Constructing Reliable Distributed Communication Systems with CORBA

Silvano Maffeis
maffeis@acm.org
Olsen and Associates
Zurich, Switzerland

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

This paper will appear in the feature topic issue on Distributed Object Computing in the IEEE Communications Magazine, Vol. 14, No. 2, February 1997.

Abstract

Communication software and distributed services for next-generation applications must be reliable, efficient, flexible, and reusable. These requirements motivate the use of the Common Object Request Broker Architecture (CORBA). However, building highly available applications with CORBA is hard. Neither the CORBA standard nor conventional implementations of CORBA directly address complex problems related to distributed computing, such as real-time or high-speed quality of service, partial failures, group communication, and causal ordering of events. This paper describes a CORBA-based framework that uses the Virtual Synchrony model to support reliable data- and process-oriented distributed systems that communicate through synchronous methods and asynchronous messaging.

1 Introduction

Communication software and distributed services for next-generation applications must be reliable, efficient, flexible, and extensible. For instance, applications like personal communication systems (PCS), real-time market data feeds, and flight reservation systems must be highly available and scalable to meet their stringent reliability and performance demands. In addition, these applications must be flexible and extensible to cope with their inherent complexity and to respond rapidly to changing application requirements that span a wide range of media types and access patterns.

These requirements motivate the use of the Object Management Group's Common Object Request Broker Architecture (OMG CORBA) [1, 2]. CORBA is an open standard for distributed object computing. It defines a set of components that allow client applications to invoke operations on remote object implementations. CORBA enhances application flexibility and portability by automating many common development tasks such as object registration, location, and activation; demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

Experience over the past several years [3] illustrates that CORBA is well-suited for best-effort, client-server applications running over conventional local area networks (such as Ethernet and Token Ring). However, building highly available applications with CORBA is much harder. Neither the CORBA standard nor conventional implementations of CORBA directly address complex problems related to distributed computing, such as real-time quality of service [4] or high-speed performance [5], group communication [6], partial failures, [7] and causal ordering of events [8].

This paper makes three contributions to the study of reliable distributed object computing systems with CORBA. First, we examine the question of whether reliable applications can (or should) be implemented with CORBA today. Next, we present an extension to the Object Management Architecture that improves support for reliability and fault-tolerance. Finally, we propose a CORBA-based framework based on the Virtual Synchrony model [7] that supports reliable data- and process-oriented distributed systems. In addition, our proposed framework supports applications requiring loosely coupled processes that communicate through asynchronous messaging.

2 Reliability Matters

Distributed systems are intended to form the backbone of emerging next-generation communication systems, including electronic commerce, PCS, satellite surveillance systems, distributed medical imaging, real-time data feeds, and flight reservation systems. An obvious benefit of distributed systems is that they reflect the global business and social environments in which we live and work. Another benefit is that they can improve the *quality of service* (in terms of reliability, availability, and performance) for complex systems.

Reliability is an important quality in mission-critical distributed applications. In many distributed environments, even small amounts of downtime can annoy customers, hurt sales, or endanger human lives. We define a distributed system as *reliable* if its behavior is predictable despite of partial failures, asynchrony, and run-time reconfiguration of the system. In addition, we require reliable applications to be *highly available*, *i.e.,* the application can provide its essential services despite the failure of computing nodes, software objects, and communication links.

Certain aspects of distributed systems make reliability more difficult to achieve. For instance, partial failures are an inherent problem in distributed systems. The "mean time to failure" (MTTF) of components in a distributed system rapidly decreases as the number of computing nodes and communication links that constitute the system increases. Another inherent problem is that developers must address complex execution states of concurrent programs. Distributed systems consist of processes that run in parallel on heterogeneous platforms and are therefore prone to race conditions, communication errors, node failures, and deadlocks. Thus, distributed systems are often more difficult to develop, administer, and maintain than their centralized counterparts.

Conversely, other aspects of distributed systems can help make applications more robust. For instance, distributed systems can be made more reliable than centralized systems by providing important services redundantly on multiple nodes. To enable redundancy, active or passive replication should be supported by the communication system used to run distributed applications.

