OODCE: A C++ Framework for the OSF Distributed Computing Environment

John Dilley
Hewlett-Packard Laboratories

Abstract

This paper presents a method for developing object-oriented distributed applications using the C++ and DCE technologies. The core of this package is a DCE IDL-to-C++ compiler and a set of C++ classes providing easy access to DCE functionality.

Using this approach we were able to develop more object-oriented distributed applications, and saw a significant decrease in application code size. This contributed to an increase in developer productivity and code maintainability.

1 Introduction

The Open Software Foundation's Distributed Computing Environment (OSF DCE) is an environment for development of distributed applications. DCE consists of a programming environment and a set of services which support distributed computing. The DCE facilities include:

- Communications—The Remote Procedure Call (RPC) mechanism allows processes to communicate across a network using procedure call/return semantics.

  The RPC component specifies a DCE Interface Definition Language (IDL), a higher-level language used to define the data types and remote procedures associated with an interface. The DCE IDL compiler converts IDL specifications into RPC communication stubs. An application developer must implement the server side routines called by the stubs (the DCE remote procedure "manager functions"). The client stubs provide transparent remote access to the implementation; a call to the server through the client stub looks just like an ordinary procedure call. DCE also defines an exception mechanism, used by RPC to notify client applications of communication or application faults.
- POSIX Threads—A multi-tasking package. Pthreads allow a single process to have many threads of control, possibly initiating or serving many RPCs concurrently.
- Naming—The Cell Directory Service (CDS) is a distributed directory service. CDS allows DCE clients to locate servers at run-time in a host-independent manner.
- Security—An implementation of MIT's Kerberos. DCE Security provides authentication and authorization for RPC calls so clients and servers can be sure of each other's identity, so data passed via RPC may be encrypted, and so servers can control access to their resources.
- File System—The Distributed File System (DFS) provides a shared global file system visible to all systems in the cell.
- Time—The Distributed Time Service (DTS) keeps machine clocks synchronized across the network.

DCE services can be accessed programmatically through an application programming interface (API) defined in the DCE AES [1]. They are also accessible through administrative commands, and in some cases through user commands.

1.1 DCE Benefits

Use of the DCE provides application developers with many benefits as compared with network programming using lower-level primitives. Some of the benefits are:

- DCE provides a powerful programming model: application developers can use the familiar procedure call/return model to communicate with remote entities.
- IDL makes network programming simpler: there is no need to write the code that marshals RPC parameters into their network representation. Compiler-generated communication stubs handle all marshaling and unmarshaling for RPC.
- Integration: client applications can use the CDS to provide location-independence; the security service provides secure RPC without requiring developers to be experts in security; integration of threads and exceptions allows easy use of these facilities in RPC applications.

- IDL requires a formal definition of the interface between network objects. The implementation of an object can be in one of a number of languages. Having a formal interface between client and server allows the implementation to be modified independently of the interface; this helps applications to evolve gradually.
- DCE has an implicit object model beyond the pure procedural model exposed by RPC. The object model includes the ability to reference individual objects in the distributed system, and to associate objects with their persistent state, or with a particular implementation in a server.

1.2 DCE Challenges

Along with the benefits, there are a number of challenges posed by DCE: programming complexity often comes with a powerful environment like this. In this case, the DCE services have complex interfaces that a new user needs to learn about before developing DCE applications. For example, there are over 400 DCE interface routines, but only 12 of them are needed in a minimal RPC application. Around 70 DCE routines are used in a more representative application that uses RPC, naming, threads, and security.

Additionally, many of the DCE calls made within an application are similar across applications. This boilerplate code makes applications redundant without adding value, and increases application learning time and long-term maintenance cost.

1.3 Why DCE in C++

The benefits of building systems using object-oriented techniques have been widely discussed in the literature. Korson and McGregor [2] provide a thorough examination of the concepts of object-oriented development. Nicol, et al. [3] explore use of object orientation in distributed systems, and discuss some current distributed object-based systems including the existing DCE object model and OMG's CORBA [4] distributed object system.

