Object Interconnections

Introduction to Distributed Object Computing (Column 1)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@ch.hp.com
Hewlett-Packard Company
Chelmsford, MA 01824

This column appeared in the January 1995 issue of the C++ Report magazine.

1 Introduction

Welcome to the first edition of our new Object Interconnections column concerning distributed object computing (DOC) and C++. In this column, we will explore a wide range of topics related to DOC. Our goal is to de-mystify the terminology and dispel the hype surrounding DOC. In place of hype, we will focus on object-oriented principles, methods, and tools that are emerging to support DOC using C++. We plan to investigate and describe various tools and environments that are available commercially. In addition, we will discuss detailed design and implementation problems that arise when using C++ to create DOC solutions. The field of DOC is already rather broad and is still growing rapidly. Therefore, we will have plenty of material to cover in the coming months. If there's any topic in particular that you'd like us to cover, please send us email at `object_connect@ch.hp.com`.

It has been claimed that distributed computing can improve

- collaboration through connectivity and interworking;
- performance through parallel processing;
- reliability and availability through replication;
- scalability and portability through modularity;
- extensibility through dynamic configuration and reconfiguration;
- cost effectiveness through resource sharing and open systems.

Our experiences and the experiences of others have shown that distributed computing can indeed offer these benefits when applied properly. However, developing distributed applications whose components collaborate efficiently, reliably, transparently, and scalably is a complex task. Much of this complexity arises from limitations with conventional tools and techniques used to develop distributed application software. Many standard network programming mechanisms (such as BSD sockets and Windows NT named pipes) and reusable component libraries (such as Sun RPC) lack type-safe, portable, reentrant, and extensible interfaces. For example, both sockets and named pipes identify endpoints of communication using weakly-typed I/O handles. These handles increase the potential for subtle run-time errors since compilers can't detect type mismatches at compile-time.

Another source of development complexity arises from the widespread use of functional decomposition. Many distributed applications are developed using functional decomposition techniques that result in non-extensible system architectures. This problem is exacerbated by the fact that the source code examples in popular network programming textbooks are based on functional-oriented design and implementation techniques.

So, in this world full of hollow buzzwords and slick marketing hype, it is natural to ask the question "what does object-oriented technology contribute to the domain of distributed computing?" The short answer to this question is that object-oriented technology provides distributed computing with many of the same benefits (such as encapsulation, reuse, portability, and extensibility) as it does for nondistributed computing.

In fact, it is often more natural to utilize object-oriented techniques in the domain of distributed computing than it is for non-distributed computing. This is due to the inherently decentralized nature of distributed computing. In conventional non-distributed applications, there is often a temptation to sacrifice abstraction and modularity for a perceived increase in performance. For example, many programmers use global variables or access fields in structures directly to avoid the overhead of passing parameters and calling functions, respectively.

In distributed computing, however, performance optimizations based on direct access to global resources are extremely difficult to develop and scale. Research and development on operating system support for distributed shared memory, for example, is not yet ready for large-scale system deployment. Therefore, most distributed applications interoperate by passing messages. There are many variations on this message passing theme (e.g., RPC, remote event queues, bytestream communication, etc.). However, it doesn't require much of a stretch of the imagination to recognize that message passing in distributed computing is very similar to method invocation on an object in object-oriented programming.

With this observation in mind, let's discuss several of the key features of DOC:

Providing many of the same enhancements to procedural RPC toolkits that object-oriented languages provide to conventional procedural programming languages: Distributed object computing frameworks enhance procedural RPC toolkits (such as Sun RPC and the OSF DCE) by supporting object-oriented language features. These features include encapsulation, interface inheritance, parameterized types, and object-based exception handling.

Encapsulation promotes the separation of interface from implementation. This separation is crucial for developing highly extensible architectures that decouple reusable application-independent mechanisms from application-specific policies. Interface inheritance and parameterized types promote reuse and emphasize commonality in a design. Object-based exception handling often simplifies program logic by decoupling error-handling code from normal application processing.