Hence, we face a peculiar situation: although programming distributed applications is a daunting and error-prone task, a high degree of failure tolerance and reliability can be achieved if the underlying communication system software supports replication. The conclusion we draw is that non-robust communication software will lead to fragile distributed applications that will be frequently unavailable and will require constant supervision. In contrast, sophisticated communication software (such as the Virtual Synchrony approach presented in Section 4.3) can lead to distributed applications that are inherently more reliable, more modular, and more scalable than centralized ones.

3 Software Quality Matters

Most reported success with object technology has involved centralized applications running on stand-alone computers. In particular, user interfaces have widely adopted object-oriented design and programming techniques [9]. In contrast, relatively few examples [10, 3] of object-oriented distributed systems are currently deployed in production commercial systems.

There are a number of interesting projects that are currently employing object-oriented distributed technology and which are planned to become operational shortly. For example the Iridium system, which is being designed and manufactured by Motorola and associated companies, intends to provide global personal satellite based communications via hand-held terminals by the year 1998. Iridium will use 66 satellites orbiting in six 450 miles altitude polar planes and will cost in excess of $4 billion US dollars. The number of subscribers to this system is expected to exceed 1 million. Motorola is using CORBA to implement portions of the Iridium system control software.

Building distributed systems is complex and expensive. Distribution presents developers with a number of *inherent problems* such as latency, concurrency control, heterogeneity, and partial failures. Further complicating matters are *accidental problems* such as the lack of widely reused higher-level application frameworks, primitive debugging tools, and non-scalable, unreliable software infrastructures.

Distributed object models like CORBA were devised to address several of these problems. In CORBA, objects are specified in a strongly typed interface definition language (IDL). Thus, CORBA objects can be used to hide *heterogeneity* and the details of the underlying system software, communication protocols, and hardware. For instance, two CORBA objects running on the Object Request Brokers (ORBs) of different manufacturers can interoperate with each other even when implemented in different programming languages, operating systems, or hardware platforms.

CORBA's synchronous method invocation model can help programmers avoid *concurrency* related problems. Moreover, CORBA object services (such as the Event, Concurrency, and Transaction Service) help to orchestrate the activities of distributed network objects that execute in parallel across local area and wide area networks.

The CORBA model by itself does not provide solutions to the problem of detecting and reacting to *partial failures* and to *network partitioning*. However, a CORBA object can encapsulate internal state and make it accessible through an IDL interface. Due to this encapsulation, fault-tolerance techniques like replication and state-checkpointing become easier to implement because the internal state of an object is isolated.

There is a need for distributed *debugging* tools and run-time validation tools in the CORBA model. Distributed debugging has not been addressed by the OMG yet. However, viewing a complex system in the form of distributed state machines that interact by invoking operations on each other can help in tracing distributed activities and in isolating and correcting problems.

Reliable distributed computing requires the presence of an execution model that allows programmers to predict the behavior of a distributed application despite asynchrony, concurrency, and partial failures. There are three interesting models that can help in building reliable systems: *message queues*, *TP monitors*, and *Virtual Synchrony*. Each model represents a certain view of distributed computing and has its specific advantages and disadvantages. Below, we give an overview of the models and explain how CORBA can be used to define a unified model for reliable distributed object computing.

4 Evaluating Alternative Solutions

4.1 Message Queues

4.1.1 Overview

A straightforward approach to reliability employs *message queues (MQ)* for inter-process communication (Figure 1).

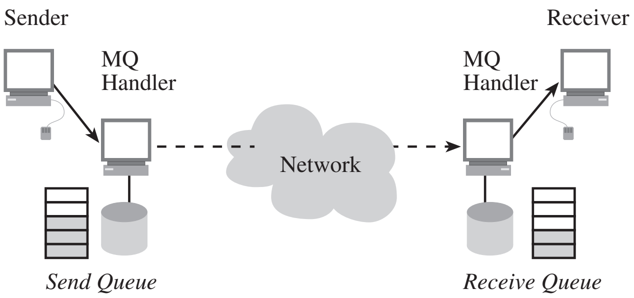

Figure 1: Inter-process Communication Using Message Queues.

A process S that wants to reliably submit message M to process R submits the message to its local MQ handler. The handler writes the content of the message on non-volatile storage to avoid message loss if a crash occurs. After having submitted the message, process S is relieved from any activity necessary to deliver M. The MQ handler consists of an independent process that is responsible for storing and delivering messages on behalf of application processes. S's handler attempts to transfer the message to R's handler. If the destination handler happens to be unavailable because of downtime, a site crash, or a network partition, S's handler will attempt to deliver the message periodically until R's handler becomes available.