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

- C++ provides object abstraction and data encapsulation, allowing developers to work at a higher level. C++ also supports interface and code reuse.
- C++ is becoming the prevalent language for object-based systems development. Increasing numbers of C++ class libraries and development tools are becoming available.
- C++ has a clean and natural interface to C, and therefore to the existing DCE implementation.
- The C++ object model is consistent with the object model used in DCE.

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

1.4 Project Goals and Requirements

This project set out with three main goals:

1. Simplify DCE application development.
2. Allow creation of DCE applications in C++.
3. Maintain full interoperability with C-based DCE applications.

In order to simplify DCE application development, this project defined a C++ class library that hides DCE's complexity from application developers and a compiler that converts DCE interface definitions into corresponding C++ client and server classes.

The class library provides default behavior for DCE functions such as security and name space registration. This eases application development 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 subclassing and providing a custom implementation. Providing a suitable default implementation reduces the learning time of DCE, and will increase the consistency of distributed applications if they use the same behavior.

Following is a detailed set of requirements for the construction of a class library for DCE.

- Usable—The class library should be easier to learn and use than the underlying technology.
- Standard—The library should build on existing DCE components. It should not require the replacement of any standard DCE component.
- Interoperable—Applications developed using C++ must be able to interoperate with ordinary DCE applications developed in C or other languages supported by DCE.
- Performance—The class library should allow an application to perform within 5% of a C-based DCE application with equivalent features.
- Coverage—While making application development simpler, the library should also provide the ability for the developer to access the full functionality of DCE.

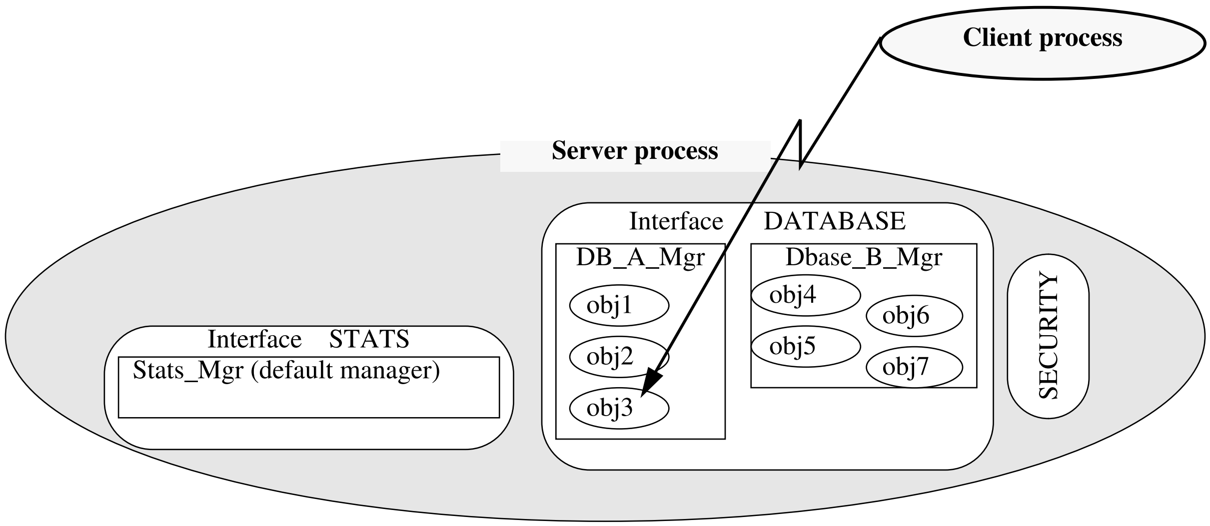


FIGURE 1. DCE Object Model

- 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.
- Customizable—The developer should be able to derive from library classes to customize their behavior. This is important for classes that provide policy for interacting with DCE components such as naming and security.
- Extensible—The library should allow developers to derive new classes from those provided to support new capabilities.

