UNIVERSITY OF CALIFORNIA
Santa Barbara

Transparent Fault Tolerance for CORBA

A Dissertation Submitted in Partial Satisfaction
of the Requirements for the Degree of

Doctor of Philosophy
in
Electrical and Computer Engineering
by

Priya Narasimhan

December 1999

The dissertation of Priya Narasimhan is approved

Committee Chairperson

September 1999

September 1999

For my parents, Kala and Nallan Chakravarty Narasimhan,
who taught me to work hard and dream big,
and who made all of this possible through
their love, encouragement and sacrifices

ACKNOWLEDGMENTS

Durgam Kāj Jagath Ke Jete
Sugam Anūgrah Tumhare Tete
(All the difficult tasks in this world
are rendered easy through Your Grace)
— *Hanuman Chalisa* by Tulsidas

I would like to thank God for making it possible for me to come here five years ago, for granting me the good fortune to work on a research problem that I enjoyed, and for seeing me through the ups and downs that every Ph.D. engenders.

My parents have sacrificed a good many years of their lives to make my education possible. As difficult as it was for them, they have always understood when the pressures of my research sometimes made it impossible for me to visit them or to be with them as often as I would have liked. This dissertation is dedicated to my parents – thank you for seeing me through every step of the way, for praying for me during times of trouble, and for rejoicing with me over every little triumph.

Nitya (my sister and apartment-mate of four years), thank you for always being there for me with your sympathetic ear, your shoulder (well-worn by now), your unquenchable optimism, and your great cooking! Rajeev, thank you for your solid advice, your support of my ambitions, and for keeping me focussed during difficult times.

I would like to thank Professors Louise E. Moser, Michael Melliar-Smith, Steven Butner, Rachid Guerraoui and Douglas Schmidt for agreeing to serve on my Ph.D. dissertation committee, for reading through the dissertation in its various stages, and for giving me the feedback that shaped the dissertation into its ultimate form. Thank you also for your willingness to accommodate my defense and dissertation filing dates into your respective schedules.

I would especially like to thank my advisors, Louise E. Moser and Michael Melliar-Smith, for teaching me so much over the past four years, and for inspiring me with the example of their hard work and their intensity. Louise, thank you for teaching me how to put my thoughts into words, and how to turn out a good paper. Michael, thank you for opening the door that brought me to UCSB five years ago. Thank you both for all the knowledge and experience that you have shared with me so freely, and for giving me the exposure and the opportunities that have helped my career.

I would like to thank Doug Schmidt, for being available to answer technical questions on CORBA or on programming in general. Doug, thank you for looking out for me, for publicizing my work, and for being a great source of information and knowledge. I would like to thank Rachid Guerraoui for seeing me through the entire course of my research – right from my very first publication on Eternal through to my dissertation. Rachid, thank you for your support of my work, and for your technical insights that have often helped me to see things in a different light.

I would like to thank all of the faculty at the BMS College for Women (Bangalore, India), in particular, Rathnamma, Mookambika, Umadevi and Swarnagowri, who believed in me during my undergraduate days, and continue to do so to this day. I would also like to thank Dr. Bhaskar N. Rao, one of the best teachers that I have ever been privileged to know, for introducing me to the joys of electrical engineering.

Special thanks are due to Albert Alexandrov and Professor Klaus Schauser (Computer Science Department at UCSB) for introducing me to interceptors, and for letting me use their code. I would particularly like to thank Steve Rago (Programmed Logic Corporation) for extensive discussions on operating system hooks for interception.

Dr. Murthy Devarakonda (IBM Research), thank you for your encouragement that led to the interview with IBM Research, and for following up on my career since. Dr. Stuart Tewksbury, thank you for taking the trouble to give me the benefit of your wisdom. Dr. Shalini Yajnik (Lucent Technologies), thank you for all the interesting and useful personal and technical discussions that we have had over the past year. Ramanathan Krishnamurthy (Object-Oriented Concepts, Inc.), thank you for being a good friend to me over the past ten years.

I would also like to thank all of my colleagues in the Computer Networks and Distributed System Laboratory who shaped this work in one way or another. Mike, thank you for being the best sounding board in the world for every one of my fledgling ideas. Thank you for letting me benefit from your vast system administration and programming knowledge – you are a joy to know! Ravi, thank you for helping me cut my teeth in this lab when I first started, and for sharing your knowledge of Totem with me. Kim, thank you for the experience and the collaboration on the Immune system that made the project worthwhile. Thank you for the tiramisu that you brought in to cheer me up! Ruppert, thank you for your continuous assistance with all of my questions about Totem. Fabrice, thank you for building a wonderful air defense demonstration that allows me to show off my code. Lauren, thank you for your moral support during stressful times.

This research has been supported by the Defense Advanced Research Projects Agency in conjunction with the Office of Naval Research and the Air Force Research Laboratory, Rome, under Contracts N00174-95-K-0083 and F3602-97-1-0248, respectively.

VITA

1990 **B. Sc.**
Departments of Physics, Mathematics and Electronics
BMS College for Women, Bangalore, India

1995 **M. S.**
Department of Electrical and Computer Engineering
University of California, Santa Barbara

1995-99 Graduate Student Researcher
Department of Electrical and Computer Engineering
University of California, Santa Barbara

PUBLICATIONS

Object-Oriented Programming of Complex Fault-Tolerant Real-Time Systems, L. E. Moser, P. Narasimhan and P. M. Melliar-Smith, *Proceedings of the IEEE Computer Society Second International Workshop on Object-oriented Real-time Dependable Systems*, Laguna Beach, CA (February 1996), pp. 116-119.

Message Packing as a Performance Enhancement Strategy with Application to the Totem Protocols, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the IEEE Global Telecommunications Conference*, London, England, UK (November 1996), pp. 649-653.

Consistency of Partitionable Object Groups in a CORBA Framework, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the IEEE Hawaii International Conference on System Sciences*, Maui, HI (January 1997), pp. 120-129.

Separation of Concerns: Functionality vs. Quality of Service, P. M. Melliar-Smith, L. E. Moser and P. Narasimhan, *Proceedings of the IEEE Third International Workshop on Object-oriented Real-time Dependable Systems*, Newport Beach, CA (February 1997), pp. 272-274.

Replica Consistency of CORBA Objects in Partitionable Distributed Systems, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Distributed Systems Engineering Journal* vol. 4, no. 3 (September 1997), pp. 139-150.

Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, OR (June 1997), pp. 81-90.

The Interception Approach to Reliable Distributed CORBA Objects, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Panel on Reliable Distributed Objects, *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, OR (June 1997), pp. 245-248.

On Technologies in Computer Networks and Distributed Systems, P. M. Melliar-Smith, L. E. Moser, K. Berket, R. K. Budhia, K. P. Kihlstrom, R. Koch, N. Narasimhan, P. Narasimhan, E. M. Royer, M. D. Santos, A. Shum, and E. Thomopoulos, *looking.forward* supplement to *IEEE Computer*, 5(3):2–6 (Fall 1997).

The Eternal System, L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, *Workshop on Dependable Distributed Object Systems, OOPSLA'97*, Atlanta, GA (October 1997).

Consistent Object Replication in the Eternal System, L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 81-92.

Supporting Enterprise Applications with the Eternal System, L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, V. Kalogeraki and L. Tewksbury, *Proceedings of the IEEE Conference on Enterprise Networking and Computing*, ICC/SUPERCOMM '98, Atlanta, GA (June 1998).

The Realize Middleware for Replication and Resource Management, P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki and P. Narasimhan, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware '98*, The Lake District, England, UK (September 1998), pp. 123-138.

Fault Tolerance for CORBA, L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, *OMG Technical Committee Document orbos/98-10-08*, Object Management Group (October 1998). *Technical Report 98-27*, Department of Electrical and Computer Engineering, University of California, Santa Barbara.

Providing Support for Survivable CORBA Applications with the Immune System, P. Narasimhan, K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Austin, TX (May 1999), pp. 507-516.

A Fault Tolerance Framework for CORBA, L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, *Proceedings of the IEEE 29th International Symposium on Fault-Tolerant Computing*, Madison, WI (June 1999), pp. 150-157.

