Object Interconnections

Modeling Distributed Object Applications (Column 2)

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 February 1995 issue of the SIGS C++ Report magazine.

1 Introduction

In our first column, we discussed several promising benefits of using object-oriented (OO) technology and C++ to develop extensible, robust, portable, and efficient distributed application software. However, the OO marketplace is often long on promises and short on viable solutions. Therefore, we'd like to start moving the discussion from the abstract to the concrete. In our next several columns, we'll present an extended example that compares and contrasts different ways of using C++ and distributed object computing (DOC) to solve a representative distributed programming application.

Our example centers around a financial services system, whose distributed architecture is shown in Figure 1. We'll focus on a stock trading application that enables investment brokers to query stock prices, as well as buy shares of stock. As shown in Figure 1, the quote server that maintains the current stock prices is physically remote from brokers, who work in various geographically distributed sites. Therefore, our application must be developed to work efficiently, robustly, and securely across a variety of wide area (WAN) and local area (LAN) networks. We selected the stock trading application since the issues involved in analyzing, designing, and implementing it are remarkably similar to many other types of distributed applications.

Distributing application services among networks of computers offers many potential benefits. However, implementing robust, efficient, and extensible distributed applications is more complex than building stand-alone applications. A significant portion of this complexity is due to the fact that developers must consider new design alternatives and must acquire many new skills.

Realizing the potential benefits of DOC requires both strategic and tactical skills [1]. Strategic skills involve mastering design patterns [2] and architectural techniques that exist in the domain of distributed computing. This month's column focuses on the strategic issues underlying the distributed computing requirements and environment of our stock trading application. Tactical skills involve mastering

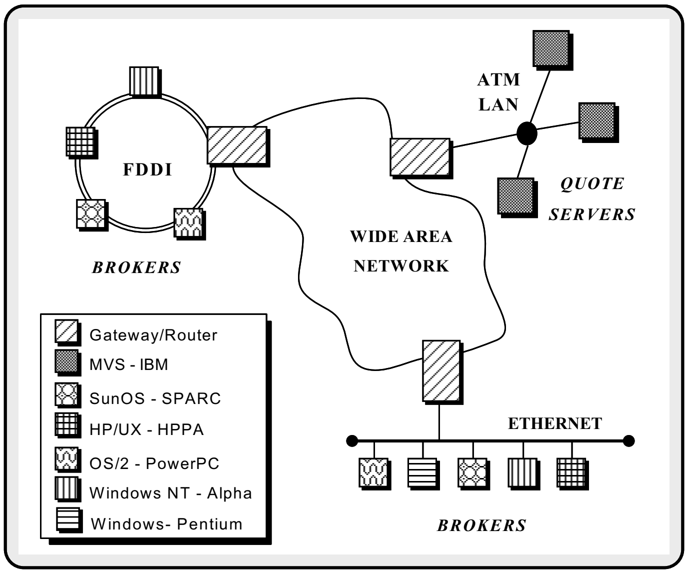

Figure 1: Distributed Architecture of Financial Services System

tools such as OO programming languages like C++ and OO DOC frameworks (such as CORBA, OODCE, and Network OLE). CORBA is an emerging standard for distributed object computing sponsored by the OMG [3], OODCE is a C++ framework for the OSF Distributed Computing Environment (DCE) [4], and Network OLE is Microsoft's technology for integrating distributed objects [5]. Subsequent columns will focus on tactical issues by evaluating detailed design and programming techniques used for the client-side and server-side of our example distributed application.

2 Application Distributed Computing Environment and Requirements

A good systems analysis begins by capturing the requirements of an application, and modeling the essential elements in its environment. This section discusses the distributed computing requirements of our stock trading application, as well as key characteristics of the distributed computing environment in which it operates. Along the way, we indicate how these requirements and environmental characteristics motivate and shape many reusable components and features found in DOC frameworks.

2.1 Distributed Computing Environment Characteristics

The distributed computing environment (shown in Figure 1) in which the stock trading application runs may be characterized as follows.