2 DCE Object Model

DCE RPC is, obviously, procedure-oriented, but DCE does define a model for object-based application development. The DCE object model is a conceptual one, in which DCE entities are thought of and manipulated as “objects”, even though they are not necessarily implemented in an object-oriented language. The DCE C++ class library exposes this model and provides an object-oriented implementation for it.

DCE Definitions

DCE *servers* provide the services of one or more object implementations 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* (written in IDL) define a set of data types and remote operations (procedures). An interface defines how DCE objects can be accessed. The interface is essentially a contract between the user of an object and the object’s implementation; it defines the public signature of the DCE object.

A DCE *object* is a logical entity contained within a DCE server; each DCE server contains one or more DCE objects. Each DCE object *exports* (supports) one or more interfaces. Clients can access DCE objects only through their supported interfaces.

A server can export multiple DCE interfaces. The server in Figure 1 exports three interfaces:

- Its main interface is a database interface. The server contains a set of objects that export this interface. Each object corresponds to a particular entity (tuple or cell) in a database.
- The server also exports a statistical interface to allow clients to retrieve usage statistics about the objects supported by the database interface.
- Finally, the server exports a security interface, used by the database manager functions to control client access to the database based upon client authorization information (implemented in terms of Access Control Lists or ACLs). This interface needs to be exported to allow external programs the ability to view and modify the ACLs.

In addition to supporting multiple interfaces, the server in Figure 1 also has multiple *implementation types* within its database interface. Implementation types A and B each implement the common database interface, but each for a different type of underlying database. The manager functions all have the same signatures and semantics, defined by the IDL, but can have different (private) implementations.

DCE C++ Object Model

We considered two possible approaches to creating a class library for DCE. One approach 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, which we chose, was to create a set of classes that build an abstract model of the data and functions needed by a DCE application. These classes were not patterned after data types and procedures, but rather based upon the set of *tasks* or *responsibilities* of the DCE model. The implementations of these classes use the underlying DCE facilities, but do so using an interface independent of those data types; in many cases users need not be aware of details of the underlying data types.

With the wrapper class approach, each underlying interface is typically mapped into a method call, providing little simplification over the current 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 end user code, and possibly to the wrapper classes themselves.

Wrapper classes are still essential in many instances to encapsulate basic data types, even when using a task-based model. One example is the encapsulation of native DCE data types, such as **uuid_t**, providing the developer a convenient C++ interface to those data types and allowing the use of conversion operators and constructors.

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 OODCE Model

The OODCE model preserves the concepts of the DCE object model while providing convenient 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.

3.1 IDL to C++ (OODCE) Compiler

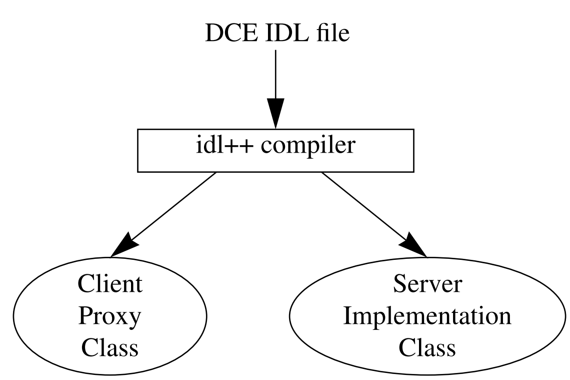
The IDL-to-C++ compiler, **idl++**, translates a DCE interface definition into interface-specific client and server classes described below. The compiler also generates the regular DCE C header file and client and server stubs used by the RPC mechanism. Since OODCE uses stock DCE client and server stubs, interoperation with DCE applications is assured.

The C++ classes created for OODCE are shown in Figure 2 and described in detail in Section 4.3. Briefly, the client proxy class includes the member functions that allow C++ method calls to invoke the RPCs defined in the IDL file. Application developers must provide the methods of the server implementation class; these methods define the application's behavior, similar to how a C developer implements DCE manager routines.

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 [5]). Any data types defined in the IDL file are used by the application developer as C data types.