Replication and Recovery Mechanisms for Strong Consistency in Reliable Distributed Systems, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the ISSAT International Conference on Reliability and Quality in Design*, Las Vegas, NV (August 1999), pp. 26-31.

Using Interceptors to Enhance CORBA, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *IEEE Computer*, vol. 32, no. 7 (July 1999), pp. 62-68.

Multicast Group Communication for CORBA, L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, R. R. Koch and K. Berket, *Proceedings of the IEEE International Symposium on Distributed Objects and Applications*, Edinburgh, Scotland, UK (September 1999), pp. 98-109.

The Eternal System: An Architecture for Enterprise Applications, L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki, *Proceedings of the IEEE 3rd International Enterprise Distributed Object Computing Conference*, Mannheim, Germany (September 1999), pp. 214-222.

Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the IEEE 18th Symposium on Reliable Distributed Systems*, Lausanne, Switzerland (October 1999), pp. 263-273.

Transparent Fault Tolerance for CORBA using the Eternal System, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the International Workshop on Reliable Middleware Systems*, Lausanne, Switzerland (October 1999), pp. 7-13.

Gateways for Accessing Fault Tolerance Domains, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware 2000*, New York, NY (April 2000).

Eternal: Fault Tolerance and Live Upgrades for Distributed Object Systems, L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki, *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC (January 2000).

Realize: Resource Management for Soft Real-time Distributed Systems, P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki and P. Narasimhan, *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC (January 2000).

Patterns for Building Reliable Distributed Object Systems, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, *Theory and Practice of Object Systems* (Spring 2000).

FIELDS OF STUDY

Major Field: Computer Engineering

Interceptors for CORBA
Professor L. E. Moser and Professor P. M. Melliar-Smith

Fault Tolerance for CORBA
Professor L. E. Moser and Professor P. M. Melliar-Smith

Survivability for CORBA
Professor L. E. Moser and Professor P. M. Melliar-Smith

ABSTRACT

Transparent Fault Tolerance for CORBA
by
Priya Narasimhan

Applications are increasingly being programmed using the CORBA distributed object standard. CORBA's Internet Inter-ORB Protocol (IIOP) and its mediating Object Request Broker (ORB) allow CORBA objects to interact, transcending differences in their locations, hardware architectures, operating systems and programming languages.

The Eternal system provides the fault tolerance that CORBA lacks. Because typical applications are already quite complex, and because typical application programmers do not have skills in fault tolerance, Eternal provides fault tolerance without requiring the modification of applications, or the modification of complex commercial ORB code.

The transparency of Eternal's fault tolerance infrastructure to both the application and the ORB is possible through the use of interception technology. The Eternal Interceptor transparently captures the IIOP messages exchanged between the CORBA objects of the application, and diverts these messages to the Eternal Replication Mechanisms and Logging-Recovery Mechanisms.

The Eternal system provides fault tolerance through object replication, with support for active and passive replication, duplicate detection and suppression, state transfer, logging and recovery. The use of the Totem reliable totally-ordered multicast protocol to communicate IIOP messages between replicated objects facilitates replica consistency. Eternal can exploit other multicast group communication protocols, such as the SecureRing secure reliable totally-ordered multicast protocol, to provide support for effective majority voting for CORBA applications.

Strong replica consistency is ensured for both passive and active replication, as replicas fail and recover, and as operations are performed that update the states of the replicated objects. Recognizing that most CORBA applications and ORBs employ multithreading, a source of non-determinism, Eternal provides mechanisms to enforce determinism transparently, thereby ensuring replica consistency even for multithreaded applications.

Eternal has been deployed on seven different unmodified commercial CORBA ORBs. Unmodified applications triply replicated by Eternal incur a 10% overhead in response time compared to their non-fault-tolerant counterparts. The technology of Eternal forms the basis of the forthcoming standard for Fault-Tolerant CORBA.

Contents

List of Figures

Chapter 1

Introduction

The distributed computing model is appropriate for many applications, especially those that require the sharing of resources across a network, the distribution of workload for load balancing purposes, or the localization of services for increased availability. Most distributed computing applications are based on the client-server model, where client software requests, and obtains services from, server software that is typically not on the same machine.

Another powerful and widely used design and programming methodology is the object model. The notions of structure (data) and behavior (methods to access and modify the data) are combined into a single entity, called an object, that can be accessed only via a published or a well-known interface. The object model encompasses the concepts of abstraction, encapsulation, inheritance and polymorphism. The power of the object model lies in its ability to separate interface from implementation, to promote modularity and the reuse of components, to provide for the hierarchical composition of interfaces, and to support the substitutability of objects with matching interfaces.

The integration of the distributed computing model and the object model leads to distributed object computing. Objects are distributed across machines, with client objects invoking operations on, and receiving responses from, remote server objects. Both the client's invocations, and the server's responses, are conveyed in the form of messages sent across the network.

1.1 Common Object Request Broker Architecture

There are several distributed object computing models. One of these, the Common Object Request Broker Architecture (CORBA) [20, 54], was established by the Object Management Group[1] in 1989 as a standard for distributed object computing. Since then, the CORBA

[1]The OMG is a nonprofit consortium of over 800 corporations, government agencies and academic institutions that was established to develop an interoperable distributed object computing standard, CORBA. The University of California, Santa Barbara is a member of the OMG.

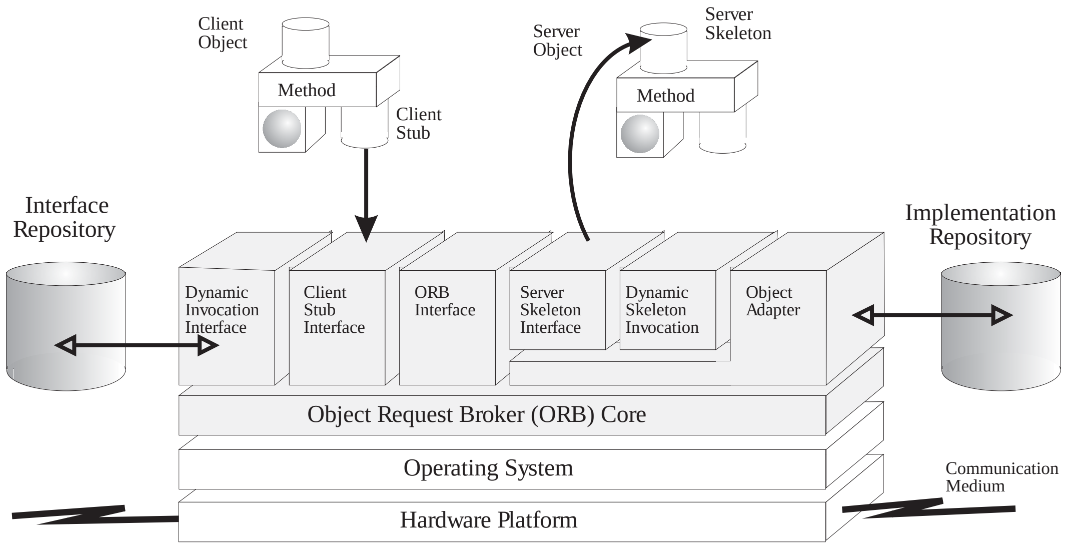


Figure 1.1: Common Object Request Broker Architecture.

standard has grown in popularity and in its ability to meet the needs of commercial and critical distributed object applications. The same set of standard specifications for CORBA is translated into implementations by different vendors onto diverse operating systems and platforms.

CORBA uses a purely declarative language, the OMG Interface Definition Language (IDL), to define interfaces to objects. The IDL interfaces are subsequently mapped, through an IDL compiler provided by an implementor of CORBA, onto specific programming languages. These IDL compilers respect the OMG-standardized IDL-to-language mappings for C, C++, Java, Smalltalk, etc. A CORBA client object wishing to make use of the services of a CORBA server object needs to know only the server object's IDL interface, and never the programming-language-specific implementation of the server object. This has the advantage that the code for the client and server objects need not be written in the same programming language.

