Open Software Foundation
Request For Comments: 48.3
April 1996

R. Viveney (DEC)

# C++ SUPPORT IN DCE RPC IDL — FUNCTIONAL SPECIFICATION

## 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. This document specifies what is considered to be the essential enhancements to the DCE IDL compiler for effectively using C++ and objects in a DCE environment.

**Note that this document supersedes RFC 48.2 as the definition of IDL C++ support planned for OSF DCE 1.2.**

### 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.

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

## 2. TERMINOLOGY

(a) **Object** — A term used to describe the aggregation of data and functionality into a unit in a software process.

(b) **Distributed Object** — An object that spans more than one process.

(c) **DCE Remote Object** — An object that is not local to the process that is accessing it, but rather, has its implementation remoted within the DCE environment.

(d) **Object Interface** — An abstraction for accessing an object instance.

(e) **Object Reference Base Class** — A class definition that is the basis of all DCE remote objects. It encapsulates any information needed to carry out remoteness of objects.

(f) **Abstract Base Class** — A class generated by the IDL compiler directly from the interface definition for use within client and server applications.

(g) **Proxy Class** — A class generated by the IDL compiler directly from the interface definition for use by the underlying support for DCE remote objects. This class is the equivalent of the DCE client stub in procedural DCE programming in that it forwards method invocation to the remote object implementation.

(h) **Manager Class** — A class definition supplied by the user that implements all the operations defined by an IDL interface to support a DCE remote object.

## 3. TARGET

This technology is targeted for DCE customers who want to use C++ to build distributed applications, for object oriented developers who want to distribute their applications using C++ and for object oriented technologists who want to use DCE as a framework for building a distributed object model.

## 4. GOALS

(a) Provide a framework for distributed objects over DCE.

(b) Maintain backwards compatibility with prior versions of DCE.

(c) Generate DCE stubs in the C++ language.

(d) Allow interfaces as IDL parameters.

(e) Allow for the creation/destruction of dynamic remote objects.

(f) Allow for the registration of long lived objects.

(g) Allow for binding to long lived objects.

(h) Support single inheritance of interfaces with IDL language syntax extensions.

(i) Allow an object to re-bind to another interface in an inheritance hierarchy.

(j) Allow clients to set authorization and authentication information in an object reference.

(k) Allow multiple implementations of a common interface.

(l) Support the C++ reference operator as an IDL syntax attribute on parameters.

(m) Provide for server management of object lookup.

(n) Allow C++ static member functions in an IDL interface.

(o) Support multiple protocols in an object reference.

(p) Support new exceptions for error conditions.

(q) Allow C clients to communicate with C++ object servers.

(r) Provide location transparency for distributed objects.

(s) Provide for native C++ objects as RPC parameters.

(t) Provide for an automatic rebind policy on object references.

(u) Extend enum support with assigned values.

(v) Provide for security within an object reference.

(w) Support for automatic generation of a manager class.

(x) Support the use of a Naming service host profile along with automatic binding.

(y) Allow for client specification of memory allocation routines.

## 5. REQUIREMENTS

A customer who wishes to develop DCE applications using this new technology is required to have DCE and C++ products installed. Application users will require the C++ runtime library. No dependencies are made on certain non-prevalent C++ features such as templates, exceptions, and multiple inheritance.

## 6. FUNCTIONAL IMPLEMENTATION SPECIFICATION

### 6.1. Description of Capabilities

This section presents an illustrative example, called `Memo`, followed by a detailed description of the capabilities which this specification supports. 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.

The example uses the `auto_handle` feature of the IDL language to construct a binding handle. The advantage `auto_handle` provides is that it alleviates the need for the client programmer to deal with remoteness in finding a compatible server. Traditionally, the disadvantage of `auto_handle` is that it is slower since the Naming service must be referenced on every operation invocation that uses it. However, when a C++ object interface is used from a client application, the binding information will be encapsulated in the object proxy and member function invocations will use the encapsulated object binding information. The programmer will not need to supply a binding handle on member function invocations and will not incur the overhead involved with `auto_handle`. Non-member functions such as object constructors and static interface member functions can still be designed to use any of the DCE supported binding methods. These types of operations are actually normal DCE RPC operations in all respects except that their names are more closely associated with the interface name and are referenced by prefixing the operation name with the interface name and the `::` operator.

