OSF DCE SIG **J. Dilley (HP)**
Request For Comments: 49.0 **October 1993**

OBJECT-ORIENTED DISTRIBUTED COMPUTING WITH C++ AND OSF DCE

1. INTRODUCTION

This paper reports on the results of a project aimed at simplifying the development of distributed applications based upon the OSF Distributed Computing Environment (DCE). The core of this package is a set of C++ classes for developing object-oriented distributed applications using the C++ and DCE technologies. This RFC proposes a model to map DCE onto C++ and discusses our experiences building a prototype DCE/C++ system. Through the use of this class library we saw a significant decrease in DCE application code size and an increase in developer productivity.

This project defined a class library that hides DCE's complexity from application developers and a mapping that converts DCE interface definitions into C++ client and server classes. Developers implement the server class methods to provide an application's intended behavior. The methods of client "access classes" (client proxy classes) make a DCE remote procedure call (RPC) to access remote objects. Users of remote objects therefore use the same programming model they use for making local method calls, preserving the object-oriented development model.

Default behavior for DCE functions such as security and name space registration is provided with the C++ class library. This reduces application complexity because suitable behavior is provided automatically by the library. If applications require different behavior, it can be provided within the framework defined by the class library by deriving from a policy class and providing a new implementation. Providing a suitable default implementation reduces the learning time of DCE, and will increase the consistency of distributed applications if they all use the same behavior.

By integrating the DCE exception and error models with C++ exceptions we were able to provide a consistent error model and make use of C++ language support for remote and local exceptions. Use of exceptions will reduce application code size as the checking of error return codes is no longer necessary.

1.1. Why DCE in C++

The benefits of building systems using object-oriented techniques have been widely discussed in the literature. Korson and McGregor [KM] provide a thorough examination of the concepts of object-oriented development. Nicol, et al. [NWM] explore use of object orientation for building distributed systems and discuss some current distributed object-based systems including the existing DCE object model and OMG's CORBA [OMG] distributed object system.

We chose to use C++ to encapsulate and abstract DCE functionality for a number of reasons:

(a) Object orientation provides the ability to create abstract interfaces and to hide data, allowing developers to work at a higher level. C++ also facilitates interface and code reuse.

(b) C++ is becoming the prevalent language for object-based systems development. Increasing numbers of C++ class libraries and development tools are becoming available.

(c) C++ has a clean and natural interface to C, and therefore to the existing DCE implementation.

(d) The C++ object model is similar to the object model used in DCE.

These last two factors help to make the resulting system intuitive and therefore easier to learn.

1.2. Requirements

Following are a set of suggested requirements for a class library for DCE.

(a) Simplicity — The class library should be easier to learn and use than the underlying technology. There should be no need to learn a new object model, just the one inherent in DCE.

(b) Coverage — While making application development simpler, the library should also provide the ability for the developer to access the full functionality of DCE.

(c) Integration — The library should integrate cleanly with the current DCE object model and implementation. It should encapsulate the current DCE behavior in a natural fashion.

(d) Performance — The class library should allow an application to perform within 5-10% of a C-based DCE application with equivalent features.

(e) Customizable — The developer should be able to derive classes from certain library classes to customize the behavior of applications. This is important for classes that provide policy for interacting with DCE components such as naming and security.

(f) Extensible — The library should allow developers to derive new classes from those provided in order to support new capabilities.

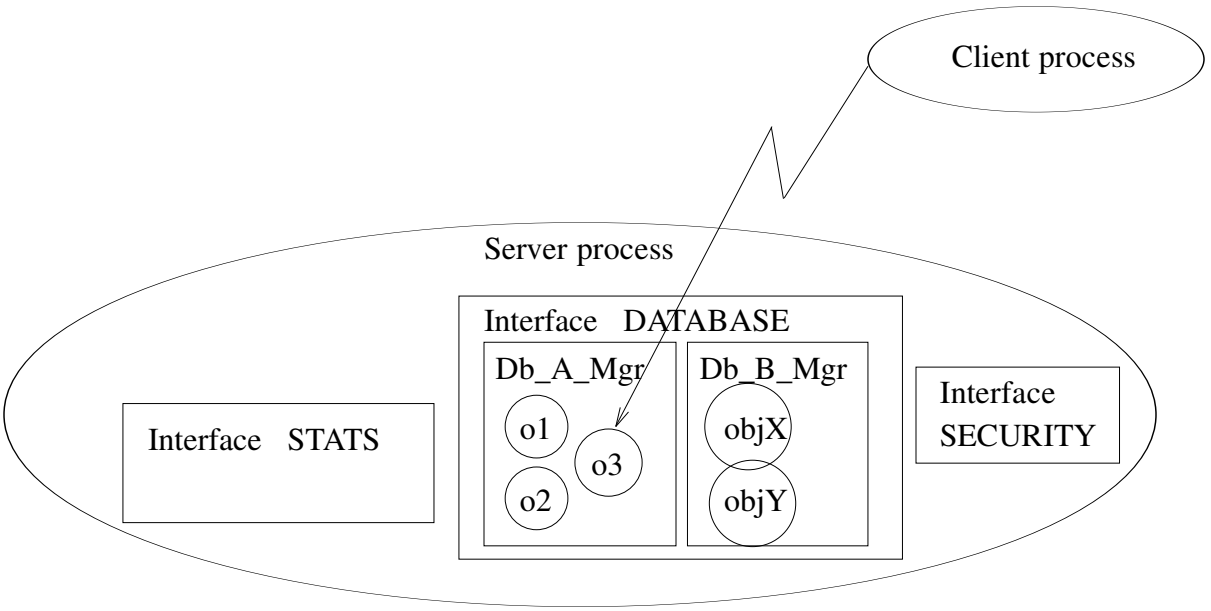
(g) Standard — The library should build on existing DCE components. It should not require the replacement of any standard DCE component.