CORBA applications can transcend the various sources of heterogeneity and incompatibility that can arise in typical distributed applications. CORBA's language transparency implies that client objects need to be aware of only the IDL interface, and never of the language-specific implementation, of a server object. CORBA's interoperability implies that a client object can interact with a server object, despite differences in platforms and operating systems on which the client and the server objects are hosted. CORBA's location transparency implies that client objects can invoke server objects, without worrying about the actual locations of the server objects.

As shown in Figure 1.1, the key component of the CORBA model is the Object Request Broker (ORB), which acts as an intermediary between the client and the server objects, and shields them from differences in platform, programming language and location. The CORBA application developer codes the interface of every application object in CORBA IDL, and an IDL compiler transforms the server object's interface description into a language-specific *client-side stub* and a *server-side skeleton*. The stub is compiled into the client's code, and the skeleton is compiled into the server's code.

When a client object invokes an operation on a server object, the request passes through the server object's stub on the client side. Before passing the request onto the ORB, the stub marshals the request into the format appropriate for the operation. Instead of using the IDL stub (which represents a "static" invocation interface), a client object may use the Dynamic Invocation Interface (DII) to select the object type and operation at runtime. The Interface Repository serves as a storehouse for the interface definitions of all the objects in the distributed system. Its role is crucial to dynamic invocation because it allows the client object to request the ORB to retrieve the target server object's IDL interface.

Upon receiving the client object's request through the stub, the client-side ORB locates the server object, establishes a connection to the server object, and dispatches the request to the server-side ORB. The server-side ORB consults the Implementation Repository to determine if the server object is currently operational. If not, the server-side ORB, by means of the Portable Object Adapter (POA), activates the server and prepares it to receive requests. The Portable Object Adapter then uses the server object's skeleton on the server side to pass the invocation onto the server object. After the server object performs the operation, it returns a response, which the server-side ORB routes back to the client that invoked the operation. It is possible for a CORBA object to assume simultaneously a client role for one operation, and a server role for a different operation. By handling the discovery of servers, as well as the routing of requests, on behalf of client objects, the ORB provides for location transparency.

CORBA also encompasses a rich suite of basic services that almost every application requires. CORBA's Common Object Services [51] include Naming, Property, Event and Lifecycle services, each of which is also specified in terms of standard IDL interfaces, and implementations of which are intended to be provided by ORB vendors for use by CORBA application programmers. This frees the CORBA application developer from the burden of having to design and to write the code for such commonly-used functionalities.

One of the aims of CORBA is to promote the interworking of objects across heterogeneous platforms, transcending differences in hardware architectures, operating systems or programming languages. CORBA makes this possible through its interoperability specifications.

1.1.1 Interoperability with IIOP

The CORBA standard's General Internet Inter-ORB Protocol (GIOP) is a set of specifications for mapping the messages exchanged by objects, through the ORB, onto any transport protocol that meets a minimal set of requirements. The GIOP interface is compact, and consists of a set of well-defined message formats for inter-object communication – `Request`, `Reply`, `CancelRequest`, `CancelReply`, `LocateRequest`, `LocateReply`, `MessageError`, `Fragment` and `CloseConnection`. The transport protocol onto which GIOP is mapped must be connection-oriented, reliable and byte-stream-oriented, and must provide notification of connection loss.

The Internet Inter-ORB Protocol (IIOP) is the concrete OMG-standardized mapping of the GIOP specifications onto the TCP/IP protocol suite. Every conformant implementation of CORBA must include support for IIOP, regardless of the hardware or software platform of choice. As additional support for interoperability, CORBA mandates that every CORBA object be uniquely identified in a platform-independent and vendor-independent way that can be interpreted by every CORBA-conformant ORB. This identification takes the form of a stringified handle, known as an Interoperable Object Reference (IOR). At a minimum, an object's IOR contains information about the host and the port on which the object is prepared to receive requests.

Because the structure of an IOR is prescribed in the CORBA standard, an IOR generated by one ORB can be interpreted by a CORBA-conformant ORB from a different vendor. Thus, a client object, having obtained the IOR of a server object, can use its ORB to retrieve a reference to the server object from the IOR, even if the client and the server objects use different ORBs. The client can then invoke the server using this reference, provided that the client-side and server-side ORBs communicate over IIOP. The coupling of IORs with IIOP allows objects hosted by different ORBs to interact.

Thus, the role of IIOP is crucial to CORBA because it provides a single protocol standard to which all conformant implementations of CORBA must adhere, and also because it enables CORBA applications hosted over IIOP-enabled ORBs to transcend differences in platform, hardware architecture and vendor-specific mechanisms. The attractiveness of IIOP manifests itself not only in the interoperability of different ORBs, but also in its adoption by other enterprise applications. For instance, the latest release of the Java Development Kit (JDK 1.2) contains a standard extension called RMI-IIOP [67] to enable the interworking of Java's Remote Method Invocation (RMI) facilities with CORBA applications. Such non-CORBA applications can avail themselves of the services of CORBA objects within a distributed system, merely by being able to "talk" IIOP.

1.1.2 What Does CORBA Lack?

Currently, CORBA applications cannot interact with distributed objects communicating over protocols that do not conform with the GIOP specifications. However, if CORBA objects are, in fact, required to communicate with applications that use a different protocol, it is left to the ORB developer to rewrite the transport-level mappings of the ORB, or to build ORB-level mechanisms to enable "pluggable protocols" [30, 55]. In either case, access to, and considerable modification of, the source code of the implementation of CORBA is required to provide the support for adding a new protocol. Furthermore, an ORB that has been modified in this way may no longer conform to the CORBA standard. Instead, it is desirable to map the IIOP messages of CORBA objects, in a transparent manner and without modification to the ORB, onto different protocols for interoperation with non-IIOP-enabled systems. Such mechanisms would enable legacy systems using protocols other than IIOP to take advantage of CORBA's other facilities, without requiring the legacy applications to be modified. Furthermore, the CORBA standard itself may not need to be modified if the protocol adaptation were transparent.

Another deficiency in the CORBA standard is its lack of debugging or profiling mechanisms or services. Nevertheless, some of the commercial ORBs are integrated with vendor-specific tools to enable CORBA programmers to "watch" some of the ORB's message-level interactions, *e.g.*, the smart proxy and filter mechanisms provided with Iona Technologies's Orbix [23]. However, to exploit these tools, the application developer must understand the mechanisms and must incorporate the required "hooks" for them within the application code. Furthermore, these mechanisms provide only for limited and specific types of probing, and cannot be easily extended or customized as required by the application. Instead, CORBA should be extended to provide profiling tools that would allow an application and its messages to be transparently traced, without the need to embed any special code into the application. Such a facility could serve not only for debugging, but also for monitoring the system and for measuring the system performance.

Although the current CORBA standard supports location transparency, language transparency and interoperability, it makes no provision for other desirable features such as fault tolerance. The fault detection mechanisms that currently exist in CORBA are rudimentary, and mostly consist of returning either system or user-defined exceptions if an object or a processor "dies." Of course, some commercial ORBs provide primitive support for the fail-over of stateless server objects, *e.g.*, the smart agent mechanism of Inprise Corporation's VisiBroker ORB [22]. However, it is left to the CORBA application programmer to implement consistent fault tolerance and recovery.

Today's applications are sufficiently complex in themselves. It is therefore undesirable to embed fault tolerance into the application, which would increase the application complexity and the application development timescale. Instead, the difficulties of providing fault tolerance, recovery and consistency should be handled transparently and should *not* be exposed to the application programmer, who is not necessarily experienced in dealing with such issues. Transparent fault tolerance for CORBA has the advantages of allowing the CORBA application programmer to focus on the application logic, and to leave the fault tolerance to experts. This would result in the speedier development and deployment of fault-tolerant CORBA applications, with a lower risk of getting the fault tolerance technology wrong.

The Object Management Group (OMG) has recognized the need to provide fault tolerance for CORBA applications by issuing a Request for Proposals (RFP) [52] for building fault-tolerant CORBA applications through the use of *entity redundancy.* One of the OMG's objectives is that the augmentation of CORBA with fault tolerance should impact the existing CORBA standard minimally. Another requirement of the OMG's RFP is that *strong replica consistency* must be maintained as operations are performed that change the states of the replicated entities.