#### 6.1.1. Memo example: network interface (IDL file)

```
[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);
}
```

### 6.1.2. Memo example: attribute configuration file

```
[ auto_handle ]
interface Memo {
        [cxx_new(AnotherMemo)] newMemo;
}
```

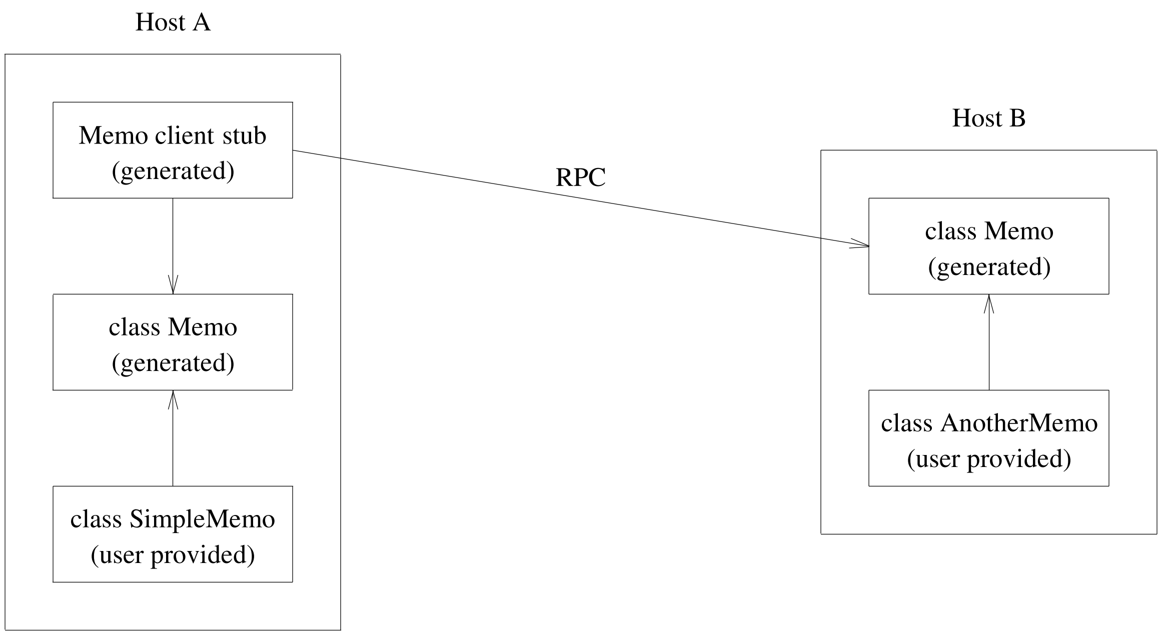
### 6.1.3. Memo example: generated network class

```
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(char * new_text) = 0;
};
```

### 6.1.4. 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(char * new_text) {
        strcat(text, new_text);
    }
};
```

### 6.1.5. Memo example: diagram of components




### 6.1.6. 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 MemoMemo class and be available on some server.

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

### 6.1.7. 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();
```

### 6.1.8. 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");
```

### 6.1.9. 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((unsigned_char_t *)
            "/.:/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, (unsigned_char_t *)
                            "/.:/messages/my_messages");
```