2. DCE OBJECT MODEL

This section provides an overview of the DCE object model. The DCE object model is a conceptual one, in which DCE entities are thought of and manipulated as "objects", even though they are not implemented in an object-oriented language. The DCE C++ class library aims to expose this model and provide an object-oriented implementation for it.

DCE servers provide the services of one or more objects to clients through well-defined interfaces. Servers are effectively object managers — the server interacts with the DCE runtime environment to make the services of its objects available to clients.

DCE interfaces (IDL specifications) define a set of data types and remote operations that can use those data types. A client performs a remote operation by making a remote procedure call (RPC) to a compatible server (one which implements the desired interface and exports the desired object).



A server can export multiple DCE interfaces. The Figure shows an example of a server that exports three interfaces. Its main interface is a database interface; in addition the server also

exports a statistical interface to allow clients to retrieve usage statistics about the database interface, and a security interface to allow database managers to control client access to the database.

A DCE object is a logical entity exported by a DCE server. A DCE server contains (and manages) one or more DCE objects. Each DCE object has the notion of a class, called in DCE its implementation type. The implementation type is associated with a set of manager routines (through a DCE entry point vector, or EPV). The server developer writes the manager routines to perform the actual remote operations defined in the interface definition file. Each implementation type implements a given interface and must meet the external specification defined by that interface. But the manager routines for each implementation type may employ different internal data structures and algorithms to accomplish the remote operation.

In addition to supporting multiple interfaces, the server in the Figure also exports multiple implementation types within its database interface. The implementation types A and B each implement the common database interface, but each for a different type of underlying database.

A server registers its interface and objects with the DCE runtime environment and awaits incoming RPCs. Each RPC is dispatched by the DCE runtime to the specific manager function that implements the interface and implementation type for the object requested by the client.

A DCE client can establish a binding to an object within a server by specifying the interface and object identifier it wishes to access. The client code and the DCE runtime use this information to select an appropriate server and to initiate the RPC. By selecting an object, the client can select which logical entity it wishes to access; in the Figure the client has bound to the object o3 so all interaction is done conceptually with that DCE object. In DCE it is the manager function's responsibility to determine which object is being accessed and to take the steps necessary to use that object to satisfy a remote request.

3. DCE C++ OBJECT MODEL

We considered two approaches to constructing a set of class libraries for DCE. One was to write wrapper classes for each of the basic DCE data types and wrapper functions for each of the interfaces delivered with DCE. These classes and functions would be patterned directly after the data types and functions defined in the DCE specification and would tend to look and operate just like the underlying types and functions.

The other approach was to create a set of classes that build an abstract model of the data and functions needed by a DCE application. These classes would not be patterned after the data types themselves but rather based upon the set of tasks or responsibilities supported by the DCE model. The implementations of these classes would use DCE data types, but do so within an infrastructure independent of those data types; in many cases the users need not be aware of the underlying data types.

With the wrapper classes, each underlying interface is typically mapped into a method call, providing little simplification over the current API (application program interface). With this approach developers are tied to a particular implementation of the product. Changes in future releases of the base implementation are more likely to require changes to customer code, possibly using a different set of wrapper classes. Wrapper classes are still essential in many instances to encapsulate basic data types even when using a task-based model. An example of this might be the encapsulation of native data types, providing the developer a clean interface to those data types and allowing the use of polymorphism and operator overloading.

The task-based approach requires more up-front design work by the class library developer, but in the end can yield a system that is more consistent for the application developer, and provides a more natural object model for accessing the product. Task-based systems may also be able to survive changes in their underlying infrastructures, provided the model presented by the new infrastructure remains similar.