1.2 Fault Tolerance for CORBA

Distributed object applications can be made fault-tolerant by replicating their constituent objects, and distributing these replicas across different computers in the network. The idea behind object replication is that the failure of one replica (or of a processor hosting a replica) of an object can be masked from a client of the object because the other replicas can continue to perform any operation that the client requires of the object.

A significant body of work exists in the area of fault-tolerant distributed object systems. These include systems that are not based on CORBA, such as Arjuna [56], FRIENDS [15], FilterFresh [5] (for Sun Microsystems' Java RMI model) and COMERA [72] (for Microsoft's DCOM model). However, different approaches [18, 46] have been developed for providing reliability specifically for applications based on the CORBA standard. These approaches are alike in their use of object replication to provide fault tolerance. However, the approaches differ in a number of aspects – the degree of transparency to the CORBA application, the degree of modification to the CORBA ORB, the specific mechanisms for achieving replica consistency, and the level of replica consistency provided.

Initial efforts to enhance CORBA with fault tolerance took an integration approach, with the fault tolerance mechanisms embedded *within* the ORB itself. With the advent of Common Object Services in the CORBA standard, other research efforts have taken a service approach, with the provision of fault tolerance through service objects *above* the ORB. The novel interception approach that we have developed allows the transparent insertion of fault tolerance mechanisms *underneath* the ORB. Interception achieves the best of the integration and the service approaches, while providing other benefits as well.

1.2.1 Integration Approach

The integration approach to providing new functionality to CORBA applications involves modifying the ORB to provide the necessary support. The extent of the modification to the ORB depends on the functionality that is being added, with the likelihood that the resulting modified ORB is non-compliant with the CORBA standard. However, because the mechanisms form an intrinsic part of the ORB, the new functionality can be made available in a way that is transparent to the application.

Thus, an integration approach to providing fault tolerance for CORBA implies that the replication of server objects can be made transparent to the client objects because the fault tolerance mechanisms are integrated into the ORB. Furthermore, the details of the replica consistency mechanisms are buried within the ORB, and can be hidden from the application programmer.

1.2.1.1 Electra

Developed at the University of Zurich, Electra [36, 37] is a fault-tolerant ORB that exploits the reliable totally ordered group communication mechanisms of the Horus toolkit [69] to maintain replica consistency. As shown in Figure 1.2(a), adaptor objects linked into the ORB and into the applications convert the ORB's messages into the multicast messages of the underlying Horus toolkit. In Electra, the Basic Object Adapter (that has been replaced with the Portable Object Adapter in CORBA 2.x) of the CORBA 1.x standard is enhanced with mechanisms for creating and removing replicas of a server object, and for transferring the state to a new server replica.

With Electra's use of the integration approach, a CORBA client hosted by Electra can invoke a replicated server object just as it would invoke a single server object, without having to worry about the location, number, or even the existence, of the server replicas.

Unfortunately, as a side-effect of the integration approach, it is not possible for any commercial off-the-shelf implementation of CORBA to exploit the technology of Electra without undergoing significant modification to the transport-level mappings of the ORB.

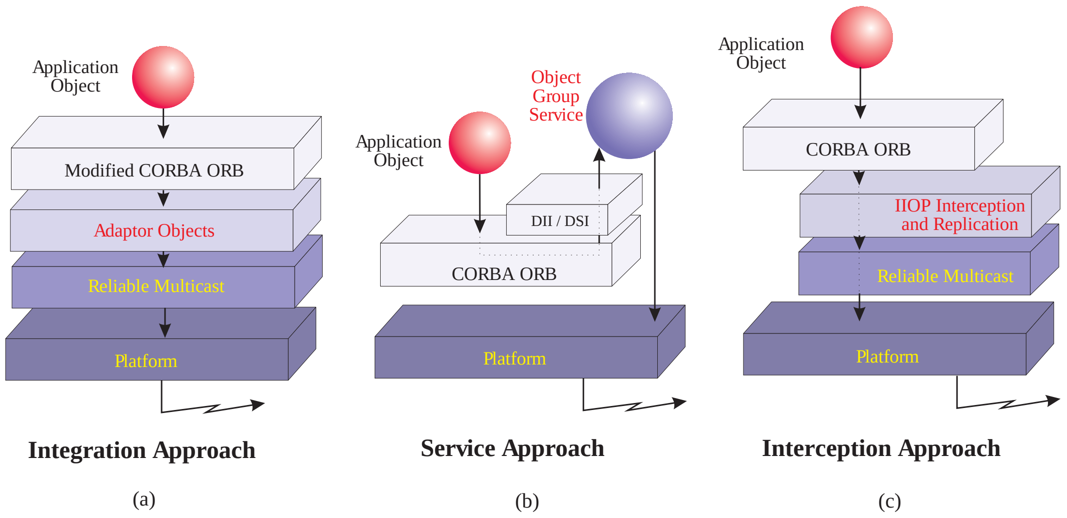


Figure 1.2: Different approaches to fault-tolerant CORBA.

1.2.1.2 Orbix+Isis

Developed by Iona Technologies, Orbix+Isis [24] was the first commercial offering in the way of fault-tolerant CORBA. Like Electra, Orbix+Isis involves significant modification to the internals of the ORB to accomodate the use of the Isis toolkit [7] from Isis Distributed Systems for the reliable ordered multicast of messages.

The implementation of a CORBA server object must explicitly inherit from a base class. Two types of base classes are provided – an Active Replica base class that provides support for active replication and hot passive replication, and an Event Stream base class that provides support for publish/subscribe applications.

The replication of server objects can be made transparent to the client objects. Orbix-specific smart proxies can be used on the client side to collect the responses from the replicated server object, and to use some policy (deliver first response, vote on all responses, *etc*) to deliver a single response to the client object.

1.2.1.3 Maestro Replicated Updates ORB

Developed at Cornell University, Maestro [70] is a CORBA-like implementation of a distributed object layer that supports IIOP communication and that exploits the Ensemble group communication system [68]. The ORB is replaced by an IIOP Dispatcher and multiple request managers that are configured with different message dispatching policies. One of these request managers, the Replicated Updates request manager, supports the active replication of server objects. “Smart” clients have access to compound IIOP object references (also known as compound IORs) consisting of the enumeration of the IIOP profiles of the replicas of a server object. A “smart” client connects to a single server replica and, in the event that this replica fails, can re-connect to one of the other server replicas using the information in the compound IOR.

In the typical operation of Maestro, a client object running over a commercial ORB uses IIOP to access a single Maestro-hosted server replica, which then propagates the client's request to the other server replicas through the messages of the underlying Ensemble system. However, the server code must be modified to use the facilities that the request managers of Maestro provide. Maestro's emphasis is on the use of IIOP and on providing support for interworking with non-CORBA legacy applications, rather than on strict adherence to the CORBA standard. Thus, Maestro's replicated updates execution style can be used to add reliability and high availability to client/server CORBA applications in settings where it is not feasible to make modifications at the client side.

Electra and Maestro support the replication of server objects only, and provide no mechanisms for the replication of client objects. Furthermore, both systems allow only for the active replication style. No mechanisms exist within either Electra or Maestro for the detection of duplicate messages that arise due to active replication and that might corrupt the states of objects if not suppressed.

1.2.1.4 The AQuA Framework

Developed jointly by the University of Illinois at Urbana-Champaign and BBN Technologies, AQuA [10] is a framework for building fault-tolerant CORBA applications. AQuA employs the Ensemble/Maestro [68, 70] toolkits, and comprises the Quality Objects (QuO) runtime, and the Proteus dependability property manager [60]. Based on the user's QoS requirements communicated by the QuO runtime, Proteus determines the type of faults to tolerate, the replication policy, the degree of replication, the type of voting to use and the location of the replicas, and dynamically modifies the configuration to meet those requirements. The AQuA gateway translates a client's (server's) invocations (responses) into messages that are transmitted via Ensemble; the gateway also detects and filters duplicate invocations (responses). The gateway handlers contain monitors, which detect timing faults, and voters, which either accept the first invocation/response or perform majority voting on the invocations/responses from the object replicas.