Reliability is achieved by decoupling sender processes from receiver processes. A sender can submit a message without having to know whether the receiver is unavailable and without having to deal with transient network failures. The model can tolerate the failure of MQ handlers because messages are written to non-volatile storage automatically and can be retransmitted, if necessary.

4.1.2 Variations

Several refinements to the basic Message Queue scheme are possible. For instance, instead of logging messages to disk, the queue could be maintained in memory to increase performance. To increase reliability, the queue must be replicated among two or more nodes, or a battery-backed RAM disk or *Flash Memory* could be employed as a non-volatile message buffer.

A publish/subscribe API can be used to present a convenient access point to the MQ. Receiver-processes register their interest in particular messages by subscribing to a specific *subject* with the MQ handler. Sender-processes submit messages, using subjects as addresses. A sender process can be replaced by another process at run-time, as long as the new process services the same subjects as the original one. The ability to replace service implementations transparently (*i.e.,* without affecting clients) is important for configurability and availability.

An area related to message queues is *mail enabled applications*. These applications use existing e-mail mechanisms to store-and-deliver inter-process communication (IPC) messages. Thus, e-mail can be used as a powerful and ubiquitous type of middleware in many organizations. Several APIs for mail enabled applications exists, for example VIM (jointly backed by Lotus, Apple, IBM, Borland, MCI, Oracle, WordPerfect, and Novell), MAPI (by Microsoft), and CMC (by the X.400 API association). These APIs provide functions for single-point and multi-point message delivery, message box manipulation, managing address directories, composing mail, authentication, and security.

4.1.3 Evaluation

The MQ model is well-suited for applications that can be interconnected by an asynchronous, one-way, "forward-and-forget" communication paradigm. The advantage of the MQ model is that it is easy to use, implement, and understand. In addition, it supports disconnected operation of mobile equipment. To that purpose, a mobile device can direct messages to a local MQ handler while disconnected. Upon reconnection to a backbone network the messages are transmitted over the network by the handler.

The drawbacks are that some type of recoverable message log must be provided and that high-availability is not supported. Using a disk to log messages is the most straightforward approach to recoverability. However, disks impose tight limits on the maximum throughput. A battery-buffered RAM disk can lead to better performance, albeit at a higher cost.

Another drawback is the lack of support for two-way communication in many MQ products. One manifestation is the lack of return values. Furthermore, senders typically have no guarantee of delivery; they don't know when or if a message is delivered.

The message queue model does not ensure high availability. In case of a crashed queue handler, a queued (but not yet delivered message) remains unavailable until the defect that lead to the crash is repaired. A replicated RAM queue can offer good performance and availability, but requires sophisticated group communication support, which we describe in Section 4.3.

4.2 Transaction Processing Monitors

4.2.1 Overview

A *transaction processing monitor* (TP monitor) [11] allows a distributed client application to bracket a series of service invocations by begin/end transaction markers. If a service fails during a transaction, the transaction monitor will *roll back* invocations issued within the transaction.

TP monitors typically provide two-way commit protocols and serializability of requests (Figure 2). Thus, TP monitors can be used to maintain distributed data consistency in spite of crash failures, by employing a roll back mechanism. TP monitors are primarily aimed at *data-oriented* applications that manage distributed, persistent data objects. Examples of data-oriented applications include management information systems, flight-reservation systems, and business workflow management.

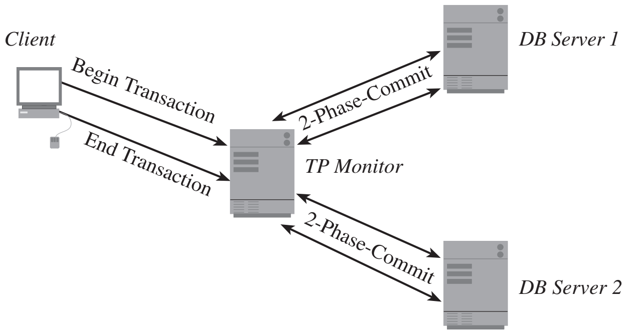

Figure 2: Transaction Processing Monitor.