The methods of client proxy objects make a remote procedure call to the equivalent method in the server

FIGURE 2. IDL++ Generated Classes



implementation object. Proxy objects provide the illusion that the client is calling the remote method directly. In this way, developers can access remote objects using the same programming model they use for making local method calls.

Proxy objects can locate (bind to) implementation objects using the directory name space (CDS), a remote host's endpoint map, an object reference, or via a DCE binding handle. The server's location is determined by how the proxy object is constructed, and via member functions on the proxy object.

3.2 Object References

While it is not currently possible to include C++ objects as arguments in RPCs, it is possible to pass references to C++ (DCE) objects. These DCE object references can allow multiple client proxy objects to refer to the same object instance, and can allow an object to invoke methods on its caller. Object references can also be made persistent and written to stable storage for later reincarnation as proxy objects.

3.3 Object Activation

Instances of DCE server implementation objects can be created at server start-up time to handle client requests. However, if the server will be managing a large number of objects, it is advantageous for the instances to be dynamically activated when requests come in for them. OODCE provides an activation mechanism which allows server developers to define an activation method for their objects.

When a request comes in for an object that is not currently active, this application-defined activation method is called to create and initialize the object. This may involve reading the object's state from persistent storage. A corresponding deactivation method can be defined to allow objects to quiesce themselves after a period of inactivity.

4 The OODCE Class Library

4.1 DCE Framework Classes

DCE framework classes are responsible for providing the DCE object model 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 overriding 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 key framework classes are discussed in the following sections. Refer to Figure 3 for a pictorial representation of the classes present in the library.

DCEServer

This class implements the portion of a DCE server that interacts with the DCE environment. A single global instance of the server class manages all the objects and interfaces exported by that server. The DCEServer 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 declared to the server by registering a policy object with the server.

DCEInterfaceMgr

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 functions), and optional type and object identifiers for that object instance. The information in the interface manager class is registered with the DCE runtime by the server class.

Derived classes of DCEInterfaceMgr, 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.

DCEInterface

This is the client abstract base class. It encapsulates the common client functions for specifying a remote object with which to communicate. The DCEInterface class provides the ability to control [re]binding, security principal information, and authentication policy.

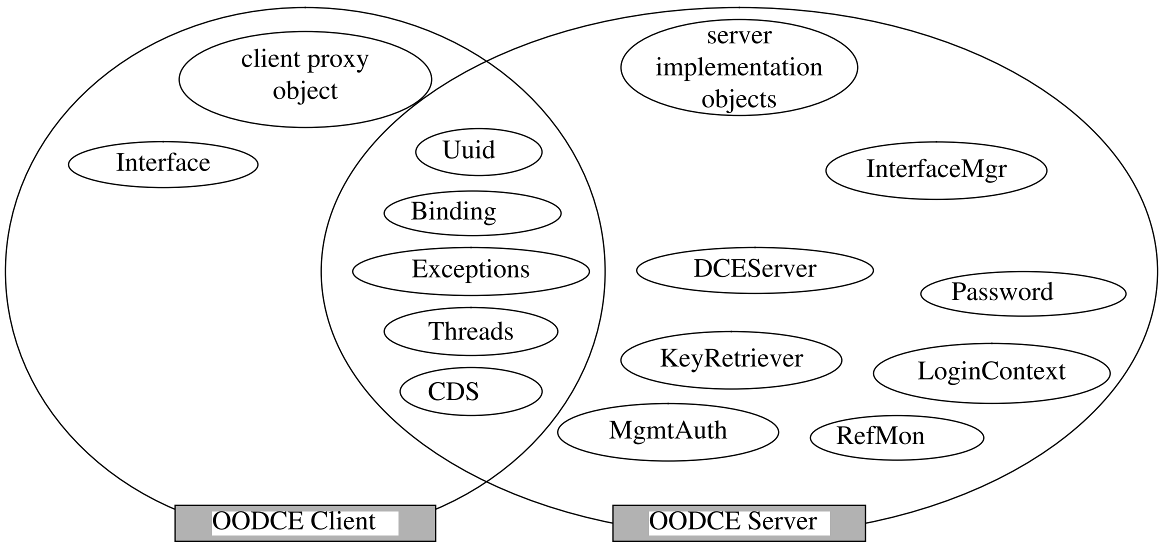


FIGURE 3. Class Library Components

Type conversions are provided from DCEInterface to a DCE client interface handle or to a string. Derived classes of DCEInterface, generated by **idl++**, provide member functions to access the remote operations. These classes are referred to as client proxy classes.