AQuA provides mechanisms for majority voting at the application object level, to detect an incorrect value of an invocation (response) from a replicated client (server). However, in order for majority voting to be effective for applications that must tolerate arbitrary faults, more stringent guarantees are required of the underlying multicast protocols. Because AQuA uses the underlying Ensemble group communication system, which tolerates only crash faults, AQuA does not provide for the detection, or tolerance, of processor commission faults or message corruption faults.

1.2.2 The Service Approach

The service approach to extending CORBA with new functionality involves providing the enhancements through a new service, along the lines of the existing Common Object Services [51] that form a part of the CORBA standard. Because the new functionality is provided through a collection of CORBA objects entirely above the ORB, the ORB does not need to be modified and the approach is CORBA-compliant. However, to take advantage of the new service, the CORBA application objects need to be explicitly aware of the service objects. Thus, it is likely that application code requires modification to exploit the functionality of the new CORBA service.

Using this approach, fault tolerance can be provided as a part of the suite of CORBA Services. Of course, because the objects that provide reliability reside above the ORB, every interaction with these objects must necessarily pass through the ORB, and will thus incur the associated performance overheads.

1.2.2.1 Distributed Object-Oriented Reliable Service (DOORS)

The Distributed Object-Oriented Reliable Service (DOORS) [63] developed at Lucent Technologies adds support for fault tolerance to CORBA by providing replica management, fault detection, and fault recovery as service objects above the ORB. DOORS focuses on passive replication and is not based on group communication and virtual synchrony. It also allows the application designer to select the replication style (cold passive and warm passive replication), degree of reliability, detection mechanisms and recovery strategy.

DOORS consists of a WatchDog, a SuperWatchDog and a ReplicaManager. The WatchDog runs on every host in the system and detects crashed and hung objects on that host, and also performs local recovery actions. The centralized SuperWatchDog detects crashed and hung hosts by receiving heartbeats from the WatchDogs. The centralized ReplicaManager manages the initial placement and activation of the replicas and controls the migration of replicas during object failures. The ReplicaManager maintains a repository that contains, for each object in the system, the number of replicas, the hosts on which they are running, the status of each replica and the number of faults seen by the replica on a given host. This repository, which forms part of the state of the ReplicaManager, is periodically checkpointed. DOORS employs libraries for the transparent checkpointing [71] of applications; however, duplicate detection and suppression are not addressed.

DoorMan is a management interface to DOORS that monitors DOORS and the underlying system in order to fine-tune the functioning of DOORS and to take corrective action by migrating objects whose hosts are suspected of being faulty and about to crash. DoorMan collects and displays data about DOORS, and provides feedback information to DOORS about the underlying computing platform, to improve its decision making.

1.2.2.2 Object Group Service (OGS)

Developed at the Swiss Federal Institute of Technology at Lausanne, the Object Group Service (OGS) [17, 18, 19] consists of service objects implemented above the ORB that interact with the objects of a CORBA application to provide fault tolerance to the application.

OGS is comprised of a number of sub-services implemented on top of the commercial ORBs, Orbix and VisiBroker. Each of these sub-services is independent and is itself implemented as a collection of CORBA objects. Messaging, multicast, monitoring and consensus are sub-services, with interfaces specified using OMG IDL. These sub-services together provide support for consistent object replication.

The multicast sub-service provides for the reliable unordered multicast of messages destined for the replicas of a target server object. The messaging sub-service provides the low-level mechanisms for mapping these messages onto the transport layer. The consensus sub-service imposes a total order on the multicast messages, while the monitoring sub-service detects crashed objects.

To exploit the facilities of the OGS objects, the replicas of a server object must inherit from a common IDL interface that permits them to join or leave the group of server replicas. Thus, in order to be replicated, the server objects must be modified. This interface also provides methods that allow the OGS objects to transfer the state of the replicated server objects, as needed, to ensure replica consistency.

OGS provides a client object with a local proxy for each replicated server with which the client communicates. The server's proxy on the client side and the OGS objects on the server side are together responsible for the mapping of client requests and server responses onto multicast messages that convey the client's request to the server replicas. The client establishes communication with the replicas of a server object by binding to an identifier that designates the object group representing all of the server replicas. The client can then direct its requests to the replicated server object using this object group identifier. Once a client is bound to a server's object group, it can invoke the replicated server object as if it were invoking a single unreplicated server object. However, because the client is aware of the existence of the server replicas, and can even obtain information about the server object group, the replication of the server is not necessarily transparent to the client. Also, with this approach, a CORBA client needs to be modified to bind, and to dispatch its requests, to a replicated CORBA server.

Because the client and the server objects must explicitly employ the service objects, the replication is no longer transparent to the application. Furthermore, the OGS objects use the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI) of CORBA, both of which adversely impact the performance of the application.

1.2.2.3 Newtop Object Group Service

Developed at the University of Newcastle, the Newtop [39] service provides fault tolerance to CORBA using the service approach. While the fundamental ideas are similar to OGS described in Section 1.2.2.2, Newtop has some key differences.

Newtop allows objects to belong to multiple object groups. Of particular interest is the way the Newtop service handles failures due to partitioning – support is provided for a group of replicas to be partitioned into multiple sub-groups, with each sub-group being connected within itself. Total ordering continues to be preserved within each sub-group. No mechanisms are provided, however, to ensure consistent remerging of the sub-groups once communication is reestablished between them.

1.2.3 The Interception Approach

The interception approach involves "capturing" specific system calls or library routines used by the application, and modifying their call parameters or return values, or even the calls and routines themselves, to alter the behavior of the application, or to enhance the application with a new and different functionality.

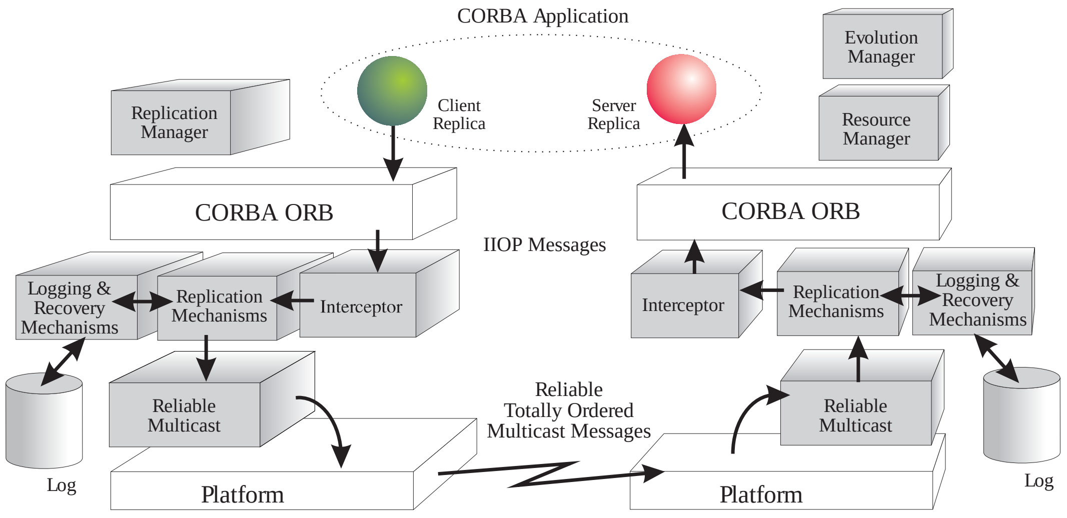


Figure 1.3: Structure of the Eternal system.

The advantages of this approach are that neither the ORB nor the objects are ever aware of being "intercepted" and, thus, the new functionality is provided to the application in a manner that is transparent both to the application and to the ORB. Thus, modification of, or even access to, the source code of the ORB is not required. Moreover, the CORBA application does not need to be modified or recompiled.

1.2.3.1 The Eternal System

The Eternal system [42, 45, 47, 48] that we have developed at the University of California, Santa Barbara, exploits the interception approach to provide fault tolerance for applications running over commercial off-the-shelf implementations of CORBA. The mechanisms implemented in different parts of the Eternal system work together efficiently to provide strong replica consistency with low overheads, and without requiring the modification of either the application or the ORB.