### 6.1.10. 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);
```

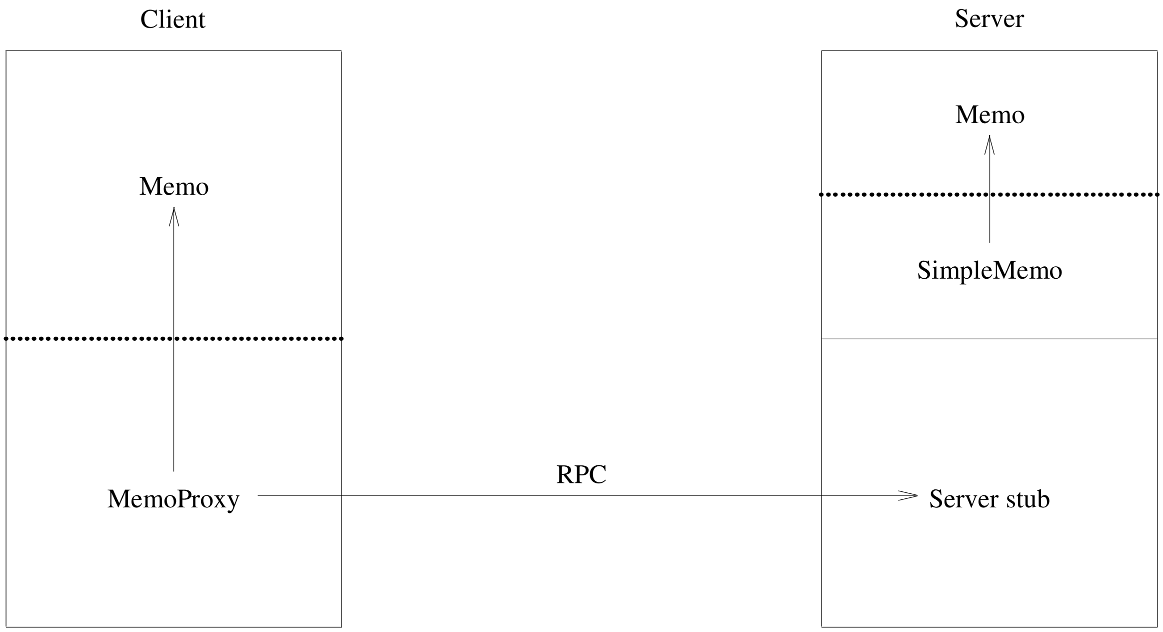
Given the following code fragment, the remote object `y` can be accessed by the local implementation `x`:

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

### 6.2. 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.




### 6.3. 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.

### 6.3.1. 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.

### 6.3.2. 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.

### 6.3.3. Object reference

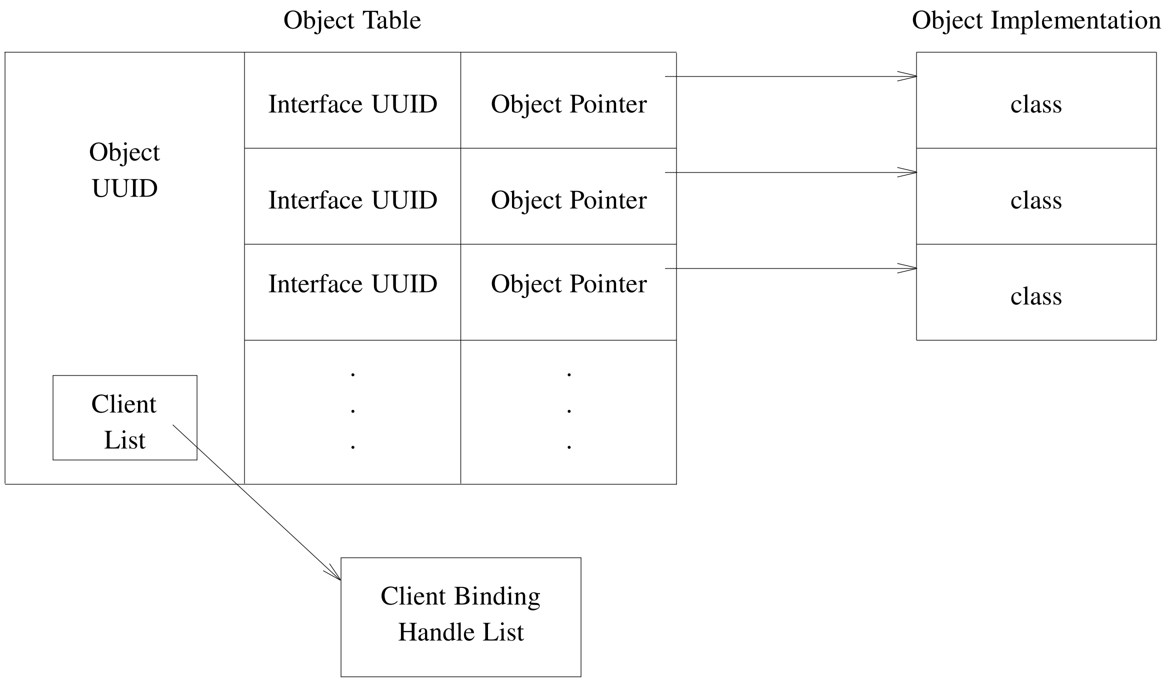
Object parameters are specified in the interface definition as the pointer type *interface-name* `*`. The wire representation of an *interface-name* `*` parameter is an `ObjectRef`, which is encapsulated in the `rpc_object_reference` base class inherited by the *interface-name* class. The IDL compiler will generate code to marshal an *interface-name* `*` parameter into an `ObjectRef` and to unmarshal an `ObjectRef` into an *interface-name* `*` type. Existing NDR types are used to construct a wire representation of an object reference, thus requiring no protocol change of new NDR type definitions.

### 6.3.4. 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(unsigned32 = DEFAULT_HASH_TABLE_SIZE);

    // Destructor
    virtual ~RpcObjectTable();

    // Subscript operator
    RpcHashTableElement &operator[](const uuid_t &);
}
```