4.2 Utility Classes

The utility classes encapsulate the basic DCE data types to make their use more convenient in C++, allowing convenient construction and use of these data types. Many classes 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. Some examples of the utility classes are DCEUuid, DCEBinding, and DCEPassword.

Another key purpose of these classes is to allow customization of the behavior of the class library. The OODCE framework classes use these utility classes through their defined class interfaces. This allows developers to create derived classes that obey the class interface, yet provide custom behavior.

Security

The class library includes an interface for defining an Access Control List (ACL) manager and ACL storage database. In addition, classes are provided to interface with DCE principals and passwords, to establish and refresh a DCE login context, and to retrieve security credentials from the RPC caller.

Naming

The DCENsi classes model objects in the directory namespace, including CDS names, NSI server entries, and RPC groups and profiles. These classes can be passed to framework classes to identify entries in the name space to use. Applications can also use these classes directly to modify entries in the directory name space.

Threads

The Pthread classes provide C++ access to DCE threads. The classes provide the ability to model pthread mutexes and condition variables, to create and start new threads, to set thread attributes, and to interact with thread specific storage.

4.3 Generated Classes

The OODCE generated classes are created specific for a given interface by the **idl++** compiler. This section describes their structure and function in greater detail.

Client Proxy Class

The client-side proxy class is derived from the abstract base class DCEInterface. The proxy class has methods corresponding to the operations (remote procedures) defined in the IDL interface. The client remote operation methods handle the location of the 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.

Server Implementation Abstract Class

For the server, **idl++** generates an abstract class derived from DCEInterfaceMgr. 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.

Server Implementation Concrete Class

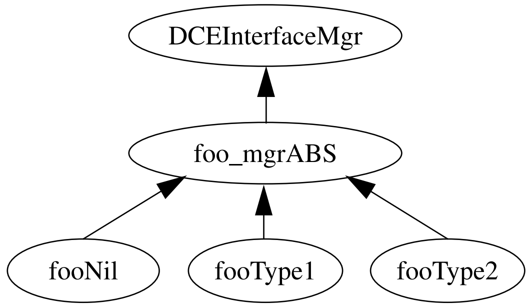
Instantiable (concrete) classes derived from the implementation abstract class must implement all of the member functions defined in the abstract class to provide the operational characteristics of the server. Each of these instantiable classes is of a distinct implementation type (i.e., has an associated manager type UUID). 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. Figure 4 shows the inheritance tree generated by the compilation of an interface foo. The classes **fooType1** and **fooType2** are developer-defined classes; **fooNil** is the concrete class generated by the compiler.

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 define 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.

FIGURE 4. Server Class Hierarchy



4.4 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 proxy classes, initialize the GUI, register the proxy classes with the GUI, and then await user input. Remote method calls would 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.

5 OODCE Exception Model

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 [6]. Use of exceptions can reduce application source code size since checking and passing through of error return codes is no longer necessary.

The area of exceptions required quite a bit of work in the design of the class library. One goal of this project was to provide a natural integration between the DCE exception mechanism and C++ exceptions. The C++ exception implementation is far more powerful and better integrated with the language than the DCE exception mechanism. Therefore C++ exceptions were 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 by the C language DCE code cannot be allowed to pass through blocks of C++ code. The DCE exception facility uses **setjmp** and **longjmp**; **longjmp** causes control to pass from one stack

