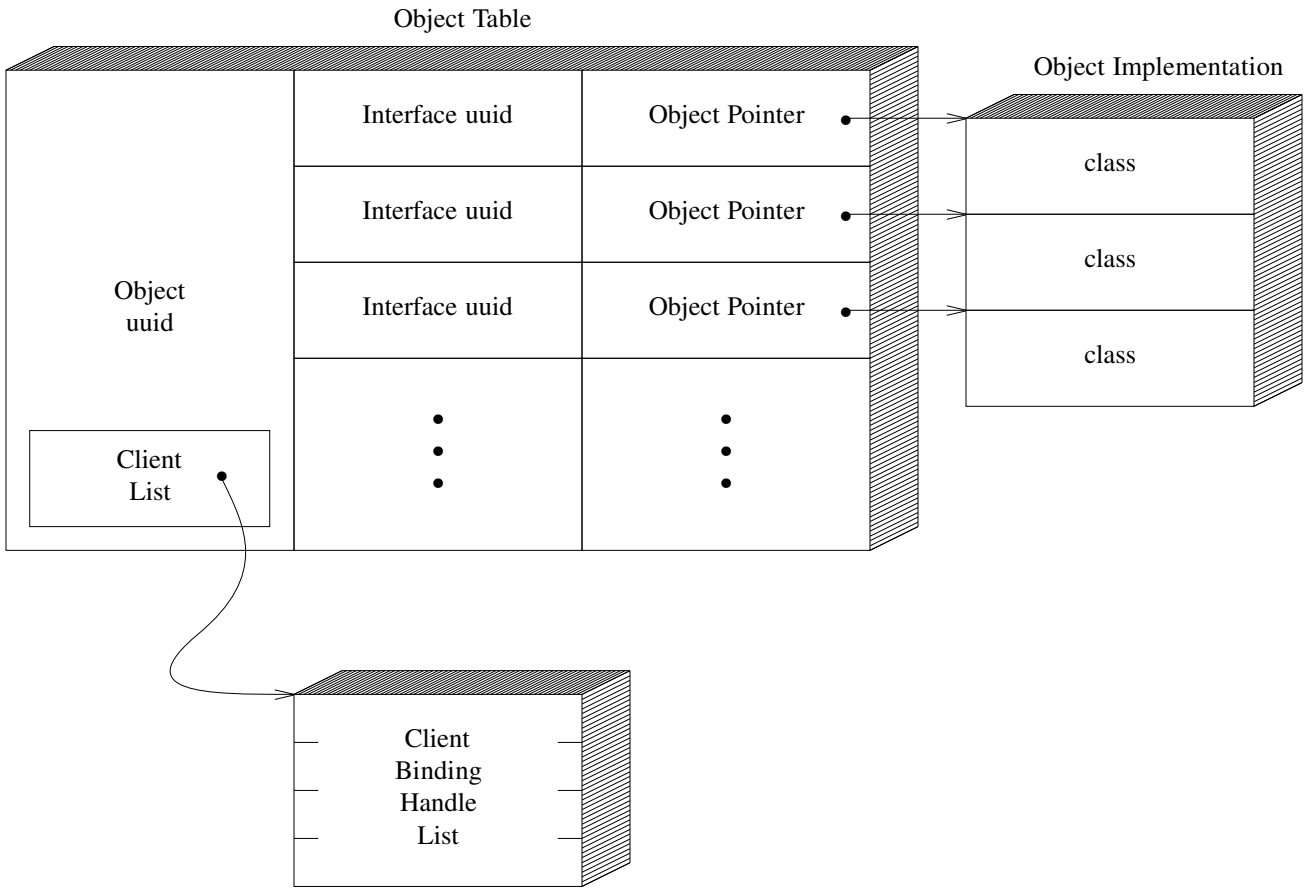
```
class rpc_object_reference {
public:
    // For registering objects in the name space
    void register_named_object(void *objectPtr,
                               char *objectName,
                               idl_boolean register=idl_true);

    // For tracking object lifetimes
    void add_ref();
    void release();
}
```