We chose to create a set of task-based classes to support the object-oriented DCE (OODCE) model, along with a set of wrapper classes to encapsulate basic DCE data types. The approach is discussed further in the following sections.

3.1. OODCE Model

The OODCE model preserves concepts of the DCE object model while providing convenient, natural C++ access to DCE objects. Each DCE object is modeled as one or more C++ objects. Most major user-visible DCE data structures are modeled by C++ classes for convenience of access. With this class library the developer deals primarily with C++ objects in the construction of DCE applications.

An interface is mapped by an enhanced IDL compiler into a set of client and server classes that represent that interface. The member functions of these classes correspond to the remote operations defined in the IDL file. The IDL data types are left unmodified, as they must be treated as regular C data types by the RPC runtime.

Within a DCE server, instances of the server-side objects are created and provide the desired behavior for the server. Client-side object instances (proxies or surrogates) are used to access remote objects as if they are local objects.

Instances of DCE objects within a server can be dynamically created to serve individual client requests, or can be created at server startup time to handle multiple client requests. References to objects can be passed between clients to allow different clients to access the same remote object within a server.

Client access class instances can locate an appropriate object through the directory name space (CDS), through a remote host's endpoint map, through an object reference, or directly via a fully bound DCE binding handle.

4. IDL TO C++ (OODCE) COMPILER

The IDL-to-C++ "compiler", idl++, is actually a script that translates an interface definition into the interface-specific client and server classes. The compiler first runs the standard DCE IDL compiler to create the client and server C stubs and the interface header file containing the operation and data type declarations. The compiler then parses the header file and creates a set of C++ classes for each interface defined in the IDL file. The client access classes contain member functions that allow C++ method calls to invoke the RPCs defined in the IDL file. On the server side the compiler generates two classes: the abstract implementation class for the interface defined in the IDL file, and a default instantiable implementation class derived from this abstract class.

In addition to the server-side implementation classes, the compiler generates a C++ entry point vector (EPV) data structure, a set of server stubs, and an object mapper, which maps an interface and object ID pair to a C++ object instance. The C++ EPV controls which C++ server stub receives the calls from the DCE runtime when an RPC is received. The server stub calls the object mapper to get the specific C++ object instance to which the call is being made. The stub then calls the desired member function on the object returned by the object mapper and handles the mapping of DCE exceptions.

The data types declared in the header file are left untouched by idl++: no mapping of IDL data types to C++ is necessary since the data types supported by IDL are a subset of the available C++ data types (see the IDL specification [OSF]). Any data types defined in the IDL file are used by the application developer as C data types. The prototype does not permit passing C++ objects as arguments to remote procedure calls.

The current implementation of idl++ is a script that processes IDL output. This capability should be rolled into the standard IDL compiler, controlled by an appropriate compile-time flag.

5. OODCE CLASS LIBRARY

This section discusses the classes provided within the OODCE class library. The framework and utility classes are discussed first, followed by the classes generated by idl++ and the developer-defined classes that make up the rest of the distributed application. Section 4 discusses the compiler in more detail.

5.1. DCE Framework Classes

DCE framework classes are responsible for providing the DCE interface abstraction. They define a particular object-based view of the distributed system. Client and server implementation classes, generated by idl++, inherit their interface and default behavior from these framework classes. Other framework classes are used to control the behavior of various DCE subsystems, including security, naming, and threads. Each framework class provides an interface (typically abstract) that defines how that class interacts with the rest of the environment. Many classes also have implementations associated with them to provide the behavior defined by those classes. The framework

classes can be specialized by derivation and overloading of virtual functions. Certain classes may not have default implementations; instead they define only the interface (policy) for how to interact with that component of DCE. Instances of those classes must be written by the application developer to provide the behavior for that DCE facility.

The framework classes are discussed in the following sections.

5.1.1. Server

This class implements the portion of a DCE server that interacts with the DCE environment. A single instance of the server class manages all the objects and interfaces exported by that server. The Server class is responsible for calling rpc_server_listen to start the server runtime listening for and dispatching incoming RPCs.

Process-global policies, such as for retrieving security keys, can be specified to the server by registering a policy implementation object with the server.

5.1.2. InterfaceMgr

This is the server-side abstract base class. Each interface manager instance is associated with a DCE interface handle (generated by the IDL compiler), a DCE entry point vector (EPV, which contains pointers to the functions that implement the server's manager routines), and optional type and object identifiers (UUIDs) for that object instance. The information in the interface manager class is registered with the DCE runtime by the server class.

Derived classes of InterfaceMgr, generated by idl++, add member functions corresponding to the interface-defined remote operations. The derived classes are referred to as server implementation classes. Their member functions are implemented by the server developer.