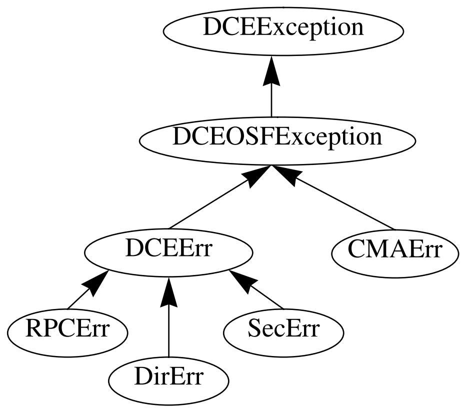


FIGURE 5. Exception Class Hierarchy

frame to another frame, possibly much earlier in the call stack. The trouble is that C++ destructors for automatic (stack) and temporary variables aren't invoked. Allowing a DCE exception to skip over this code can therefore cause consistency problems and memory leaks in the C++ application.

To model DCE exceptions in C++, an abstract base class DCEOSFException was defined; this class specifies the common behavior for all exceptions. Two more base classes, DCEErr and DCECmaErr, were derived from DCEOSFException to model the distinct types of exceptions that can be raised by the DCE and CMA (thread and exception) subsystems. The DCEErr class was further subdivided into RPC, security, and directory service exceptions. Then each specific exception that can be raised by DCE or CMA was modeled as a subclass derived from one of these base classes. Each exception class can be converted into a string (via **operator char*()**) in order to print a description of the exception.

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. This is inconsistent with the more powerful exception mechanism available in C++.

5.1 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.

5.2 Client Stubs

The C++ proxy 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 **TRY/CATCH** clause. If an exception is caught in the C++ stub, it is rethrown in C++ as the corresponding exception subclass.

6 Application Development

We ported a set of C sample applications from the HP DCE Developer's Toolkit and created a set of new applications in order to experiment with OODCE. The purpose of porting existing applications was to determine the benefits of the OODCE approach. The new applications were necessary to explore the capabilities available in OODCE. Overall we found creation of OODCE applications to be significantly easier than developing ordinary DCE applications.

Our application development environment was HP-UX 8.0 and 9.0, with HP DCE/9000 release 1.1 and 1.2, and HP C++ version 3.20 and 3.40.

6.1 Example code

This section contains sample code from a simple DCE application. This example illustrates what a minimal OODCE application looks like. What follows is fundamentally all the code the application developer needs to write.

IDL file

```
[uuid(...), version(1.0)]
interface sleeper
{
 void Sleep([in] handle_t h,
                  [in] long time);
}
```

Client Main

```
main(int, char** argv)
{
 try {
   sleeper_1_0 sleepClient(argv[1],
                              "ncadg_ip_udp");
   long sleep_time;
   sleep_time = atoi(argv[2]);

   // invoke the remote procedure
   sleepClient.Sleep(sleep_time);
 } catch (DCEErr& exc) {
    cout << "Caught DCE exception" << (char*)exc;
    exit(1);
 }
}
```

In the above code the constructor for **sleeper_1_0** takes a host name and protocol sequence (UDP/IP) to instruct the client how to bind to the server object. The constructor could also have taken a name in the Cell Directory Service (CDS) or an object reference.