4.2.2 Evaluation

TP monitors are well-understood and have been applied to mission critical applications for many years. The primary limitations with traditional transaction models are (1) they are hard to program and (2) they introduce substantial performance overhead and excessive serialization in many situations such as groupware applications and real-time market data feeds. Various extensions have been proposed to cope with the serialization problem, notably nested atomic transactions.

4.3 Virtual Synchrony

4.3.1 Overview

Virtual Synchrony is a distributed execution model that is lower-level and more fundamental than the message queue and transaction process monitoring mechanisms discussed earlier. The Virtual Synchrony model was originally developed by Ken Birman in the context of the Isis toolkit [7]. Virtual Synchrony guarantees that the behavior of a distributed application is predictable regardless of partial failures, asynchronous messaging, and objects that join and leave the system dynamically. At the core of the model are a *failure suspector service* and a *group abstraction*.

The failure suspector service detects faulty objects and guarantees that non-faulty objects have a consistent view of which objects are believed faulty. The failure suspector relies on timeouts to detect suspicious objects. It only detects crash failures; it assumes that objects fail by crashing without emitting spurious messages.

In an asynchronous system it is impossible to distinguish a crashed object from one that is very slow [12]. A downside of failure suspector services is that they might report a healthy object as suspicious, for example when a machine or a network happens to be temporarily overloaded. In practice, this is not a problem as long as (1) false suspicions occur infrequently and (2) failure beliefs are propagated to all objects that have an interest in the suspicious objects.

Without consistent views on partial failures the following situation might occur. A client application mistakenly believes a server object as faulty, due to a high temporary load on the server's machine. However, another client application is able to interact with that server without any problem. Believing the server has crashed, the first client requests the run-time system to create a new instance of the server on another host. The result is that now two servers with an inconsistent internal state exist, and that both clients and servers believe that the system is running correctly. In reality, the clients are submitting update requests to two different servers, which places the system into an inconsistent state.

High availability is ensured by process replication. To achieve this, Virtual Synchrony provides a *group abstraction* and a reliable multicast mechanism. Highly-available objects can be created on several computing nodes, the instances are then joined to an *object group* [6]. The object group abstraction allows programmers to assign a single object reference to a set of network objects that implement the same interface (Figure 3).

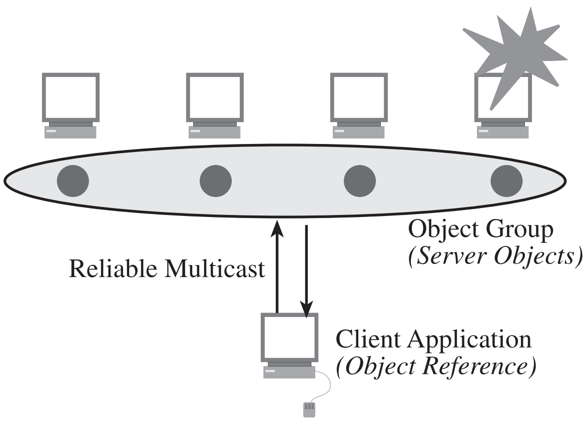

Figure 3: Object Group Computing in the Virtual Synchrony Model.

Object groups appear like single entities to the client application. Client requests are transparently multicast to the group. A request succeeds as long as at least one group member is operational. In contrast to TP monitors, Virtual Synchrony implements a *roll forward* recovery mechanism in which crashed objects are restarted and rejoined to their object group. Upon joining a group, an object obtains a copy of the replicated group state.

The application areas of Virtual Synchrony are primarily *process-oriented* applications. Process-oriented applications maintain a volatile state and communicate mainly by message passing and RPC. Examples of process-oriented applications include teleconferencing systems, PCS, real-time stock exchange feeds, and satellite surveillance systems.

The primary characteristics of process-oriented applications is that (1) low-overhead IPC is required, (2) their network objects often need to communicate in an asynchronous manner, (3) part of their network objects need to be highly available, and (4) little or no persistent state needs to be maintained. Database management systems and transaction monitors are thus not well-suited to coordinate the activities of a process-oriented application because they introduce an excessive amount of synchronization and overhead.