In the Eternal system, the client and server objects of the CORBA application are replicated, and the replicas are distributed across the system. Different replication styles – active, cold passive, warm passive and hot passive replication – of both client and server objects are supported. To facilitate replica consistency, the Eternal system conveys the IIOP messages of the CORBA application using the reliable totally ordered multicast messages of the underlying Totem system [1, 40], also developed at the University of California, Santa Barbara.

The structure of the Eternal system is shown in Figure 1.3. The Eternal Replication Manager replicates each application object, according to user-specified fault tolerance properties (such as the replication style, the checkpointing interval, the fault monitoring interval, the initial number of replicas, the minimum number of replicas, *etc.*) and distributes the replicas across the system. The Eternal Resource Manager monitors the system resources, and maintains the initial and the minimum number of replicas.

The Eternal Interceptor captures the IIOP messages (containing the client's requests and the server's replies), which are intended for TCP/IP, and diverts them instead to the Eternal Replication Mechanisms for multicasting via Totem. The Eternal Replication Mechanisms, together with the Eternal Logging-Recovery Mechanisms, maintain strong consistency of the replicas, detect and recover from faults, and sustain operation in all components of a partitioned system, should a partition occur.

The Eternal Evolution Manager exploits object replication to support upgrades to the CORBA application objects. The Replication Manager, the Resource Manager and the Evolution Manager are themselves implemented as collections of CORBA objects and, thus, can benefit from Eternal's fault tolerance.

The types of faults tolerated by Eternal, and the underlying Totem system, are communication faults, including message loss and network partitioning, and processor, process, and object faults. Eternal can also tolerate arbitrary faults by exploiting protocols such as SecureRing [28], also developed at the University of California, Santa Barbara, with more stringent guarantees than are provided by Totem. To tolerate value faults in the application, Eternal uses active replication with majority voting [44] applied on both invocations and responses for every application object.

The technology of Eternal formed the basis of our response [13], in October 1998, to the Object Management Group's Request for Proposals on fault-tolerant CORBA. With our close involvement in the ongoing OMG standardization process, it appears likely that the technology of the Eternal system will form the basis of the forthcoming CORBA standard for fault tolerance that is due in January 2000.

Chapter 2

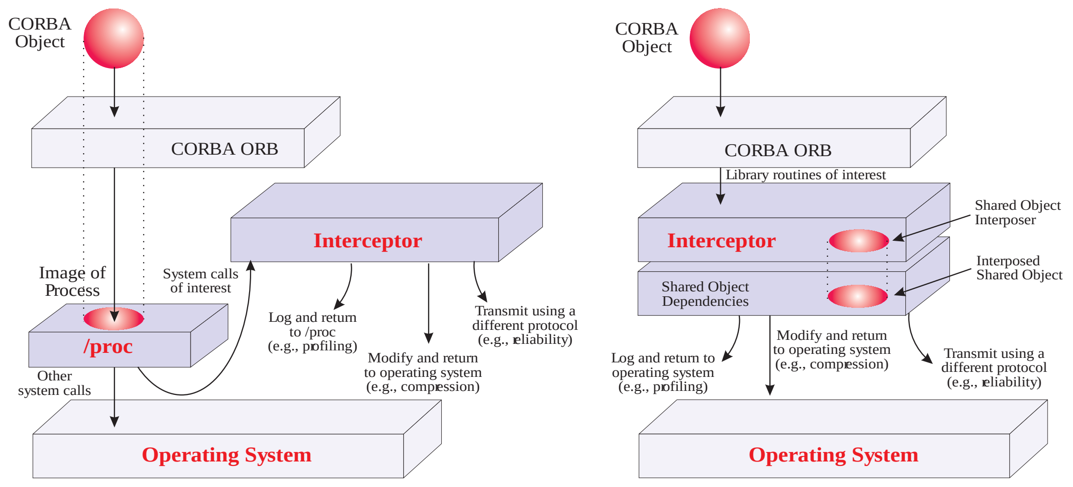
Interception

CORBA currently lacks support for the use of alternative protocols, for the profiling and monitoring of application objects, and for security and fault tolerance. To equip the application with these additional features, the CORBA application programmer must either build, or be able to use, the code that provides these features. Building such specialized code requires the application programmer not only to invest substantial effort, but also to worry about issues that are essentially outside the application domain. Using existing code that provides these features, while involving less effort than building them, also requires some level of understanding to manage the interaction of those features with the CORBA application and the ORB.

An *interceptor* [50] is an entity that can alter the processing of requests and responses, and the behavior of activities that are under the control of the ORB or the CORBA application. Through the use of interceptors, it is possible to enhance CORBA applications, at run-time, with additional features in a manner that is transparent to (and thus requires no modification of) either the application or the ORB. The OMG has also recognized the value of augmenting the CORBA standard with portable ORB-level interceptors [53] that can be inserted at different points in the ORB's processing.

2.1 Interceptors for CORBA

In the context of the Eternal system, an interceptor is a non-ORB-level, non-application-level entity (a process or a shared library, depending on the specific implementation of interception). The interceptor transparently "attaches" itself to every executing CORBA object, without the object's or the ORB's knowledge, and is capable of modifying the object's behavior as desired, at run-time. The advantage of this kind of interceptor, over ORB-level interceptors, lies not only in its transparency both to the ORB and to the application, but also in the possibility of its implementation in an ORB-independent manner. For the remainder of this dissertation, the interception of CORBA application objects refers to the


CORBA Object
CORBA ORB
Interceptor
Image of Process
System calls of interest
/proc
Other system calls
Log and return to /proc (e.g., profiling)
Modify and return to operating system (e.g., compression)
Transmit using a different protocol (e.g., reliability)
Operating System
CORBA Object
CORBA ORB
Library routines of interest
Interceptor
Shared Object Interposer
Shared Object Dependencies
Interposed Shared Object
Log and return to operating system (e.g., profiling)
Modify and return to operating system (e.g., compression)
Transmit using a different protocol (e.g., reliability)
Operating System

(a) (b)

Figure 2.1: Possible implementations of an interceptor for CORBA as (a) a separate process using the */proc*-based approach, and (b) a shared library using the library interpositioning approach.

use of non-ORB-level interceptors, such as the Interceptor of the Eternal system shown in Figure 1.3.

Current operating systems provide “hooks” that can be exploited to develop interceptors. With the Unix operating system, at least two possible implementations of interceptors exist. The first of these approaches, the */proc*-based implementation, provides for interception at the level of system calls, and results in an interceptor that is a separate process. The second approach, the library interpositioning implementation, provides for interception at the level of library routines, and results in an interceptor that is a shared library. While the techniques may differ, the intent and the use of the interceptor in both cases is identical, and requires no modification of the intercepted CORBA objects, the ORB or the operating system.

The specific system calls to intercept in a */proc*-based implementation or the specific library routines to redefine in a library-interpositioning implementation, depends on the extent of the information that the interceptor must extract (from the ORB or the CORBA application) to enhance the application with new features. The interceptor may capture all, or a particular subset, of the system calls or library routines used by the CORBA application, depending on the feature being added.

2.1.1 Implementation of Interceptors

2.1.1.1 System-Call Interception

The mechanisms underlying the */proc*-based implementation have been developed in the context of global file systems [2, 3], and serve to extend the functionality of standard oper-

```
sysset_t sysCallsToCatch;
  if ((process = open(“/proc/08011”, O_RDWR)) == -1) {
   /* Process cannot be intercepted */
   exit(1);
}

/* Initializes the set of system calls to catch*/
premptyset(&sysCallsToCatch);

/* Specifies the system calls to intercept */
praddset(&sysCallsToCatch, SYS_poll); /* Catch the poll() system call
praddset(&sysCallsToCatch, SYS_stat); /* Catch the stat() system call

/* Enables the specified system calls to be caught on entry to the call,
i.e., before the system call is processed. */
ioctl(process, PIOCSENTRY, &sysCallsToCatch);

/* Enables the specified system calls to be caught on exit from the call,
i.e., after the system call has been processed. */
ioctl(process, PIOCSEXIT, &sysCallsToCatch);

/* The interceptor runs forever, executing an PIOCWSTOP ioctl
to wait for the process to stop on a specified system call, and a
PIOCRUN ioctl, to release the process after handling the system call. */
        :
        :
```

Figure 2.2: Snippet of C code for a */proc*-based interceptor that is designed to catch the `poll()` and `stat()` system calls of a process.

ating systems at the user level. An interception layer can transparently “attach” itself to an executing process, in our case, a CORBA client or server, in order to monitor and control its behavior. The client or server does *not* need to be modified or recompiled to exploit this approach.