Server Main

```
void main()
{
 try {
   // Construct the Sleeper object
   sleeper_1_0_Mgr sleeper;

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

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

The server constructs an instance of a manager (implementation) object and registers it with the global server object. When **main** exits, the global server object is destructed, at which point it unregisters the interfaces it registered at start-up time.

Manager Code

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

The only implementation required is for the single remote operation, **Sleep**.

6.2 Performance Analysis

One of the requirements for this project was to be within 5% of the performance of an equivalent DCE application. Early experiments have shown a minimal degradation in performance, indicating we are within the 5% figure. The following performance tests looked at the time to construct a binding to the remote server, the time for the first RPC, the average time for ten (10) subsequent RPC calls, and the total time for all these calls. The tables below present the mean time in milliseconds and the 95% confidence interval for the mean. The tests were run on HP 9000 Series 720 workstations configured with 64MB RAM.

In the first test, clients using both DCE and OODCE make a simple RPC request (with a single integer parameter) to an OODCE server.

TABLE 1. RPC to OODCE server

	DCE	OODCE
Binding	42.9 ± 4.2	52.7 ± 2.5
First call	14.2 ± 2.4	3.75 ± 0.24
Other calls	3.19 ± 0.36	3.20 ± 0.45
Total	86.6 ± 4.0	88.6 ± 3.1

The table above shows that the binding time for the OODCE client is longer than for the DCE client. This is due to the type checking being done when constructing a client proxy object: the construction of OODCE objects is type safe, whereas the construction of DCE binding handles is not type safe. Note also that DCE's first call time is much greater than for OODCE. This is because OODCE type checking causes the object to be fully bound to the server, whereas in DCE the resolution of the binding handle happens in the first call.

In the second test, DCE and OODCE clients both contacted a DCE server (in C).This test showed similar performance characteristics as the first test. Additional tests were conducted with client and server on different systems. The results from these tests showed lower call times than the local case, due to the client and server applications being able to run concurrently.

6.3 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 a GUI based on the C++ InterViews user interface class library [7]. 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.

For example, the code size (non-comment source statements) of the OODCE sample applications is about 30% of the corresponding C samples (though the applications are not equivalent: the OODCE sample applications have more capabilities than the corresponding DCE applications).

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 proxy class was quite natural.

7 Recommendations

We have found it is possible and useful to develop object-oriented distributed applications with C++ and DCE, but there are still some technical issues with doing so. Included below is a set of recommendations intended to smooth the road for others developing object applications using DCE and C++. They are mainly aimed at DCE vendors.

- 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. If not, the application must either wrap calls to those subsystems or ensure that only a single thread will call them.
- The availability of thread-aware distributed debuggers will also greatly aid in the creation of (working) DCE applications. (This has nothing to do with C++, but will help C++ developers).
- The DCE and C++ exception mechanisms must be reconciled. One alternative is for DCE exceptions to use the same basic mechanism as the C++ exception facility. This would allow consistent exceptions to thrown from one language to the other.
- Users would benefit from a standard framework for developing DCE-based C++ applications. Cooperation between OSF and DCE vendors and standardization on a single approach can help to bring this about. HP intends to work the OSF DCE Special Interest Group (SIG) Object-Orientation Working Group to help make this happen.

8 Related Work

There are many distributed object systems documented in the literature with a variety of goals. The Arjuna system [8][9] focuses on fault tolerance and persistence using a custom RPC mechanism built atop an existing kernel. The Clouds project [10] built a distributed, object-based operating system using a custom microkernel and remote object invocation. Emerald [11] defines a language that uses an object-thread approach to provide distributed object communication. The Spring system [12] provides an object-based distributed system built atop a custom microkernel with a fast, secure object-based IPC mechanism. Schill [13] 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.

The above are primarily research projects exploring the operation of distributed object systems using custom operating systems or messaging systems. By contrast, our work focuses on integrating C++ with the existing DCE system infrastructure, and on simplifying the use of the existing DCE object model. We do this by providing an extension to the DCE while maintaining compatibility and interoperability with existing C-based DCE systems.

OMG CORBA [4] will eventually address creating distributed object systems in C++; some implementations may run on top of DCE. OMG's IDL provides for interface inheritance, which DCE IDL is currently lacking, and provides a more language-neutral syntax for interface definition. The CORBA runtime environment provides a richer set of object invocation and object passing capabilities than does the DCE environment. However, CORBA today lacks several of the features provided with DCE, such as standard security and naming integrated with the RPC facility, interoperability, and has less distributed computing support in the area of operation semantics and data types supported.