4.3.2 Evaluation

Generally speaking, Virtual Synchrony is appropriate for applications that need to maintain a distributed, volatile state. The model assumes that processes are rather tightly coupled and interested in up-to-date information on failures and group membership changes. This implies that applications are aware of the underlying Virtual Synchrony middleware. The middleware signals membership changes and failures to the server applications by delivering *upcalls* to them, the applications must be prepared to deal with those upcalls.

One downside of present work on Virtual Synchrony is that it has led to a variety of toolkits with incompatible, and often low-level, programming interfaces. Examples of toolkits that implement Virtual Synchrony are Horus, Isis, and Transis. Those toolkits provide a rather low-level message passing API in the form of `C` and `SML` programming libraries. Due to the lack of high-level abstractions, standard APIs, and frameworks, applications can become hard to implement, administer, and maintain.

5 A Unified Reliability Model

The three models we described above provide different types of reliability, which are summarized in Figure 4. TP monitors are effective for orchestrating *data-oriented* distributed systems that manage persistent data. Message queues are well-suited for applications that consist of *loosely-coupled* processes that mainly interact asynchronously. Virtual Synchrony is a fundamental reliability model intended for *process-oriented* distributed systems that must be highly available.

While none of the models is powerful enough to be viewed as a complete solution to reliable distributed computing, they do provide complementary functionality and guarantees. In this section, we outline a unified object-oriented architecture that combines the three models and allows applications to pay only for reliability guarantees they need. For instance, in this architecture, asynchronous applications may employ a message queue without having to pay for the overhead incurred by a transaction monitor. Likewise, tightly-coupled, process-oriented applications can run efficiently atop of a virtually synchronous communication subsystem. Finally, data-oriented applications can access a "plug-in" TP monitor to coordinate access to distributed data objects.

Our architecture consists of an extended CORBA Object Request Broker (Figure 5). The ORB runs on top of a virtually synchronous group communication subsystem like Horus, Isis, Totem, or Transis [13]. This enhanced ORB supports the abstraction of an *object group* [6], meaning that CORBA objects of the same type can be named and accessed as a single entity. Object groups allow the run-time replication of stateful CORBA objects and efficient multicast of CORBA requests.

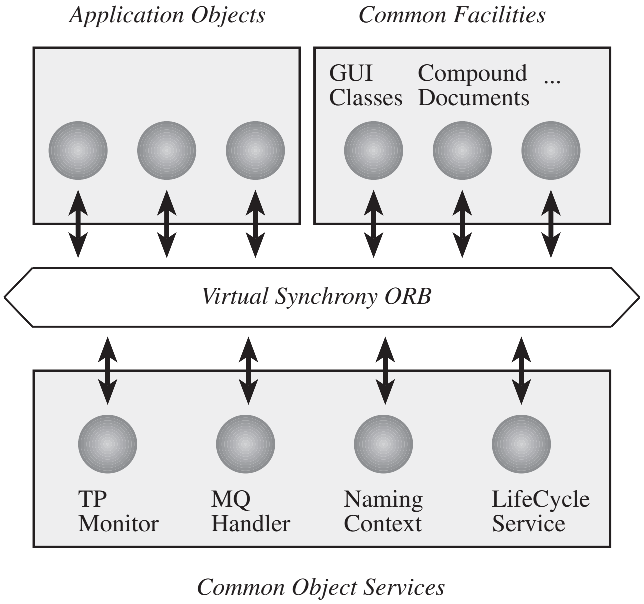

Figure 5: Extended CORBA Architecture for Reliable Systems.

From the programmer's point of view, a virtually synchronous ORB appears like a conventional ORB, except that an extended CORBA API is presented [8]. The extended API provides functions for creating object groups, joining an object implementation to a group, destroying groups, and removing an object implementation from a group. These calls are provided through a special CORBA Object Adapter. Further, the ORB provides functionality that enhances reliability by allowing applications to set *upcalls*. These upcalls will be invoked by the ORB when certain object implementations are believed to be faulty. Such an upcall might either display a warning message, attempt to restart the crashed object, or introduce another application-specific behavior.