The broker clients and quote servers run on separate computers: These computers are joined by a heterogeneous internetwork of LANs and WANs (such as Ethernet, FDDI, and ATM). The network protocol stack connecting the distributed application components may be based on one of any WAN and LAN protocol families such as TCP/IP, X.25, ISO OSI, and Novell IPX/SPX. All these protocol families support end-to-end communication. However, they have subtly different characteristics and constraints that complicate software portability and interoperability. For example, TCP/IP is a bytestream transport protocol that ignores application message boundaries, whereas IPX/SPX maintains message boundaries [6].

Writing applications that operate transparently across different protocol stacks is often tedious and error-prone. Ideally, a DOC framework should shield applications from knowledge of these types of communication protocol-level details. OODCE is particularly strong in this area since it was designed to run over many protocol stacks. First-generation CORBA and Network OLE implementations, in contrast, have not addressed protocol stack transparency as vigorously. The CORBA 2.0 specification [3] requires exactly-once semantics for operations that return normally, at-most-once semantics for operations throwing an exception, and best-effort for oneway messages. However, the CORBA 2.0 specificiation does not define what an ORB is supposed to do when confronted with temporary lack of network/host resources indicated by transport layer flow control. Therefore, each vendor implements this differently, which complicates the development and deployment of highly portable and correct applications that run on multiple transport protocols.

Clients and servers may be heterogeneous end systems: These end systems may run on various hardware platforms (such as PA-RISC, Intel 80x86, DEC Alpha, SPARC, or the Power PC). Different hardware platforms possess instruction sets with either little-endian and big-endian byte orders. To improve application portability, DOC frameworks typically provide tools such as interface definition languages (IDLs) and IDL compilers. These tools generate code that automatically *marshals* and *demarshals* method parameters. This process converts binary data to and from, respectively, a format that is recognizable throughout a heterogeneous system of computers with instruction sets containing different byte orders.

The broker clients and quote servers may also run on different operating systems (such as variants of UNIX, Windows NT, OS/2, or MVS). These operating systems provide different sets of features (such as multi-threading, shared memory, and GUIs) and different system call interfaces (such as POSIX or Win32).

DOC frameworks provide different levels of support for shielding applications from differences in heterogeneous OS features and interfaces. Several DOC frameworks provide portable interfaces for certain OS-level features (such as the thread interface available with OODCE). However, other OS features (such as text file I/O, shared memory, and graphics) are often not standardized by DOC frameworks. Network OLE addresses OS heterogeneity by the focusing primarily on a relatively homogeneous OS platform (*i.e.,* the Win32 family of APIs [7]).

CORBA does not attempt to define a standard set of interfaces to OS features, ostensibly to give users the freedom to select their favorite OS tools. They may, however, define a standard set of interfaces for accessing DOC framework features, such as CORBA's Dynamic Invocation Interface (DII). The DII allows a client to directly access the request mechanisms provided by an Object Request Broker (ORB). Applications use the DII to dynamically issue requests to objects without requiring interface-specific stubs to be linked in. This allows clients to make use of services that are "discovered" at runtime.

It remains to be seen which of these different approaches to heterogeneity will be embraced by the marketplace.

2.2 Application Requirements for Distributed Computing

All DOC frameworks provide reusable components that simplify the development of distributed applications. These components elevate the level at which applications are designed and implemented. This enables application domain experts to focus on application-specific aspects of the problem (such as determining user-friendly interfaces for trading stocks), rather than wrestling with low-level communication details.

Our stock trading application has a number of distributed computing requirements. Many other types of distributed applications have similar requirements. Figure 2 shows some of the DOC components that we use in this section to motivate and explain our application's distributed computing requirements. Some of these components are specific to our application (such as the stock quoter, stock trader, and trading rules objects), and would typically be developed in-house. Other components are more generic (such as the printer, network time, location broker, authenticator, and heartbeat monitor objects), and are often provided by a DOC framework.