4.4. Object Table

Objects created by a server are uniquely identified by an object uuid associated with a binding handle. A server that receives an operation request on an implied `this` pointer must translate it to an actual object pointer. An Object Table will be maintained in the server's address space which maps an object uuid to the real object address. The Object Table is implemented as a C++ class containing a table of object uuids, interface uuids, and object addresses. The interface uuid is used to uniquely identify the object reference in a derivation tree of interfaces. An object that is derived from a single interface does not use the interface uuid in the Object Table. An Object Table entry also contains a list of clients associated with the object. An Object Table class has the following interface as seen by the proxy class, and is presented here solely as an aid to the understanding of the underlying RPC object model.

```
class RpcObjectTable {
public:
    // Constructor builds an Object Table with a default size
    RpcObjectTable(idl_long = DefaultTableSize);

    // Destructor
    virtual ~RpcObjectTable();

    // Add a table entry
    void add(uuid_t *objectId, void * objectPtr,
                               uuid_t *interfaceId = 0);

    // Remove a table entry
    void remove(uuid_t *objectId);

    // Returns the object instance pointer
    void *getPtr(uuid_t *objectId, uuid_t *interfaceId = 0);
}
```

4.5. Object Table Layout



4.6. Interface Inheritance

As with local C++ class declarations, a C++ network interface may inherit behavior from other network interfaces. This is specified via the new IDL language operator `:`. The IDL grammar will be modified to change an Interface Definition Header to:

```
<inheritance_spec> ::= ":" <identifier> ["," <identifier>] ...
<interface_header> ::= "interface" <identifier>
                           [ <inheritance_spec> ]
```

The identifier specified in an `inheritance_spec` must be the class name of another, previously declared IDL-generated network interface. If the IDL interface does not import or contain 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 generated header files.

4.7. Multiple Compilation Units

To allow a natural way for specifying multiple interfaces for an object and to make an interface hierarchy more readable, the IDL grammar should also be extended to allow multiple interface definitions in a file. A forward declaration syntax should be introduced to allow an interface to be declared and used before it is defined.

```
<forward_dcl> ::= "interface" <identifier>;
```

Currently, all types defined within an IDL file are generated as global typedefs. Types defined within an interface definition are also global to the file and may be referred to in subsequent interface definitions within the same file.

4.8. Object Creation

For applications where object implementations are 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* * from which it is derived.

If there is not an implementation of the class *implementation-name* provided in the address space, it is still possible to create instances of pointers to the *interface-name* class linked to an object on a remote server if the interface provides a creator function via the ACF file. Invocation of the creator function 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 and an `ObjectRef` is returned to the client. On the client a proxy is created using the `ObjectRef` to provide transparent access to the actual remote object and returned to the user.

4.9. 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 encapsulated in a C++ proxy class generated by the IDL compiler. An RPC object reference may be instantiated in the client process in the following ways:

(a) Utilizing the defined remote object creator operation as described previously.

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

(c) By calling one of the static bind operations generated into the proxy class by the IDL compiler.

(i) *interface-name* `* bind(const char * name)` uses the name parameter to search the name space for binding handles.

(ii) *`interface-name`* `* bind(const uuid_t *)` uses the uuid parameter to search the name space for binding handles. Searches are started at the default path specified by the RPC-specific environment variable `RPC_DEFAULT_ENTRY`.

(iii) *`interface-name`* `* bind(const handle_t)` uses the binding handle parameter as the object reference.

Each of the three bind operations is available as a static member function in the *`interface-name`* class.

4.10. Support for Named Objects

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

4.10.1. Named object creation/registration

This is accomplished by first creating the C++ object, assigning the newly created object to an interface pointer variable, and then invoking the `register_named_object()` member function with the desired name. If the named object does not yet exist in the name space, this invocation causes the name to be created. Otherwise, it simply registers the object with the RPC run time for automatic lookup when requests are directed at the object. The `register_named_object()` member function also has an optional 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.

4.10.2. Named object lookup

Servers may elect not to register all of their objects at server initialization time, but to provide their own lookup function. The `this` pointer will indicate that the object reference needs to be registered in the Object Table as there will be no entry for the specific object instance wanted.

An ACF attribute will be available for the purpose of identifying such lookup functions. The operation modified by the ACF attribute will be generated as a static member function in the server stub and will be typed checked as returning an `interface *` value.

(a) `cxx_lookup ( name )`

Usage: interface

This attribute identifies the operation name as a special lookup function. The lookup function allows a server to manage its own table of object pointers. The operation must return a pointer to the *interface-name* class. The generated operation is accessed as a normal C++ static member function in the *interface-name* class.

4.11. Reference Counting

Object references may be passed to other functions or duplicated freely by client applications. Similarly, object references that refer to the same object may be passed to different client applications by a server. In order to keep track of copies of object references, an Object Reference Counting paradigm must be followed.