In the Unix System V operating system, the */proc* local file system [16] on each computer provides access to the image of each process currently hosted by that computer. The */proc* interface was originally introduced for debugging purposes. Debugging utilities such as **`truss`** on Solaris 2.x are examples of commercial */proc*-based interceptors.

Each entry in the */proc* directory is a file whose name corresponds to the Unix process-ID of a current process on the computer. For instance, on Solaris 2.x, a process executing with process-ID 8011 would have the numerical filename entry */proc*/08011 inside the */proc* file system. The files in the */proc* file system, and thus the processes that they represent, can be manipulated via a standard interface.

This standard interface, consisting of the `open()`, `close()`, `read()`, `write()` and `ioctl()` system calls, and the `praddset()`, `prfillset()`, `prdelset()` and the `premptyset()` macros, allows the image of each process to be accessed for either process monitoring or process control. An `open()` for reading and writing enables process control; a read-only `open()` allows inspection but not control. The tracing mechanism is enabled by specifying, through this interface, the system calls to “catch” for each process. Figure 2.2 shows a snippet of the in-

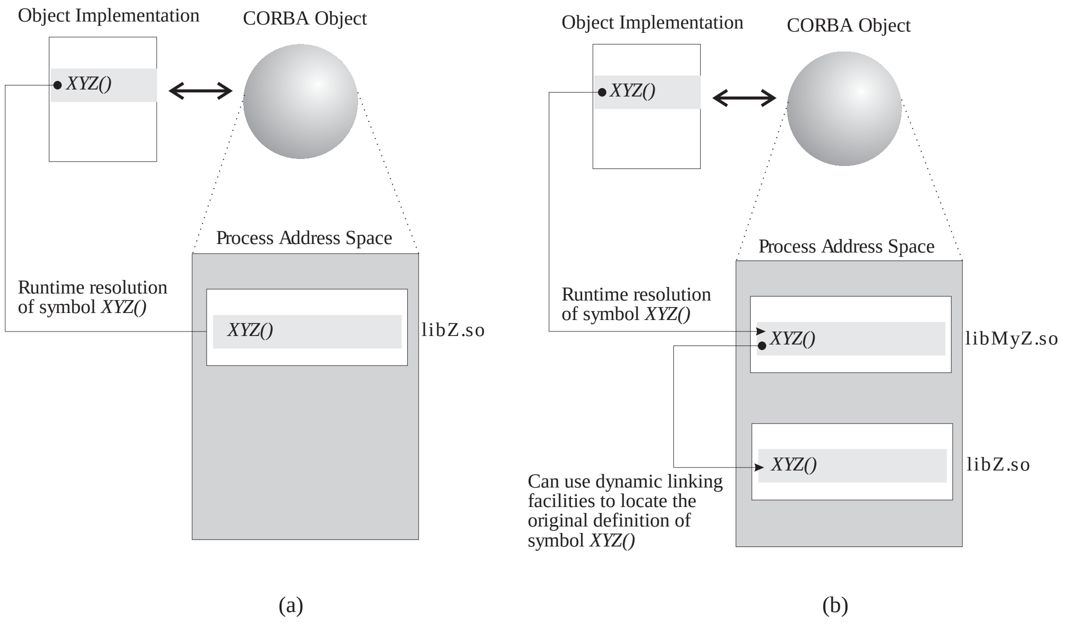


Figure 2.3: Resolution of a symbol at runtime (a) without library interpositioning, and (b) using library interpositioning to provide an alternative symbol definition, as well as access to the original symbol definition.

terceptor code that uses the */proc* interface routines to specify that the `poll()` and `stat()` system calls are to be intercepted for a process with process-ID 8011. The arguments and the return values of the intercepted system calls can be extracted, and can even be modified. By appropriate "patching" of the arguments of these system calls, or even the function of the system call itself, the behavior of the intercepted process can be altered.

As shown in Figure 2.1(a), the */proc*-based implementation of the Eternal system's Interceptor views each CORBA client or server as a process that can be controlled via the facilities of the */proc* interface. Thus, Eternal's Interceptor can monitor each CORBA object for the lifetime of the object, for system calls related to memory management, network communication or file access. For CORBA applications, the interactions between distributed objects are of the greatest interest to Eternal, and, thus, the Interceptor is designed to "watch" for specific system calls made by CORBA objects when they communicate over IIOP.

2.1.1.2 Library-Routine Interpositioning

Instead of examining a process's behavior at the granularity of system calls, the library interpositioning implementation of an interceptor can examine and modify a process's behavior at the granularity of library routines. The library interpositioning implementation exploits the operating system's runtime linker-loader facilities [11, 34, 66] that allow shared libraries to be loaded into a process's address space, at run-time, as a part of the process's initialization. These additional shared libraries can be mapped into a process's address space *after* the binary executable (that the process represents) is loaded into memory, and *before* any of the compile-time shared library dependencies can be mapped into the process's space, as shown in Figure 2.1(b). The runtime linker achieves this effect with *no* modification, *no* relinking and *no* recompilation of the application source code, and requires access only to the binary executables of the application.

Library interpositioning exploits the fact that a compiled binary can have dynamic library symbols that remain intentionally unresolved until run-time. At run-time, the first shared library in the process's address space that resolves a symbol (*i.e.*, provides a definition for the symbol) becomes the accepted source for that symbol's definition for the lifetime of the process. If subsequent shared libraries within the same process's address space also provide definitions of the same symbol, the first accepted symbol definition *interposes* on, or hides, all of the definitions that follow for the same symbol.

Figure 2.3 shows the implementation of a CORBA object that requires access to the function *XYZ()*. In the normal course of events, as shown in Figure 2.3(a), the definition for the symbol *XYZ()* is resolved at runtime from the standard library `libZ.so`. Figure 2.3(b) shows a custom library interposer `libMyZ.so` that also contains a definition for the symbol *XYZ()*. When `libMyZ.so` is inserted into the process's address space ahead of `libZ.so`, the definition of the symbol *XYZ()* as provided by the interposer is accepted ahead of the default definition. The default definition of the symbol can still be located, and invoked (if needed), by the library interposer using the dynamic linking facilities.

Thus, the definition of a symbol within a library interposer (*i.e.*, a library that is inserted into the process's address space ahead of all other shared libraries) is accepted by the process as the symbol's default definition. The trick, then, is to insert custom or desired definitions of symbols of interest into a library interposer, and then to let the runtime linker do its job. The dynamic linked libraries (DLLs) of the Windows NT operating system provide similar hooks that can be exploited to build interceptors [4].

The only requirement for the use of library interpositioning is that the application must be dynamically linked to the shared libraries that we are interested in interposing. The library interposer need not completely replace the library of interest, and can provide alternative definitions for only those library functions or system calls of interest to the Interceptor. Through the extensive support for dynamic linking and libraries provided by the Unix operating system, it is also possible for a function interposer to use the dynamic linking facilities to find the address of, and invoke, the real definition of the function that it replaces. This is very useful in adding to, or enhancing, a library routine without essentially altering its functionality, *e.g.*, in building profilers, debuggers and IIOP message parsers for unmodified CORBA applications.

2.1.1.3 Comparison

With the library interpositioning technique, the granularity of interception or of process modification is the library routine (and not the system call as it is in the */proc* interception case). This greatly reduces needless interception, and, thus, results in less overhead.

For instance, in Unix, a TCP/IP connection is opened with the `socket()` library routine (defined in `libsocket.so`), while a file is opened with `fopen()` (defined in `libc.so`).

However, both `socket()` and `fopen()` ultimately invoke the `open` system call on a file, with the `socket()` case involving the special "device filename" */dev/tcp*, and the `fopen()` case involving a regular filename in the file system. Thus, the interception of the `open` system call will catch both types of files being opened. However, if the intent of interception is to capture only TCP/IP communication, the interception of the `open()` system call leads to more undesired interceptions (because of the additional intercepted accesses to regular files). Instead, if interpositioning on the `socket()` library routine were used, regular file system accesses would effectively never be "seen".