### 6.3.5. Object table layout




### 6.3.6. Interface inheritance

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

```
<inheritance_spec> ::= ":" <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.

### 6.3.7. 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.

### 6.3.8. Object access

The manner in which client applications access objects depends to some extent upon the type of processing being performed. 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(unsigned_char_t *)` uses the `unsigned_char_t` parameter to search the name space for binding handles.

  (ii) *`interface-name`* `* bind(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(rpc_binding_handle_t)` uses the binding handle parameter as the object reference.

  (iv) *`interface-name`* `* bind(rpc_object_reference *)` uses the object reference in the base class pointer parameter as the object reference.

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

### 6.3.9. 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.

#### 6.3.9.1. 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 `TRUE`, it enables multiple servers for the same interface, but a distinct set of named objects to be running on the same machine concurrently.

#### 6.3.9.2. 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 *`name`* as a special lookup function. The lookup function allows a server to manage its own table of object pointers. The function must be supplied by the user and return a pointer to the *`interface-name`* class. The generated server stub will call the function identified by *`name`* when it discovers that the object is not registered with the object table. If the object can not be found by the server stub, an `rpc_x_object_not_found` exception is raised and propagated to the client.

### 6.3.10. 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, a private reference count is maintained by the DCE runtime for all dynamic objects. The runtime will appropriately adjust reference counts for objects as they are passed as parameters.

### 6.3.11. 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 passed 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((unsigned_char_t *)
          "/.:/MemoObjects/myObject1");
mlocal = new SimpleMemo("some text");
mremote->append(mlocal);
```