High Reliability: Distributed applications often require substantial effort to achieve levels of reliability equivalent to those expected from stand-alone applications. Detecting service failures in a stand-alone application is relatively

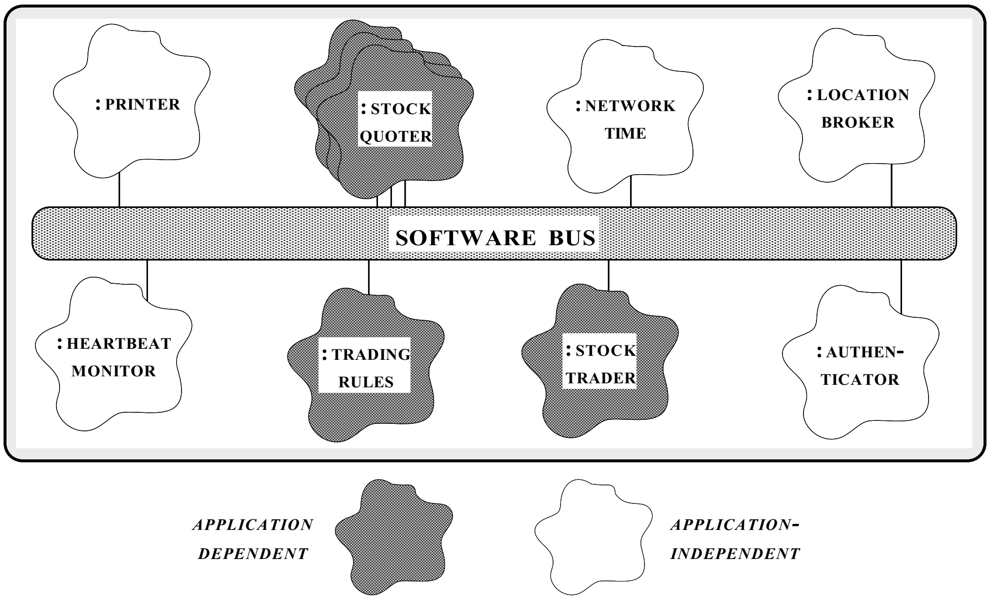

Figure 2: DOC Components

straightforward. For example, if a service fails gracefully, the caller is usually notified by a designated return value.

In contrast, detecting failures in distributed applications is often extremely complicated. For example, separate components in our stock trading application possess incomplete knowledge of global system state (such as the current price of a stock). By the time this information becomes available it may no longer be valid. This is a serious problem for distributed applications (such as an algorithmic trading system) that may exhibit transient inconsistencies due to caching on clients and/or servers.

Distributed transaction monitors, such as DCE-based Encina from Transarc, help to improve the reliability of a distributed application by ensuring that changes to system state occur atomically, consistently, and repeatably. In practice, our stock trading application would undoubtedly use some type of transaction service to ensure reliability. The OMG recently standardized on a Transaction Object Service [8], but few if any ORB vendors have yet to offer an implementation of it with their ORB products. Network OLE does not provide support for distributed transaction monitoring

Developing services that are resilient to independent host and network failures is also difficult. For instance, distributed applications are designed to tolerate some amount of variation in network transmission delay. Thus, a client may not detect an abnormal server termination until after valuable information is lost. Likewise, server responses may get lost in the network, causing clients to retransmit duplicate requests. Isis RDO [9] (and the follow on Orbix+Isis) is a DOC framework that supports reliable distributed object computing. It provides "fail-stop" semantics that ensure applications can distinguish reliably between request delays due to network congestion and lost messages due to host or network failures [10].

High Availability: Brokers earn their living by buying and selling stocks. Any time they are unable to access current stock prices or place trades their business suffers. Since loss of revenue due to downtime is generally unacceptable, it is essential that the stock trading system operate with high availability.

One technique for improving application availability is the replication of objects and services. For example, the stock quote object in Figure 2 is replicated to ensure a market data feed is always accessible (we've duplicated the `Stock Quoter` object in the figure to illustrate the replication).

Another technique for improving availability is to invoke applications under the control of a *heartbeat monitor*. This service detects and automatically reinvokes an application if it terminates unexpectedly. The Orbix+Isis ORB [11] supports transparent replication and reinvocation of CORBA objects.

Object Location and Selection: Traditional stand-alone applications generally identify their constituent services via memory addresses that point to objects and subroutines. In contrast, distributed applications require more elaborate mechanisms for naming and locating their remote services.