5.1.3. Interface

This is the client abstract base class. It holds the common client functions for specifying a remote object with which to communicate. The Interface class provides the ability to control [re]binding, security principal information, and authentication policy. Type conversions are provided from the Interface type to a DCE client interface handle (the rpc_if_handle_t type) or to a string. Derived classes of Interface, generated by idl++, provide client member functions to access the remote operations. These classes are referred to as client access classes.

5.1.4. Exception

The exception abstract class defines common behavior for DCE exceptions within C++ applications. Derived classes implement the specific exceptions. See Section 6 for a discussion of the OODCE exception model.

5.1.5. RefMon

The reference monitor abstract class defines the method for creating DCE reference monitors, which are used by servers to grant or deny access based upon a client's use and level of DCE authentication. Each interface manager can have a reference monitor associated with it. The interface manager class calls the reference monitor object to determine whether the object's manager function should be called based upon the client's security level.

5.1.6. Password

The password abstract class provides an interface for the storage and retrieval of DCE security principal passwords. The interface is used by other members of the framework (mainly LoginContext) to retrieve user password information. No implementation for this class is supplied, as there are many possible ways an application might wish to retrieve a password from a user; one possible way is to pop up an X windows dialog box.

5.1.7. LoginContext

The login context abstract class defines the behavior required to maintain a DCE login context for a given security principal. This class encapsulates the sec_login_handle_t data structure. The default implementation supplied starts a thread to keep the login context and associated keys validated. It uses the Password class to retrieve the DCE principal's password from the user.

5.1.8. KeyRetriever

The key retriever abstract class provides an interface to access security keys. The server class uses the key retriever to implement its required security policy. The default implementation accesses keys in a system keytab file, but other implementations can be supplied to provide alternative methods for key retrieval.

5.1.9. MgmtAuthorizer

The management authorizer abstract class defines an access control mechanism for DCE server management functions (rpc_mgmt_inq_*). Implementations derived from this class can be used to check the credentials of clients attempting to execute these management functions. The server application code must register a management authorizer object with the server object, if one is desired. If no management authorizer is registered the server will use the default DCE behavior.

5.2. Utility Classes

The utility classes encapsulate the basic DCE data types to make their use more convenient within C++. The purpose of these (wrapper) classes is to allow convenient construction and use of these data types. Many provide operators to convert to the corresponding DCE C representation (such as to a string or uuid_t), allowing them to be passed directly to DCE calls without need for separate translation.

5.2.1. Uuid

The Uuid class encapsulates a DCE uuid_t. It can be constructed from a string or uuid_t. The Uuid class provides equality tests against uuid_t and char* for convenience. Uuid provides conversion operators to uuid_t and char*.

5.2.2. UuidVector

The UuidVector class implements a vector of Uuid objects. They can be converted into the corresponding DCE UUID vector type.

5.2.3. Binding

The Binding class encapsulates the DCE binding handle type, rpc_binding_handle_t. A Binding can be constructed from string or from binding handle components. They can be converted to a rpc_binding_handle_t or to char*.

5.2.4. BindingVector

The Binding Vector class implements a vector of binding handle objects. They can be converted into the corresponding DCE vector type.

5.2.5. SecKey

The SecKey class provides an interface to a security key used by the DCE security system, DCE type sec_passwd_rec_t. Keys are normally stored in keytab files; this class provides access to a single security key and the information associated with this key.

5.2.6. SecKeyStore

The SecKeyStore provides access to a keytab file, which can contain a number of security keys. The SecKeyStore class allows insertion of keys into keytab files, and access to the keys already in the keytab file.

5.2.7. EndpointMap

The EndpointMap class provides an abstraction onto the endpoint mapper implemented by the rpcd process on a DCE host. This class can be used to browse, modify, and query endpoint information.

5.2.8. CdsObject, CdsEntry, CdsGroup, CdsProfile

These classes provide access to information in the DCE Cell Directory Service (CDS). The CdsObject base class provides the ability to lookup bindings from a CDS RPC NS object; the derived classes CdsEntry, CdsGroup, and CdsProfile provide access to the specific information in RPC NS server entries, groups, and profiles.

5.2.9. Pthread, PthreadMutex, PthreadCond

These classes provides an abstraction on a DCE thread, with the ability to spawn a new thread and check the status of a thread. Also includes classes for access to Pthreads attributes, mutexes, and condition variables.

5.3. Generated Classes

The generated classes are created from the interface definition by idl++. Their structure is described in detail in the following sections.

5.3.1. Client access class

The client-side access class is derived from the abstract base class Interface. The access class has methods corresponding to the operations (remote procedures) defined in the IDL interface. The client remote operation methods handle the binding to a server and then call the corresponding C stub generated by the DCE IDL compiler. The client methods also map DCE exceptions returned by the RPC into C++ exceptions.

5.3.2. Server implementation abstract class

For the server, idl++ generates an abstract class derived from InterfaceMgr. This class, called the server implementation abstract class, provides specifications for the member functions corresponding to the remote operations declared in the interface definition file.