### 6.3.12. 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.

#### 6.3.12.1. 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);
}
```

#### 6.3.12.2. Matrix.acf

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

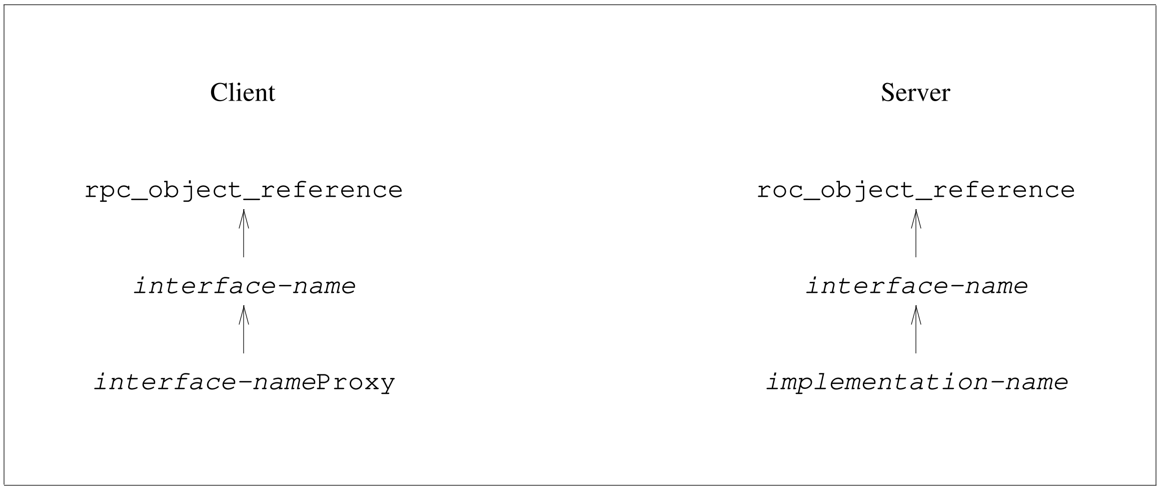
#### 6.3.12.3. Client.cxx