A traditional scheme for addressing remote services involves Internet (IP) host addresses and communication port numbers [6]. However, this mechanism is generally inadequate for large-scale distributed systems since it is difficult to administer in a portable and unambiguous manner. For example, "port 5000" does not necessarily refer to the same service on separate host machines configured by different vendors or by network administrators.

DOC frameworks generally provide *location brokers* that allow clients to access remote object services via higher-level names (rather than by low-level memory addresses or IP/port numbers), and *traders* that allow remote objects to be selected based on the desired characteristics of the services they provide.[1] Location brokers and traders simplify distributed system administration and promote more flexible and dynamic placement of services throughout a network by automating distributed object selection.

If a service has been replicated for improved reliability or availability, applications may use a location broker or trader to determine the most appropriate service provider. For example, the OODCE Cell Directory Service (CDS) can be considered to be a type of trader service. CDS supports the selection of remote services based upon a set of interfaces and objects associated with each service. In the stock quote application, the client may rely on a trader to help it locate a stock quote service that also happens to support the stock trading service attribute. Likewise, a broker might use the service attributes to print a postscript document by determining which printer(s) possess the postscript attribute and/or by determining which printer has the shortest queue. Using service attributes to select the shortest queue is an example of *load balancing*, described in the following paragraph.

Strongly-typed Interfaces: a major problem in a large-scale distributed environment is ensuring the consistency and integrity of the messages and methods shared senders and receivers. If the type signatures of these messages and methods become inconsistent, the reliability and correctness of a distributed system will be severely compromised. Tools that automate the process of ensuring this consistency are extremely important, in terms of (1) decreasing the potential for failure, (2) increasing the extensibility of the system by decoupling interfaces from implementations, and (3) providing a convenient means to document the behavior of the distributed system architecture.

Load Balancing: A bottleneck may result if many services are configured into the server-side of the application and/or too many clients simultaneously access these services. Conversely, configuring many services into the client-side may also result in a bottleneck since clients often execute on cheaper, less powerful host machines.

In general, it is difficult to determine the relative processing characteristics of application services *a priori* since workloads may vary over time. Therefore, load balancing techniques that enable developers to experiment with different application service partitioning and placement policies may be necessary. These techniques are supported by flexible distributed OS mechanisms that migrate services to other host machines either *statically* at installation-time or *dynamically* during run-time. Fully automated dynamic load balancing is still primarily a research topic [12], and few commercial DOC frameworks support it.

Security: Distributed applications are generally more vulnerable to security breaches than are stand-alone applications since there are more access points for an intruder to attack. For example, most shared-media networks (such as Ethernet, token ring, and FDDI) provide only limited built-in protection against cable tapping and promiscuous-mode "packet snooping" tools. Likewise, distributed applications must guard against a client or server masquerading as another entity in order to access unauthorized information.

DOC frameworks provide various forms of authentication (*e.g.,* Kerberos), authorization (*e.g.,* OODCE access control lists), and data security (*e.g.,* DES encryption). As of this writing, CORBA offers no standard security service, but technology submissions proposing a Security Object Service are due to the OMG Object Services Task Force by February 1995, with the selection of the standard to follow (hopefully) sometime within 1995.

Synchronous Communication and Threading: The communication between a broker client and a quote server may be performed synchronously. In other words, while a client application is querying the database for a stock quote, or waiting to place a trade, it may not need to perform additional processing. This requirement helps to simplify the client-side program structure since multi-threading and asynchronous I/O may be avoided. Both of these techniques tend to decrease portability and increase development and debugging effort.

It may be necessary to use multi-threading for the server-side of the stock trading application, however. Multi-threading helps to improve throughput and performance in circumstances where multiple clients make service requests simultaneously. We'll cover server-side threading issues in a future column.

OODCE and Network OLE both have provisions for multi-threading (OODCE via DCE pthreads and Network OLE via Win32 threads). The OMG CORBA standard, on the other hand, considers threading to be outside of its scope. Certain CORBA ORBs (such as Orbix [11]) provide hooks that integrate threading into an application in a relatively transparent and portable manner.