Enabling interworking between applications at higher levels of abstraction: Distributed applications have traditionally been developed using relatively low-level mechanisms. Common mechanisms include the TCP/IP protocol, the socket transport layer programming interface, and the `select` event demultiplexing system call. These low-level mechanisms provide applications with reliable, untyped, point-to-point bytestream services. In general, these services are optimized for performance, rather than ease of programming, reliability, portability, flexibility, or extensibility.

A primary objective of DOC is to enable developers to program distributed applications using familiar techniques such as method calls on objects. Ideally, accessing the services of a remote object should be as simple as calling a method on that object. For example, consider an object `obj` that provides a service `op` with arguments `arg1`, `arg2`, and `arg3` and a return value of type `reply`. We'd like our client applications to invoke `op`, pass it arguments, and obtain a reply by simply writing `reply r = obj->op (arg1, arg2, arg3)`.

A surprisingly large number of fairly complicated components must be developed to support remote method invocation on objects transparently. These components include directory name servers, object request brokers (ORBs), interface definition language compilers, object location and startup facilities, multi-threading facilities, and security mechanisms. In subsequent columns, we will define these terms and illustrate how they work together to solve real-world problems.

Providing a foundation for building higher-level mechanisms that facilitate the collaboration among services in distributed applications: Supporting transparent remote object method invocation is only the first step in the long journey into the realm of distributed object computing. An increasing number of distributed applications require more sophisticated collaboration mechanisms. These mechanisms include common object services such as global naming, event filtering, object migration, reliable group communication, transactional messaging, and quality of service facilities. More advanced tools will support electronic mail, visualization, collaborative work, and concurrent engineering.

When all these provisions of DOC are realized and standardized, we may very well finally see the long-awaited arrival of "plug and play" software components and "Software ICs." Object vendors will be able to market various implementations of industry-standard interfaces, and users will be able to mix and match those components, investing in the ones that they believe best fulfill their needs. Until that time, however, there is still quite a bit of work to do. Only recently have the very lowest levels of support for DOC, such as object request brokers, become commonplace in the market.

2 But What About C++?

So far, we've barely even mentioned C++. In future columns, we'll discuss ways in which C++ may be used to simplify distributed object computing. We believe that when used properly, C++ is well suited for the construction of both distributed object support systems and the object components themselves. C++ combines high-level abstractions with the efficiency of a low-level language like C. Many of the emerging frameworks and environments for distributed object computing are based on C++, due to its widespread availability and appeal. For example, commercial tools such as several CORBA ORBs, HP OODCE, and Network OLE, as well as freely-available software toolkits such as ILU from Xerox PARC and the ADAPTIVE Communication Environment (ACE), support object-based distributed programming using C++.

Certain C++ features are well-suited for programming distributed objects. For example, abstract base classes, pure virtual inheritance, virtual functions, and exception handling help to separate object interfaces from object implementations. However, the lack of other features in C++ increases the complexity of developing robust and concise distributed applications. For instance, support for garbage collection would greatly reduce memory management complexity. Likewise, `before` and `after` methods would enable greater control over the marshaling and demarshaling of parameters passed to remote method calls. In the coming months, we will discuss C++ language idioms that have been successfully used in practice to address certain C++ limitations.

3 Next Time

Over the next several months, our column will examine an extended example that compares and contrasts different ways to use C++ to program a representative distributed application from the domain of financial services. In these columns we'll compare several solutions for developing the client-side and server-side of this solution. These solutions will range from using the C language sockets network programming interface, to using C++ wrappers for sockets, to the use of distributed object computing frameworks (such as CORBA, Network OLE, and OODCE). The example will illustrate the various tradeoffs between efficiency, extensibility, and portability involved with each approach.

Electronic versions of these columns are available on-line at the following WWW URL:

`http://www.cs.wustl.edu/~schmidt/corba.html`

A brief overview of the topic in each column is provided below, sorted in chronological order:

1. "Modeling Distributed Object Applications," *C++ Report*, SIGS, Vol 7. No. 2, February 1995. This column describes the key features of DOC frameworks (such as CORBA, Network OLE, and OODCE) and explains how these frameworks address distributed application requirements (such as reliability, heterogeneity, location independence, security, and performance).