```
#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);
}
```

### 6.4. 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:

RPC Object Model Inheritance Diagram



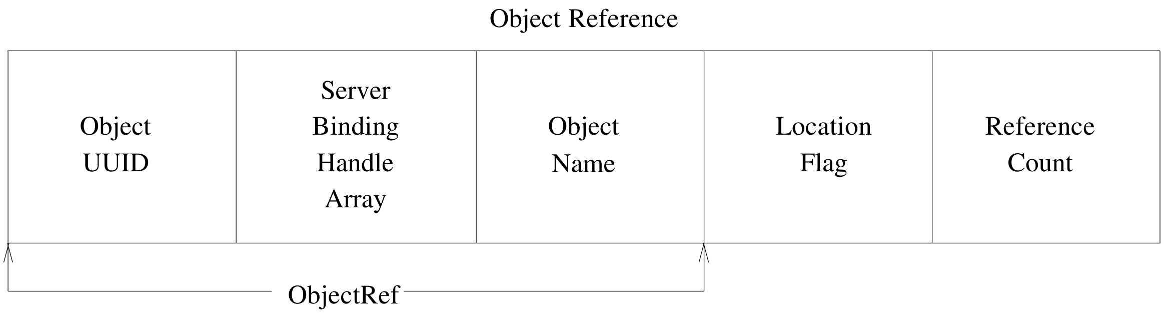

#### 6.4.1. The `rpc_object_reference` class

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 handle tower associations. 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.




### 6.4.2. 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 *`*.

### 6.4.3. 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.

### 6.5. 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 four *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. An `auto_handle` attribute would cause the operation to not require an explicit binding handle to be supplied when invoking the operation.)

### 6.6. 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.

### 6.7. 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.

### 6.8. Static Interface Operations

In order to provide the specification of a static interface operation, the IDL language is extended to support the `static` keyword and `cxx_static` ACF attribute:

(a) `static` *operation_definition*

Usage: IDL.

When used in an IDL file preceding an operation definition, identifies the operation as a static member function of the interface.

(b) `[cxx_static]` *operation()*

Usage: ACF.

When used in an ACF file as an operation attribute, identifies the operation as a static member function of the interface.

(c) `[cxx_static( name )] operation()`

Usage: ACF.

When used in an ACF file as an operation attribute, identifies the operation as a static member function of the interface which invokes the function identified by *name* from the server stub.

### 6.9. Object Security

All generated object interfaces will support an operation which allows the client to set authorization and authentication information on the encapsulated object reference.

All non-static member functions of an interface will implement remoteness with an IDL-supplied explicit binding handle parameter. This parameter will be constructed on the the client side from the encapsulated binding handle in the object proxy and passed as an argument to the operation. This handle parameter is only used to provide remoteness and is not passed as an argument to the manager object operation.

Servers that wish to gain access to the binding handle parameter from within a member function operation will be provided with an interface which returns the binding handle. Static member functions and object creator operations can specify the use of an explicit handle in the operation definition. Access to the binding handle from a member function is provided for the purpose of implementing a reference monitor for validating authentication and authorization.

### 6.10. Object Location Transparency

Client applications may wish to create local objects of a remote interface and pass them as `[in]` parameters to remote procedures. Similarly, server applications may want to access non-local objects with a remote interface as they are passed in as parameters. This can be accomplished simply by linking the server stub with the client application and the client stub with the server application respectively. Clients that pass an object reference as a parameter to a remote procedure will automatically become registered servers of the object interface.

Should the client or server application not be linked with the necessary stub, an exception will be raised indicating the need for the stub at runtime. Servers that raise such an exception will propagate the exception back to the client:

(a) `rpc_x_no_server_stub` indicates that the server stub was not linked with the client application and a local object reference is being passed as an input parameter to a remote procedure.

(b) `rpc_x_no_client_stub` indicates that the client stub was not linked with the server application and a reference is being attempted for a non-local object reference.

### 6.11. C Bindings to C++ Objects

In order to support C bindings to C++ objects, all IDL generated header files will contain macros that will allow a C++ like interface to remote objects. Macro names will be the concatenation of the interface name and operation name. The `this` pointer will be a required user supplied parameter on macros representing non-static member functions.

### 6.12. C++ Bindings to C Servers

The DCE IDL compiler already generates client and server stubs that allow access to operations from C++. By supplying an ACF attribute making all operations in an IDL file a static member function of an interface, the operations can be referenced in a slightly more object oriented way from the client application by prefixing the name with the *interface-name*`::` syntax.

### 6.13. Native C++ Objects as RPC Parameters

The passing of native C++ objects as an RPC parameter is support by two techniques. One technique involves the packaging of the object's data into a compatible NDR format and the reconstruction of the object on the receiver side of an RPC call. The other technique passes a reference to the actual object as the RPC argument, allowing the server to make callbacks to the real object.

To pass an object by value, the ACF attribute `represent_as` is applied to the class name. The developer would then supply the typical four routines to convert the C++ object to NDR format, reconstruct the C++ object from its NDR data format, and deallocation routines for the NDR and C++ representation of the data. The IDL language is extended to support the `represent_as` attribute on unique pointer types.

To pass an object by reference, the ACF attribute `cxx_delegate` is applied to an interface for the object. The IDL compiler will generate the necessary class hierarchy and implementation methods to allow a delegate class pointer as an RPC parameter. When an invocation is made on the delegate object reference, the DCE runtime will propagate the method call back to the original object transparently.