5.3.3. Server implementation concrete class

Instantiable (concrete) classes derived from the implementation abstract class must implement all of the member functions defined in the abstract class to provide the operational characteristics of the server. The compiler generates one such class by default with a nil manager type UUID (the nil manager class). If multiple implementation types (and therefore multiple managers) are needed, additional classes can be derived from the abstract manager class.

Each object (instance) of an implementation class must have its own object UUID and must be registered with the server class so the C++ server stub can locate it when a call comes in for that object.

The implementation classes implement the manager methods in C++ in the same way the manager functions are implemented in a C-based DCE server. The manager methods use the data passed in as arguments, perform their task, and possibly return data to the caller.

If manager methods throw C++ exceptions they will be caught in the C++ server stub and transmitted back to the C++ client (as a DCE-compatible data type) where they are raised again as C++ exceptions for the client to catch.

5.4. Developer-Defined Classes

In addition to all the classes in the class library and those generated by the IDL compiler, application developers have an opportunity to write some classes of their own! Developer-defined classes provide whatever implementation behavior the ultimate application requires. For example, an application might have a graphical user interface (GUI) that collects input and makes RPCs on behalf of the user. In this type of application, the GUI classes would typically control the application — the client main routine would create instances of the desired GUI and DCE client access classes, initialize the GUI, register the access classes with the GUI, and then await user input. Remote method calls would probably be made in response to various user inputs. The server implementation classes may also wish to use C++ objects in order to accomplish their tasks.

The OODCE application developer must also implement any of the framework or utility classes for which implementations have not been provided. These abstract classes define an interface that must be obeyed, but have no default implementation. It is up to the application developer to define the behavior or policy that they wish for their applications.

6. OODCE EXCEPTION MODEL

The area of exceptions required quite a bit of work in the design of the class library. The C++ language-based exception model is by far more powerful and better integrated with the language than the DCE CMA exception package. Therefore the C++ exception model was used as the basis for all exception handling in this class library.

The first challenge was that the C++ and C-based exception mechanisms cannot interoperate — an exception thrown or raised by one mechanism cannot be caught and interpreted directly by the other. Furthermore, an exception raised in one language context cannot be allowed to pass through blocks from the other language. If this is allowed to happen, cleanup code might not be properly executed. The DCE exception facility uses setjmp and longjmp; longjmp allows control to pass from one stack frame to another frame, possibly much earlier in the call stack. The trouble is that destructors for automatic variables aren't invoked. Allowing a DCE exception to skip over this code can cause consistency problems and memory leaks in the C++ application.

To model DCE exceptions in C++, an abstract base class OSFException was defined; this class specifies the common behavior for all exceptions. Two more base classes, DCEexception and CMAexception, were derived from OSFException to model the distinct types of exceptions that can be raised by the DCE and CMA (threads) subsystems. The DCEexception class was further subdivided into RPC, security, and directory service exceptions. Each exception that can be raised either by the DCE or CMA subsystem was created as a subclass derived from one of these base classes. Each of the specific exception classes has a method that will print out an informative description of the exception, and can be converted into a string through an operator char*.

The choice to model exceptions as individual classes, instead of as a generic class with an exception value, allows individual exceptions to be caught by class — in the C++ exception model any specified data type can be caught. With the generic exception-with-value model, fewer classes would be needed, but if you want to catch a particular exception you would have to catch the generic exception and test its value.

6.1. Server Stubs

Exceptions cannot be directly passed across the network back to the DCE client. Any exceptions raised by the C++ manager methods must be caught and translated into a data type that can be raised again as a C++ exception in the client. To facilitate this, the idl++ compiler adds a hidden status parameter to the interface definition. That status parameter holds the unique integer value of the exception that was raised. On the client side that integer is thrown again as a C++ exception. Since an integer type is used, only the DCE error_status_t and unsigned32 data types can currently be transmitted back to the client. Any other exception type is mapped to a generic exception code.

6.2. Client Stubs

The C++ access method calls the DCE C stub, which can raise a variety of DCE exceptions, including communication status, fault status, and user defined DCE exceptions. To prevent problems caused by the inconsistent DCE and C++ exception models, the DCE C stub call is wrapped by a CMA TRY/CATCH clause. If an exception is caught in the C++ stub, it is rethrown in C++ as the corresponding exception subclass.

7. APPLICATIONS PORTED

We ported a set of sample applications from the HP DCE Developer's Toolkitto experiment with the class libraries, with the InterViews C++-based GUI [LVC], [IPPG], and with object database technologies. The applications ported are described in the following sections. More detail about these applications can be found in [HP], available as part of the HP DCE Developer's Toolkit.