2. "Comparing Alternative Client Distributed Programming Techniques," *C++ Report*, SIGS, Vol. 7. No. 4, May 1995. This column examines an evaluates three different programming techniques for developing the client-side of a distributed application. These techniques include using the socket network programming interface, using C++ wrappers for sockets, to using a distributed object computing solution based on CORBA.

3. "Comparing Alternative Server Distributed Programming Techniques – the Reactive Model," *C++ Report*, SIGS, Vol 7. No 8. October 1995. This column examines and evaluates three techniques for developing the server-side a distributed application using a single-threaded, reactive model. These techniques include using the socket network programming interface, using C++ wrappers for sockets, to using CORBA.

4. "Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-per-Request Concurrency Model," *C++ Report*, SIGS, Vol 8. No 2. Febuary 1996. This column examines and evaluates four techniques for developing multi-threaded servers using the *thread-per-request* concurrency model. These techniques include using the socket network programming interface, using C++ wrappers for sockets, and using two multi-threaded versions of CORBA (MT Orbix and HP ORB Plus).

5. "Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-Pool Concurrency Model," *C++ Report*, SIGS, Vol 8. No 4. April 1996. This column examines and evaluates three techniques for developing multi-threaded servers using the *thread-pool* concurrency model. These techniques include using the socket network programming interface, using C++ wrappers for sockets, and using a multi-threaded version of CORBA (MT Orbix).

6. "Comparing Alternative Programming Techniques for Multi-threaded Servers – the Thread-per-Session Concurrency Model," *C++ Report*, SIGS, Vol 8. No 6. June 1996. This column examines and evaluates three techniques for developing multi-threaded servers using the *thread-per-session* concurrency model. These techniques include using the socket network programming interface, using C++ wrappers for sockets, and using multi-threaded version of CORBA (MT Orbix).

7. "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, SIGS, Vol 8. No 9. October 1996. This column examines *distributed callbacks* in CORBA and illustrates why they are useful for decoupling relationships between consumers and suppliers in object-oriented communication applications. The source code examples are based on the HP ORB Plus CORBA implementation.

8. "The OMG Events Service," *C++ Report*, SIGS, Vol 9. No 2. February 1997. This column outlines the roles of the key components in the OMG Events Service, examines the IDL interfaces of the Events Service components in detail, shows how to use it to build a flexible implementation of the distributed stock quoter system, and evaluates the strengths and weaknesses of the OMG Event Services model and its specification.

9. "'Overcoming Drawbacks with the COS Events Service," *C++ Reort*, SIGS, Vol. 9, No 6. June, 1997. This column describes techniques for overcoming drawbacks with the OMG Events Service. These techniques range from changing the COS Events Service specification, to changing implementations of the COS Events Service specification, as well as changing applications that use a COS Events Service implementation.

10. "Object Adapters: Concepts and Terminology," *C++ Report*, SIGS, Vol. 9, No 10. October, 1997. This column presents issues surrounding CORBA *Object Adapters* (OAs). It focuses on what Object Adapters are and describe their roles within a CORBA-based system. In addition, it begins an in-depth discussion of the new Portable Object Adapter (POA) specification that was recently adopted by the OMG.

11. "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, SIGS, Vol. 10, No 4. April, 1998. This column continues our presentation of the new OMG POA, focusing on POA fea-

tures that support *transient* and *persistent* CORBA objects.

12. "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, SIGS, Vol. 10, No 5. June, 1998. This column shows the definitions of the C++ servant classes and explains how the new POA specification separates the client-side stub hierarchy from the server-side skeleton hierarchy in order to facilitate collocation and ensure source-level portability.

13. "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, SIGS, Vol. 10, No 5. June, 1998. This column shows the definitions of the C++ servant classes and explains how the new POA specification separates the client-side stub hierarchy from the server-side skeleton hierarchy in order to facilitate collocation and ensure source-level portability.

14. "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, SIGS, Vol. 10, No 7, September, 1998. This column describes servant managers and default servants. Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. Default servants can process requests for an object if no other servant is available for it.

Please let us know if you have any comments or suggestions for improving these columns by writing us at `object_connect@cs.wustl.edu`.