Deferred Activation: Deferred activation is a technique that activates only those objects that are actually requested to perform services on the behalf of clients. Such activation, which is completely transparent to the client, is needed in large-scale networks to allow finite computing resources (such as memory and CPU cycles) to be used efficiently. Objects that are not currently in use may remain dormant, knowing that they will be activated if necessary.

Support for deferred activation is useful for certain types of objects in our stock trading application such as trading rules illustrated in Figure 2. Certain trading rules may only be required under circumstances that occur infrequently (such as a major market correction). By activating these objects "on demand," the load on host process systems may be reduced significantly.

A conforming CORBA ORB is required to activate certain types of objects when requests arrive for them, if the objects are not already up and running. Likewise, DCE provides a similar service via its "DCE daemon" (`dced`), which is modeled after the Internet superserver `inetd` [6]. Windows

[1] Unfortunately, there are several overloaded terms here! Location brokers and traders in DOC frameworks are quite different from stock brokers and traders, though they share some striking similarities.

NT and Network OLE provide a service control management facility that initiates and controls network services on a Windows NT endsystem [13].

Binary Data Exchange: The stock trading application passes binary data between little- and big-endian machines. Therefore, marshaling and unmarshaling must be performed for requests and responses with fields that contain binary values. As described in Section 2.1, all DOC frameworks perform these tasks reasonably well.

As you can see, the requirements for a distributed application like our stock trading example are numerous and complex, perhaps to the point of being overwhelming. The DOC frameworks and systems discussed in this article provide varying degrees of support for these requirements. In general, the current generation of commercially available DOC frameworks handle certain requirements fairly well (such as network heterogeneity, object location and selection, synchronous communication and threading, deferred activation, security, and binary data exchange). Other requirements are not handled as thoroughly at this point (such as OS heterogeneity, reliability and availability, load balancing, and interoperability). Naturally, there are exceptions to these generalizations (*e.g.,* Isis supports fault tolerant DOC and OODCE handles interoperability quite well). We expect the next-generation DOC frameworks to provide more comprehensive and better integrated support for common distributed application requirements.

3 Concluding Remarks

In this column, we analyzed the distributed computing environment and requirements of a representative distributed stock trading application. Using this application, we identified a number of common object services that help satisfy the distributed computing requirements of many emerging applications. In our next several articles, we will evaluate various programming techniques for the client-side and the server-side of our example application. We'll compare several solutions, ranging from using the conventional sockets network programming interface (which is written in C), to using C++ wrappers for sockets, all the way up to using the CORBA Interface Definition Language (IDL) and a CORBA ORB. Each solution illustrates various tradeoffs between extensibility, robustness, portability, and efficiency.

In future columns we'll cover the DOC frameworks and features mentioned above in much greater detail. Our goal is to be as comprehensive and unbiased as possible. Please let us know if we've failed to mention your favorite DOC framework, or if you feel that we've misrepresented the features and functionality of certain tools and technologies. As always, if there are any distributed object topics that you'd like us to cover in future articles, please send us email at `object_connect@ch.hp.com`.

References

[1] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition).* Redwood City, California: Benjamin/Cummings, 1993.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 (draft) ed., May 1995.

[4] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," in *Proceedings of the Winter Usenix Conference*, USENIX Association, January 1995.

[5] Microsoft Press, Redmond, WA, *Object Linking and Embedding Version 2 (OLE2) Programmer's Reference, Volumes 1 and 2*, 1993.

[6] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[7] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[8] Object Management Group, *Common Object Services Specification, Volume 1*, 94-1-1 ed., 1994.

[9] Isis Distributed Systems, Inc., Marlboro, MA, *Isis Users's Guide: Reliable Distributed Objects for C++*, April 1994.

[10] K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*. Los Alamitos: IEEE Computer Society Press, 1994.

[11] C. Horn, "The Orbix Architecture," tech. rep., IONA Technologies, August 1993.

[12] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[13] Microsoft Press, Redmond, WA, *Microsoft Win32 Programmer's Reference*, 1993.