We suggest that our work might assist in the migration of applications from DCE to CORBA by providing an intermediate C++-based distributed object system until CORBA implementations are widely available. Migrating from OODCE to CORBA should be easier than migrating from C or from C++ using explicit DCE calls, because the direct use of DCE is unnecessary. 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 four other current C++-based DCE object systems. One is the DCE++ system proposed by HaL Computers [14]. 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 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 [15]. 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 proposal has been published by DEC [16]. This DCE RFC describes a mapping from IDL to C++ through extensions to DCE IDL and 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 by 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 also 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). However, this proposal does not address DCE usability by providing a class library framework as ours does.

Finally, Citibank made public a set of classes that make DCE development easier by encapsulating many of the DCE API calls [17]. The Citibank offering defines a new data definition language (YADDL) and a compiler that converts YADDL definitions into C++ classes and DCE IDL definitions. These C++ classes can then be passed in RPCs, like any regular IDL data types can be. In this case the entire object instance is serialized and transmitted; there is no notion of private object state (implementation details, for example) that should not be transmitted.

9 Conclusions

Object-oriented design and development provides a convenient, natural model for distributed systems development—applications can be cleanly modeled as collections objects with remotely callable methods. In specific, the modeling of DCE in C++ has provided significant benefits over the regular DCE interface: the class library and compiler described here significantly reduce the burden of creating DCE applications, and allow creating object-based DCE systems using C++. Most of these benefits could be obtained with an equivalent C library implementation, but the C++ implementation allowed us to expose the DCE object model more naturally. To summarize, the benefits we experienced after use of this technology were:

- Having powerful abstractions on top of DCE allowed us to concentrate on developing the application, not writing code to interface with DCE.
- 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.
- Application development and debugging time were shortened because basic DCE calls are encapsulated in an already-tested library.
- Having sensible defaults for many DCE values prevented the need for redundant code, allowing code reuse; also having defaults prevents users new to DCE from having to make choices they may not be prepared to make.
- The library provides full coverage of DCE, obviating 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.
- 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.
- 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.

- OODCE is much easier to learn and use than DCE, even for developers who were new to C++. It was easier for them to learn enough C++ to use OODCE than to write a regular DCE application.
- The C++ exception model is more powerful and more 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.
- C++ is a more powerful language for describing object interfaces and implementations. C++ allows separation of interface from implementation, which is necessary in a distributed environment.

10 Acknowledgments

The idea, initial design, and prototype implementation of this project were mainly due to Jeff Morgan. We also wish to thank Deborah Caswell and Bob Fraley for their advice and contributions to this product, and Amy Arnold, Serena Chang, Jack Danahy, Linda Donovan, Mickey Gittler, John Griffith, Mike Luo, Luis Maldonado, and John Stodieck for their effort in making this system into an HP product. Thanks also go to those who have reviewed earlier drafts of this document, in particular to Rich Friedrich, Joe Martinka, Christopher Mayo, and the USENIX draft reviewers.

11 References

[1] Open Software Foundation: *OSF DCE Application Environment Specification*. Open Software Foundation, 1992.

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

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

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

[5] Open Software Foundation: *OSF DCE Application Development Guide*.Open Software Foundation, 1992.

[6] M. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. May, 1991.

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

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

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

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

[11] 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.

[12] G. Hamilton, P. Kougiouris: *The Spring nucleus: A microkernel for objects*. 1993 Summer USENIX, p 147-159.

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

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

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

[16] R. Annicchiarico, J. Harrow, R. Viveney: *C++ Support in DCE RPC—Functional Overview.* OSF DCE RFC 48.1, April, 1994.

[17] L. Poleshuck: *Objtran Programmer's Guide*. Citibank Distributed Processing Technology group, December 3, 1993.

Earlier versions of this work were published in the proceedings of the October, 1993 DCE Workshop in Karlsruhe, Germany, and as OSF DCE RFC 49.0.

Biography

John Dilley is a distributed systems architect with Hewlett-Packard Laboratories. His interests include architecture and construction of distributed systems, object location (naming) and distributed directory services, and object-oriented design and development. Mr. Dilley received Bachelor of Science degrees in Mathematics and Computer Science from Purdue University in 1984, and a Master of Science degree in Computer Science from Purdue in 1985. Since then he has been working for Hewlett-Packard in network and systems architecture groups. He was a member of the team that designed the original OODCE prototype.