The `rpc_object_reference` class provides for reference counting by encapsulating a reference count variable and a public interface to the variable. All RPC objects have access to this interface. When an object pointer is duplicated in a client application, it is the client programmer's responsibility to notify the object that it is being duplicated. This is done by invoking the `add_ref()` member function to increment the reference count for the object. When the extra reference to the object is no longer needed, the `release()` member function is invoked to decrement the reference count.

An object pointer returned by any RPC operation will initially include a non-zero reference count. Proxy reference counts are maintained locally, and do not cause any network traffic until the reference count reaches zero, at which time the client notifies the server that the object instance is no longer needed.

Servers also track object references. During marshaling, a server that is about to return a new object reference to a client invokes an Object Table member function with the client binding handle parameter which logs the client as a user of the object. When the client notifies the server that the object reference is no longer needed, the server disassociates the client from the object. An object with an empty client list can be freed from the server's address space and removed from the Object Table, provided the object is not a long-lived object.

4.11.1. Reference counting optimizations

Some optimizations are possible for the reference counting paradigm. A client that knows the scope of a duplicated object pointer need not call the `add_ref()` operation. For example, if an object reference is completely nested within a longer lived object reference to the same remote object, then the `add_ref()/release()` operations can be bypassed since the lifetime of the object is contained within the outer scope of the original object reference.

Likewise, if a duplicated object pointer's lifetime overlaps the original, then the copy can inherit ownership of the reference count on behalf of the original.

4.12. Local Objects

The IDL C++ extensions should not preclude a client application from creating its own object instantiations. If the user knows an implementation class of an interface it can be used in the normal way by calling the constructor function for the class. If the object is to be used as an RPC object, i.e., the user wants to pass the object to a remote procedure, the object pointer should be assigned to an interface pointer for the class. Then the interface pointer can be passed as a parameter to any supporting remote procedure without ramifications. The underlying mechanism to do this will essentially make the client a server of the object, logging the object instance in the Object Table, tracking clients of the object, and translating the object instance to an object reference wire representation that is passed to the remote procedure.

In the following example, a `SimpleMemo` implementation object is created locally, assigned to an interface pointer and passed to a remote procedure. The remote object references the local object.

```
Memo *mremote, *mlocal, *m;

mremote = Memo::bind("/.:/MemoObjects/myObject1");
mlocal = new SimpleMemo("some text");
mremote->append(mlocal);
```

4.13. 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 identifies the object creator operation in the IDL file.

(c) `Client.cxx`

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

4.13.1. Matrix.idl

```
[uuid(c664b260-a5db-11cb-832c-08002b2a1bca)]
interface Matrix
{
    /* Create a 2x2 matrix. */
    Matrix * new2x2(
                          [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 * invert();

    /* Return the product. */
    Matrix * multiply(
                      [in] Matrix * m);

    /* Return the sum. */
    Matrix * add(
                 [in] Matrix * m);
}
```

4.13.2. Matrix.acf

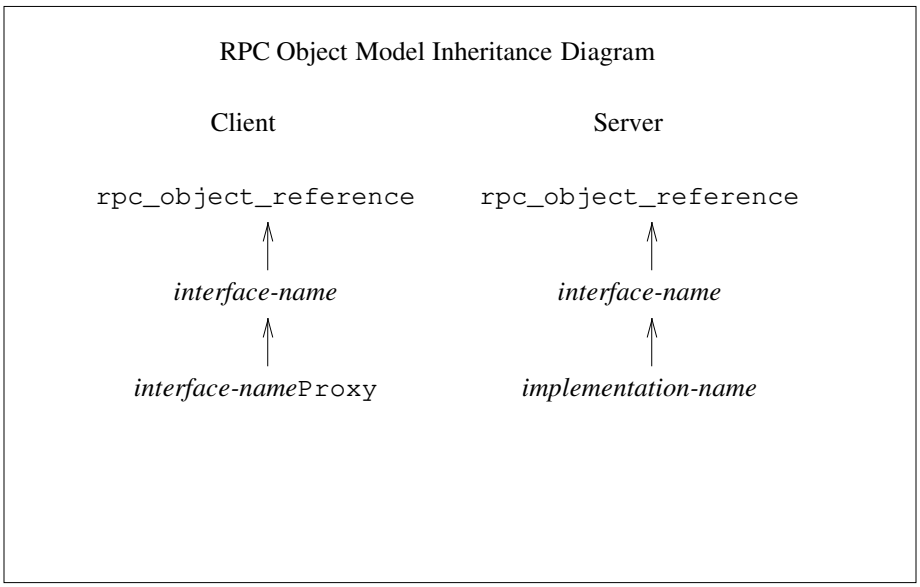
```
interface Matrix
{
    /* identify the remote object creator function */
    [cxx_new(MatrixImpl)] new2x2;
}
```