Our application development environment was HP-UX 8.0 and 9.0, with HP DCE/9000 release 1.0, which is based upon OSF DCE 1.0.1.

7.1. sleeper

The sleeper sample application is a very simple application whose interface takes an integer number of seconds to sleep. The manager sleeps for that many seconds and then returns. This application demonstrates how small the simplest DCE application can be when using C++ as compared with using DCE directly.

7.2. rmt_load

The remote load application monitors the load average of a group of server systems. The application client concurrently requests the load average of each of a set of servers by doing RPCs in separate threads. Each server returns its one-minute load average (as emitted by the uptime program) which is then reported by the client.

In our system, the remote load client was given an InterViews-based GUI to display the load averages. Only the client was ported; it contacted existing C-based servers to demonstrate interoperability between C and C++ DCE

applications and the ability to write a threaded DCE application using these class libraries and an X11-based GUI technology.

7.3. authrpc

The authrpc client uses the authenticated RPC component of DCE security to prove to the server that it is not an impostor, and to verify that the server is not an impostor. The authrpc application is based upon sleeper and was used to test the implementation of authenticated RPC.

The class libraries provide access to the DCE runtime authentication calls through client and server side method calls that allow the specification of the security-relevant parameters the RPC runtime should use in the client or require in the server.

7.4. refmon

The refmon application implements a DCE security reference monitor, which allows the server to grant or deny service to a client based upon the client's authentication preferences (status, authorization policy, and protection level). This application is based upon authrpc since authenticated RPC is required to implement a reference monitor.

A default reference monitor class is provided with the class libraries. The C++ EPV generated by the idl++ compiler makes a call to the reference monitor if an instance of the RefMon class was registered with the server object.

7.5. phone_db

The phone database application looks up personal phone number and email information. The UI allows specification of name or regular expression search criteria. With that information the client makes an RPC to the database server and displays the results. The phone_db server maintains a database of names and related information. It uses the HP OpenODB object database to hold the data.

The server manager functions were leveraged from the C version. They maintained the same interface but were reimplemented to use OpenODB underneath. The client was almost entirely rewritten due to the new InterViews graphical user interface.

This application uses DCE security to control access to the database for both the read and update operations.

7.6. Example Code

This section contains sample code from the sleeper application to illustrate what a minimal application that uses this library looks like. What follows is all the code the application developer needs to write.

7.6.1. IDL file

```
[uuid(...), version(1.0)]
interface sleeper
{
 [idempotent] void Sleep
 ([in] handle_t h,
 [in] long time); /* Sec. to sleep */
}
```

7.6.2. Server main

```
void main()
{
  // try block for exc handling
  try {
    // Construct the Sleeper object
    sleeper_1_0_Mgr sleeper;

    // Register with server object
    theServer.RegisterObject(sleeper);
```

```
  // activate the server
  theServer.Listen();
 }
 // Catch any DCE related errors
 catch (DCEErr& exc) {
    cout << "Caught DCE exception"
        << (char*)exc;
    exit(1);
 }

 // Destructors are called here.  The
 // server object takes care of
 // cleanup with the DCE runtime
}
```

7.6.3. Manager code

```
void sleeper_1_0_Mgr::Sleep(long time)
{
 sleep((unsigned int)time);
}
```

7.6.4. Client main

```
main(int, char** argv)
{
 // Set up a try block to catch exc.
 try {
   // Construct an instance of the of
   // the Sleeper Client object using
   // Network Address and protocol seq
   sleeper_1_0     sleepClient(
            (unsigned char*)argv[1],
            (unsigned char*)"ip");

   long          sleep_time;

   sleep_time = atoi(argv[2]);

   // Call member function that
   // invokes the remote procedure on
   // the server object.
   sleepClient.Sleep(sleep_time);
 }

 // report any exception that occurred
 catch (DCEErr& exc) {
   printf("DCE Exception: %s0,
        (char*)exc);
 }
}
```

8. RESULTS

Porting the existing C applications to C++ took very little time — the main time-consuming tasks were the removal of the code to interface with DCE and the addition of the InterViews GUI. Since the interface definitions remained the same, the code dealing with the data types and remote procedures was highly leveraged. In some of the

applications the manager code remained in C, as it was perfectly adequate. This saved development time, and illustrates the integration of the C and C++ environments.

All the GUI work was done in C++. Since we were using a new UI technology, this was all new code. We found that interfacing the GUI code with the C++ client access class was quite natural.

To summarize, the primary benefits we experienced after use of this technology were:

(a) Providing powerful abstractions on top of DCE allowed us to concentrate on developing the application, not writing code to interface with DCE.

(b) There is a significant amount of complex code in the library that the user no longer has to write. In particular, the code to deal with the security subsystem is complex and hard to learn. Having easy-to-use library calls is a major benefit.