The TP Monitor and MQ Handler are provided in the form of plug-in OMG Common Object Services on top of the ORB. The virtually synchronous ORB facilitates the implementation of a TP monitor considerably. For instance, reliable multicast can be employed to deliver *commit* and *abort* notifications to objects that participate in a transaction. Failure detection is useful for detecting crashes that occur during a transaction.

In addition, Virtual Synchrony facilitates a robust and ef-

	System Model	*Overhead*	*Reliability Mechanism*	*High Availability*	*IPC*	*State Transfer*
TP Monitor	Data Oriented	High	2 phase commit, Roll-Back	No	Queries RPC	(Yes)
Message Queue	Message Oriented	Low or High	Spooling, Multi-plexing	No or Yes	Async. Messages	No
Virtual Synchrony	Process Oriented	Low	Replication, Roll-Forward	Yes	Async./ Sync. Messages, Multicast	Yes

Figure 4: TP Monitors Versus Message Queues Versus Virtual Synchrony.

ficient implementation of the Message Queue model. An object group can be used to implement a replicated in-memory message log. Request-multicast is useful for distributing a message to all objects that have registered an interest in a certain subject. Thus, the messaging infrastructure can exploit hardware multicast where available, and does not impose any single point of failure.

6 Concluding Remarks

Contemporary CORBA Object Request Brokers were built in a straightforward fashion using communication support provided by operating systems (notably BSD sockets and RPC). As a consequence, predicting the behavior of distributed applications implemented atop of such CORBA ORBs is often impossible when partial failures occur. This is becoming a serious problem since many organizations are planning to implement mission-critical software with CORBA.

In this paper we recommend extending the CORBA model to support the Virtual Synchrony model. This can be achieved by layering an ORB on top of a toolkit such as Horus, Isis, Totem, or Transis, rather than building directly atop sockets or other low-level communication mechanisms. Examples of CORBA ORBs that implement the Virtual Synchrony model are Orbix+Isis and Electra [8].

TP monitors have been in use for almost a decade, for example in mission critical banking applications. Well-known examples are Tuxedo and Transarc's Encina. CORBA request brokers also have been combined with TP monitors leading to products such as Orbix+Tuxedo. The OMG has recently standardized a TP monitor specification designed to be used in conjunction with CORBA applications.

Message queues have been extensively used in the financial domain for several years. This type of service is often referred to as *message oriented middleware (MOM)*, *information buses*, or *event streams*. Examples are Isis Message Distribution System, TIBCO Information Bus, Iona's OrbixTalk, and IBM's MQSeries. The OMG has recently standardized an Event Channel service specification to be used in conjunction with CORBA applications.

OrbixTalk, for example, is an OMG compliant Event Service that distributes requests via IP multicast. It employs negative acknowledgements to make sure that a request is delivered to every object that has subscribed for it. However, OrbixTalk neither provides totally ordered multicast nor provides Virtual Synchrony. Furthermore, requests can get lost under high load situations or when a receiver detects that it has missed a message after the sender has discarded that message from its internal message queue.

Portions of the integrated architecture outlined in this paper have been realized in the context of the Orbix+Isis and Electra projects [8]. Orbix+Isis and Electra are CORBA object request brokers that run on top of Virtual Synchrony toolkits such as Isis. Unfortunately, both ORBs are suited to support process-oriented applications, but do not provide a TP monitor service to accommodate applications that manage distributed data objects. We expect that subsequent generations of CORBA middleware will support this behavior.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1995. Revision 2.0.

[2] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[3] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[5] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), ACM, August 1996.

[6] S. Maffeis, "The Object Group Design Pattern," in *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, (Toronto, Canada), USENIX, June 1996.

[7] K. P. Birman and R. van Renesse, eds., *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[8] S. Landis and S. Maffeis, "Building Reliable Distributed Systems with CORBA," *Theory and Practice of Object Systems, John Wiley Publisher, NY, to appear*, Apr. 1997.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[10] M. L. Brodie, "Putting objects to work on a massive scale," in *Foundations of Intelligent Systems, 9th International Symposium ISMIS'96*, Lecture Notes In Artificial Intelligence, (Zakopane, Poland), Springer-Verlag, June 1996.

[11] R. Orfali, D. Harkey, and J. Edwards, *The Essential Client/Server Survival Guide*. Wiley and Sons, 1996.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, Apr. 1985.

[13] "Special section on group communication." Communications ACM, 4 39.