4.13.3. Client.cxx

```
#include "Matrix.h"

void main ()
{
    // Utilizing automatic binding, a server is found and
    // a 2-dimensional matrix is created.
    Matrix *m = Matrix::new2x2(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 remotely)
    // Thus result is a reference to the remote object.
    Matrix *result = m->multiply(sum);
}
```

5. IDL-GENERATED CLASS HIERARCHY

When the `-lang cxx` option is used, the IDL compiler will generate a C++ class hierarchy to support remote objects. The base class for this hierarchy, `rpc_object_reference`, includes any necessary functionality to provide this support. The *interface-name* abstract class provides the public interface as defined in the interface description along with object creator functions. The *interface-name*`Proxy` class supplants the RPC client stub to carry out remote operations. Users will provide their own object implementation class, which must be derived from the *interface-name* abstract class. The diagram below illustrates this class hierarchy, which is then followed by a more detailed description of the IDL generated classes.



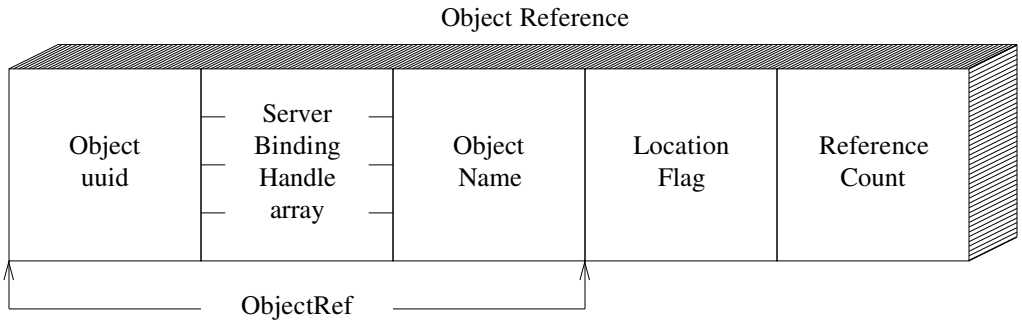
5.1. The rpc_object_reference Class

The `rpc_object_reference` class definition provides a framework for identifying, distributing, and tracking objects. It is the base class for the IDL-generated *interface-name* class. The interface to this class has already been presented. The details of the class described below are subject to change according to design decisions.

An object reference, implemented by an `ObjectRef` structure contained within the `rpc_object_reference` class, has already been defined to be minimally a binding handle and a uuid. The binding handle identifies the server providing the object implementation and the uuid identifies the specific object instance on the server. It is the `ObjectRef` structure that is passed across the wire as the object reference. This definition is expanded here to allow an `ObjectRef` to contain an array of binding handles. All transport protocols supported by the server will be stored in the binding handle array, allowing clients to choose which protocol to use. This is useful when one client passes an object reference to another client that uses a different protocol than the first to talk to the server.

To support named objects, an `ObjectRef` will also contain a name field. The name field identifies where in the name space to search for the actual binding handles for the object. Clients may use this name field to search the name space when none of the server binding handles in the array are sufficient. The server uses it to distinguish a named object from a dynamic object so that named objects are not inadvertently deleted by clients.

A location flag and reference count are also encapsulated in an `rpc_object_reference` class. The location flag identifies whether the object is implemented locally or remotely and the reference count is used to track object lifetimes.



5.2. The *interface-name* Class

As outlined previously, this abstract base 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 class is

interface-name.

The generated class is derived from the `rpc_object_reference` class, and has a public, pure virtual member function for each non-creator operation in the IDL interface. No constructor or destructor functions are allowed in the IDL definition file for the interface. Specifying such functions will cause a warning to be generated and the function to be ignored. The IDL compiler will generate its own destructor function for the class. No constructor function is generated since this is an abstract base class.

The presence of the `[cxx_new(`*implementation-name*`)]` *operation-name* attribute in the ACF file causes the implementation for the *operation-name* to be placed in the *interface-name* class as a public static member function.

For example, if the ACF file for the Matrix example contains

```
[cxx_new(MatrixImpl)] new2x2;
```

then the generated signature for this function in the *interface-name* class is

`static Matrix * new2x2(`*same parameter list*`);`

This member function is then used by client applications to instantiate remote objects of type `MatrixImpl` and return an interface pointer to the caller. The IDL compiler will do type checking to insure that the IDL definition for the member function returns an *interface-name* `*`.