(c) Application development and debugging time were shortened significantly because the basic DCE calls are encapsulated in an already-tested library.

(d) Having sensible defaults for many DCE values prevented the need for redundant code by allowing code reuse; also having defaults prevents users new to DCE from having to make choices they may not be prepared to make.

(e) The library provides full coverage of the DCE interface, preventing the need to code to the DCE API. The underlying DCE features are all still accessible by customizing classes provided with the library, so use of the DCE is not limited by this package.

(f) The C++ DCE application source code size is significantly smaller than the C code. We noted a five-to-one decrease in the actual lines of code that dealt with the DCE environment in the sleeper application. However, the C++ DCE application object size is predictably larger than the C version. The object size grew by 30-40%, mainly due to the addition of the C++ exception mechanism and the OODCE exception classes.

(g) No performance measurements have been taken yet.

(h) The C++ exception model is more powerful and useful than the DCE exception model. A greater variety of exceptions can be transmitted more easily across the RPC and handled in the client in a natural language-supported way.

(i) Having standard policies defined for name space registration and security should assist in making future applications using this library consistent with each other. This will reduce the management effort required to maintain a set of client/server applications.

(j) The higher-level abstractions seem to be easier to learn than when using DCE directly. It is hard for us to judge this fairly as we were all proficient with DCE before porting the sample applications.

9. RECOMMENDATIONS

We have demonstrated it is possible and useful to develop object-oriented distributed applications with C++ and DCE, but there are still technical issues with doing so. We have come up with a set of recommendations intended to smooth the road for others developing distributed object-oriented applications using DCE and C++. Our recommendations fall into two classes: vendor and user recommendations.

9.1. Vendor Recommendations

The sooner thread-safe libraries are available, the easier it will be to develop applications using DCE threads. In particular, the C++ runtime library, X11 and GUI technologies built upon it, and commercial products such as databases must be made thread-safe.

The availability of thread-aware distributed debuggers will also greatly aid in the creation of DCE applications.

The DCE CMA and C++ exception mechanisms must be reconciled. One alternative is for the CMA package to use the same basic mechanism as the C++ exception package. This would allow consistent exceptions to thrown from one language to the other.

Users would benefit from a standard approach to developing DCE-based C++ applications. Cooperation between OSF and DCE vendors can bring this about.

9.2. User Recommendations

The key issues for users to be aware of are those that impede integration of the object-oriented and distributed computing technologies. Users should:

(a) Verify whether the C++ compiler emits thread-safe code (most do), and whether its libraries are thread-safe (most aren't).

(b) Verify whether X11 or other library calls can be made from multiple threads concurrently. If not, the application must either wrap calls to those subsystems or ensure that only a single thread will call them, to prevent reentrance problems.

(c) Make sure the DCE exception mechanism is properly dealt with: applications must not allow DCE or CMA exceptions to escape past C++ blocks. When using the OODCE library, the application should not make calls directly to DCE stubs.

10. RELATED WORK

There are many distributed object systems documented in the literature with a variety of goals. The Arjuna system [SDP], [P] focuses on fault tolerance and persistence using a custom RPC mechanism built atop an existing kernel. The Clouds project [DLAR] built a distributed, object-based operating system using a custom microkernel and remote object invocation implementation. Emerald [RTHBHJ] defines a language that uses an object-thread approach to provide distributed object communication. These are primarily research projects exploring the operation of distributed object systems using custom platforms. Schill [S] presents a model for building an object-oriented distributed system within the framework of Open Distributed Processing (ODP) on top of an existing OSI-based infrastructure.

By contrast, our work focuses on integrating C++ within the existing DCE system infrastructure, simplifying the use of the DCE object model.

OMG CORBA will address creating distributed object systems in C++; some implementations will run on top of DCE. CORBA IDL provides for interface inheritance, which DCE IDL is lacking, and provides a more C++-like syntax for interface specification. The CORBA runtime environment provides a richer set of object invocation and passing than the DCE environment.

We suggest that our work may assist in the migration from DCE to CORBA by providing an intermediate C++-based distributed object system until CORBA implementations are widely available. Migrating from DCE/C++ to CORBA should be easier than migrating from C or from C++ using explicit DCE calls, because the direct use of DCE APIs will no longer be necessary. In addition, the use of object-oriented design and programming should assist in making the resulting application more portable into a C++-based CORBA environment than a regular DCE application would be.

There are at least three other current C++-based DCE object systems. One is the DCE++ system proposed by HaL Computers [LK]. The HaL solution is fundamentally a wrapper-based approach, providing a C++ interface to the underlying DCE components, but it does not provide a unifying object model. It provides the ability to compile a C++ interface definition into a client and server stubs using a custom stub generator; it uses a common base class for all distributed objects.