### 6.14. Automatic Rebind Policy

An object reference will contain potentially several binding handles which can be used to communicate with the manager object. If a communication or connection error occurs, the client stub can choose another appropriate binding handle to try. The policy for choosing a fail-over handle is set by the `SetRebind()` API. Choices include:

(a) `never_rebind` — Do not attempt to rebind the object reference.

(b) `attempt_rebind` — Attempt to rebind using only the encapsulated handles at most once each.

(c) `wait_on_rebind` — Attempt to rebind using the encapsulated handles until successful.

(d) `attempt_rebind_n` — Attempt to rebind using the encapsulated handles a maximum number of times.

### 6.15. Generation of a C++ Manager Class

The IDL compiler will automatically generate a manager class for the interface with the method bodies appropriately stubbed out. Developers can then derive their own manager class from the IDL-generated class and implement the various methods incrementally. This facilitates the development of server applications by insuring that all interface operations in the manager class have the correct signature while also providing an implementation for all the pure virtual methods in the interface. To prohibit the generation of a manager class, the IDL compile time option `-no_cxxmgr` is provided.

### 6.16. Exception Model

The use of the DCE exception model is supported for exceptions.

### 6.17. Extended `enum` Support

The IDL language includes an `enum` feature similar to the supported by the C language. The feature is extended to allow the specific assignment of values to `enum` types.

### 6.18. Automatic Binding and the Host Profile

The automatic binding feature of IDL queries the DCE Naming service for an entry designated by the environment variable `RPC_DEFAULT_ENTRY`. If that environment variable is not set, the DCE runtime will query the host's Naming service profile for an entry matching the interface.

### 6.19. Client Specification of Memory Allocation Routines

The IDL language is extended with an ACF interface attribute `client_memory(`*`malloc_routine`*`,` *`free_routine`*`)` to specify the memory allocation routines used by client stubs. The routines specified as the arguments to the attribute must match the system's signatures for `malloc()` and `free()`.

## 7. DATA STRUCTURES

The only new data structures exposed to the user are the class definitions generated to support a distributed object given an interface. Specifically, this class definition would have the same name as the interface it is built from.

## 8. USER INTERFACE

The command line option `-lang cxx` to the IDL compiler will cause the generation of C++ stubs for distributed object support.

The command line option `-no_cxxmgr` to the IDL compiler will suppress the generation of a C++ manager class.

## 9. APIs

(a) `interface * interface::bind( unsigned_char_t * )`

Usage: client.

Allows the client to bind an interface pointer to a long lived remote object using a name as advertised in the name space.

(b) `interface * interface::bind( uuid_t * )`

Usage: client.

Allows the client to bind an interface pointer to a long lived remote object using an object identifier and searching the name space starting at the location indicated by the `RPC_DEFAULT_ENTRY` environment variable.

(c) `interface * interface::bind( rpc_binding_handle_t )`

Usage: client.

Allows the client to bind an interface pointer to a long lived remote object using a binding handle. The name space is not accessed for this operation.

(d) `void interface::register_named_object( unsigned_char_t *, boolean32 = TRUE)`

Usage: server.

Allows the server to advertise and optionally register an object with the Naming service and DCE runtime respectively.

(e)
```
void interface::secure(
      unsigned char * = 0,                        // svr princ name
      unsigned32 = rpc_c_protect_level_default // prot level
      unsigned32 = rpc_c_authn_default          // authorization
      rpc_auth_identity_handle_t = NULL,        // auth identity
      unsigned32 authz_svc = rpc_c_authz_name  // authorization
)
```

Usage: client.

Allows the client to set authorization and authentication information in an object reference.

(f) `handle_t interface::get_binding_handle()`

Usage: server.

Allows access to the caller's binding handle within a manager class method implementation. Per call binding handles are maintained in thread specific storage.

(g)
```
void interface::SetRebind()
      DCERebindPolicy,    // rebind policy
      unsigned32 = 0      // for attempt_rebind_n
)
```

Usage: client.

Allows the client to specify a rebind policy for interface method invocations that fail due to communication or connection failures. The default policy is `attemp_rebind`.

## 10. REMOTE INTERFACES

None.

## 11. MANAGEMENT INTERFACES

None.

## 12. RESTRICTIONS AND LIMITATIONS

The use of non-prevalent C++ compiler options such as C++ Templates and C++ Exceptions is avoided in the interest of portability.

## 13. OTHER COMPONENT DEPENDENCIES

### 13.1. DCE Naming

Named objects that are advertised by a server use the Naming service APIs to advertise the object in the name space. The use of this capability is optional.

### 13.2. DCE RPC

To transmit an object reference, an internal tower data structure is used to enhance the efficiency of the remote procedure call. The DCE RPC component requires the addition of two internal system programmer interfaces to convert between a binding vector and a tower representation.

### 13.3. C++ Compiler

A new runtime library will contain all the C++ support for DCE distributed objects. This library requires a C++ compiler in order to be built. Vendors not wishing to integrate this C++ support into their DCE products do not need to build this library, therefore eliminating the requirement of a C++ compiler for building DCE 1.2.

## 14. COMPATIBILITY

The IDL compiler with C++ extensions is fully backwards compatible.

## 15. STANDARDS

None.

## 16. OPEN ISSUES

None.

## 17. REFERENCES

None.

## AUTHOR'S ADDRESS

Bob Viveney
Digital Equipment Corporation
110 Spit Brook Road ZK2-3/Q18
Nashua, NH 03062-2698
USA

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