5.3. The *interface-name*Proxy Class

A proxy class is derived from the abstract base class and is named *interface-name*`Proxy`. IDL provides an implementation in *interface-name*`Proxy` for each of the non-creator member functions defined in the *interface-name* class. These implementation functions provide access to remote instantiations of the class. The proxy class implementation will be responsible for translating between object interface pointers and object references.

While the client typically deals with *interface-name* type classes, it is the derived proxy class that actually represents the remote object. Polymorphism in the class hierarchy allows transparent access to the remote procedure calls in the client stub.

6. 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 are supplied per-operation) are 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 and encapsulated in the `rpc_object_reference` base class. Binding information is only specified in an operation when using one of the three *interface-name* `* bind()` static member functions or an object creator operation. (The latter case actually depends upon how the binding method attribute is specified in the IDL operation definition. A `auto_handle` attribute would cause the operation to not require an explicit binding handle to be supplied when invoking the operation.)

7. 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) An entry in the DCE name space is created for this object if needed.

(b) If not already contained by the object, a uuid is associated with the object.

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

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

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

All of these issues are encompassed within the single `register_named_object()` member function provided by the `rpc_object_reference` class.

8. MEMORY RECLAMATION

In a perfect world, client applications follow the Object Reference Counting paradigm and objects are created and freed only as needed. However, clients that fail to follow this paradigm or that crash may leave orphaned objects on some remote server. There needs to be some way to reclaim the memory occupied by orphaned objects.

To this end, the Object Table will contain a list of client binding handles. Clients who are active users of an object are stored here. This list is maintained by the RPC server stub routines to track client usage of objects. Before an object reference is passed to a client, the client is logged as a user of the object in the server's client binding handle list.

A client stub that receives an object reference from another client will notify the server of the object that it has a received the object reference. When clients adhering to the reference counting paradigm notify servers that the object reference is no longer needed, the client binding handle is removed from the server's client binding handle list.

Periodically, it will be necessary for object servers to process the Object Table and look for objects that have become orphaned. This is accomplished by a separate server thread that loops through the client binding handle list for each entry in the Object Table, pinging each client. Unanswered pings will cause the client binding handle to be removed from the list. The algorithm must also allow sufficient time for client proxies to be created for recently added object clients, and to handle temporary communication failures. An empty list indicates that there are no clients currently accessing the object, and the object can be removed from the server's address space. Named objects are treated slightly different in that they are allowed to have an empty client list.

9. C++ EXCEPTIONS

C++ programmers will obviously want to use the C++ exception mechanism for remote objects if their compiler supports it. The `-lang cxx` option to the IDL compiler will cause a C++ exception reporting mechanism to be generated in the RPC stubs.

Client stubs will continue to use the DCE exceptions. If a DCE fault is caught and C++ exceptions are supported, the catcher should then use it to raise a C++ exception to the client application.

Server stubs must be able to catch C++ exceptions also. A `try-catch` block needs to surround each manager code callout and caught exceptions need to be propagated back to the client as a special DCE exception which will then cause the client stub to raise a C++ exception. Again, if C++ exceptions are not supported, this code should be bypassed.

10. PASS-BY-COPY OF C++ OBJECTS

To pass a native C++ object as a parameter, the parameter attribute in the IDL file should have a `represent_as` attribute and the user will have to supply the `..._to_local()` and `..._from_local()` functions to translate the object into a wire representation that can be passed as a parameter. See the OSF DCE Application Development Guide for information on how to use this attribute.

11. THE C++ REFERENCE OPERATOR (&)

When a parameter in the IDL file is specified as being passed by reference by using the C++ reference operator (&), it will be transmitted as a `[ref]` pointer.

AUTHORS' ADDRESSES

Rico Annicchiarico
Digital Equipment Corporation
110 Spit Brook Road ZK3-2/Q18
Nashua, NH 03062-2698
USA

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

Jerry Harrow
Digital Equipment Corporation
110 Spit Brook Road ZK3-2/Q18
Nashua, NH 03062-2698
USA

Internet email: harrow@clt.enet.dec.com
Telephone: +1-603-881-2193

Bob Viveney
Digital Equipment Corporation
110 Spit Brook Road ZK3-2/Q18
Nashua, NH 03062-2698
USA

Internet email: viv@zko.dec.com
Telephone: +1-603-881-0362