The second approach is DCE++ from the University of Karlsruhe [M]. This offering provides a new distributed object model based upon fine-grained distributed objects and dynamic object migration. Migration is supported by proxy servers (tombstones), and a per-node location daemon assists in the location of objects (supported by DCE servers). Application classes in this framework are all derived from a common base class; they require a separate, parallel interface and thus an extra compilation. This scheme is somewhat more complex than the one described here, and seems to address a different set of goals.

A third C++ DCE class library has recently been published by DEC [A]. This DCE RFC describes a mapping from IDL to C++ through extensions to the attribute configuration file (ACF) facility of the DCE IDL compiler, and therefore modification of the IDL compiler's stub generator. This work approaches the problem from the

standpoint of distributing C++ objects (although without formal language support), using IDL as the language for describing distributable objects. Among the areas addressed are dynamic object invocation, object migration, and the ability to pass object references between processes. Interface inheritance is supported at the IDL level. The object migration scheme uses a forwarding mechanism to redirect client requests from a migrated object to its new location. The distinction between local and remote objects is made transparent, allowing a client to be written without needing to know whether an object is a local copy or a surrogate (with a few exceptions). This proposal does not address DCE security, threads, or exceptions.

11. CONCLUSIONS

Using object-oriented design and development techniques provides significant benefits in distributed systems development. In particular, the class libraries and compiler described here significantly reduce the burden of creating DCE applications, and allow creating object-based DCE systems using C++.

We suggest that a standard approach to providing C++ access to DCE would benefit customers and DCE vendors alike. It is our intention to work through the DCE SIG to help make this happen.

12. FUTURE WORK

There are several enhancements that we have considered for this work. One key area is to provide support for distributed transaction processing with Encina. Migration of applications using this library to the CORBA environment is also being studied.

Although not described in this document, the OODCE library does contain a full implementation of a DCE security ACL manager. The ACL manager will be described in a future version of this document.

13. ACKNOWLEDGMENTS

The idea and initial design and implementation of this project were done by Jeff Morgan. We wish to thank Deborah Caswell and Bob Fraley for their advice and contributions to this library, and Mickey Gittler, John Griffith and Mike Luo for their review and feedback on this project.

14. REFERENCES

[A] R. Annicchiarico: C++ Support in DCE RPC — Functional Overview. OSF DCE SIG RFC 48.0; August, 1993.

[DLAR] P. Dasgupta, R. LeBlanc Jr., M. Ahamad, U. Ramachandran: The Clouds Distributed Operating System. IEEE Computer (to appear).

[HP] HP DCE Sample Applications Overview, Hewlett-Packard Document NSA-92-024, 1992.

[IPPG] InterViews Plus Programmer's Guide. Hewlett-Packard, 1992.

[KM] T. Korson, J. McGregor: Understanding Object-Oriented: A Unifying Paradigm. CACM Vol. 33, No. 9, September, 1990, p. 40-60.

[LK] W. Leddy, A. Khanna: DCE++: A C++ API for DCE. HaL document #030-00209; March 31, 1993, Draft 0.3.

[LVC] M. Linton, J. Vlissides, P. Calder: Composing User Interfaces With InterViews. IEEE Computer, Vol 22, No. 2, February 1989.

[M] M. Mock: DCE++: Distributing C++-Objects using OSF DCE. Proceedings of the International Workshop OSF DCE; Karlsruhe, Germany; October 1993.

[NWM] J. Nicol, C. Wilkes, F. Manola: Object Orientation in Heterogeneous Distributed Computing Systems. IEEE Computer, Vol. 26 No. 6, June 1993, p. 57-67.

[OMG] Object Management Group: Common Object Request Broker Architecture and Specification. Document Number 91.12.1, Revision 1.1.

[OSF] Open Software Foundation: OSF DCE Application Development Guide.

[P] G. Parrington: Reliable Distributed Programming in C++: the Arjuna Approach. USENIX C++ 1990.

[RTHBHJ] R. Raj, E. Tempero, H. Levy, A. Black, N. Hutchinson, E. Jul: Emerald: A General-Purpose Programming Language. Software Practical Experiences, Vol. 21 No. 1, January, 1991.

[S] A. Schill: OSI, ODP and Distributed Applications: Towards and Integrated Approach. IEEE Global Telecommunications Conference and Exhibition, 1991. p. 638-642.

[SDP] S. Shrivastava, G. Dixon, G. Parrington: An Overview of the Arjuna Distributed Programming System. IEEE Software, Vol. 8, No. 1, January, 1991.

An earlier version of this paper was published in the proceedings of the October, 1993 DCE Workshop.

AUTHOR'S ADDRESS

John Dilley
Mailstop 43UA
Hewlett-Packard
19410 Homestead Road
Cupertino, CA 95124-9810
USA

Internet email: jad@nsa.hp.com
Telephone: +1-408-447-2419