Another benefit of library interpositioning is that a library interposer for one ORB can be readily used with other ORBs. This is primarily because the point of interception is at the level of library routines, which are more or less implemented in the same way across different operating systems. System calls, on the other hand, tend to be very closely related to the operating system, and are thus kernel-specific and largely undocumented!

For instance, to communicate over TCP/IP, all CORBA objects must ultimately use sockets through the routines of a standard socket library. The socket library `libsocket.so` is, for the most part, generic, well-documented and similar across many variants of the Unix operating system, although their implementations in terms of system calls may vary. Thus, by making the library routines, rather than system calls, the point of interception, the interception code does not need to be modified for different commercial ORBs. Section 8.1 describes some of the particular difficulties in building interceptors for different commercial ORBs.

With the */proc* interception technique, the interceptor is required to be implemented as a separate process under whose control all CORBA application objects must be launched for interception to be effected. After launching a CORBA object, the interceptor "attaches" itself to the process containing the CORBA object through the */proc* interface, and then proceeds to intercept the system calls of interest. With the library interpositioning technique, however, because the interceptor is implemented as one of the shared libraries inserted into the process space of a CORBA object, the interceptor is not a separate process in its own right and, thus, the overhead of context switching (between the interceptor and the intercepted application) is greatly reduced.

Of course, there are situations where a */proc*-based interceptor is preferred over the library-interpositioning interceptor. This is particularly the case when commercial ORBs use proprietary libraries whose API is not documented or easily inferred. Furthermore, if CORBA applications employ static linking, rather than dynamic linking, of libraries, it is infeasible to use the library interpositioning approach. Only the */proc*-based interceptor can be used in such cases because all libraries (whether proprietary, statically linked, or dynamically linked) must necessarily use system calls to communicate with the operating system. Thus, the system calls invoked by the application code form an easier point of interception than the unknown or statically linked library routines.

2.2 Interceptors for Fault-Tolerant CORBA

Perhaps the most striking use of interceptors is in their enhancement of CORBA with fault tolerance, through the addition of multiple features, including those for message parsing,

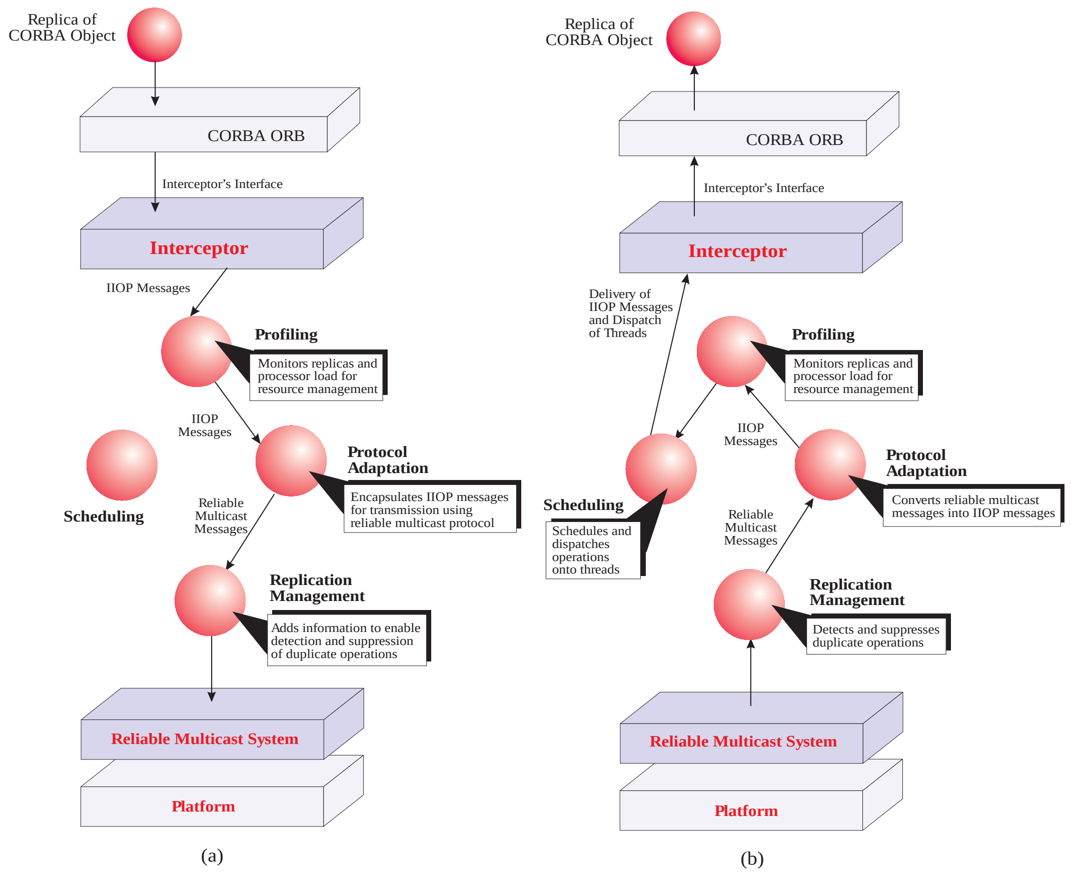

Replica of CORBA Object
CORBA ORB
Interceptor's Interface
Interceptor
IIOP Messages
Profiling
Monitors replicas and processor load for resource management
IIOP Messages
Protocol Adaptation
Encapsulates IIOP messages for transmission using reliable multicast protocol
Scheduling
Reliable Multicast Messages
Replication Management
Adds information to enable detection and suppression of duplicate operations
Reliable Multicast System
Platform
(a)
Replica of CORBA Object
CORBA ORB
Interceptor's Interface
Interceptor
Delivery of IIOP Messages and Dispatch of Threads
Profiling
Monitors replicas and processor load for resource management
IIOP Messages
Protocol Adaptation
Converts reliable multicast messages into IIOP messages
Scheduling
Schedules and dispatches operations onto threads
Reliable Multicast Messages
Replication Management
Detects and suppresses duplicate operations
Reliable Multicast System
Platform
(b)

Figure 2.4: Enhancements provided by interceptors for fault-tolerant CORBA in the path of (a) outgoing messages, and (b) incoming messages.

thread scheduling, protocol adaptation and consistency management. The Eternal system [42, 45, 47, 48] exploits interception in this manner to enhance CORBA with fault tolerance. Eternal's Interceptor allows fault tolerance to be provided transparently to the ORB and to the application, by mapping the intercepted system calls or library routines onto the Eternal Replication Mechanisms and the Eternal Logging-Recovery Mechanisms.

In the current version of the Eternal system, the Interceptor is implemented as a collection of library interposers, each interposer overriding a specific subset of a standard library and, thus, providing specific enhancements. To divert the CORBA application's TCP/IP-based IIOP communication to the Replication Mechanisms, the Interceptor employs a *socket library interposer*. To manage the activation and dispatch of operations to the different threads in a multithreaded object for reasons of replica consistency, the Interceptor employs a *thread library interposer*.

2.2.1 Socket Library Interposer

To facilitate the consistent replication of objects, the IIOP messages are conveyed over a reliable totally ordered multicast protocol, instead of over TCP/IP. For this purpose, Eternal employs the Totem system [40], not only for its high performance and its fault tolerance, but also for its simple and elegant interface.

The Interceptor, collocated with the application objects in the process's address space, completely hides the alteration of the course of the IIOP messages from both the ORB and the CORBA application objects. Thus, all communication between CORBA objects occurs, without their knowledge, over the reliable totally ordered multicast protocol, instead of over TCP/IP.

2.2.1.1 Default Behavior

Figure 2.5 shows the typical sequence [65] of library routines that a CORBA client and server must invoke in order to establish a TCP/IP connection, and to communicate using IIOP, using a connection-oriented protocol such as TCP/IP.

Default `socket()` Routine

The `socket()` routine is used by the CORBA server to create its listening socket (where it receives client requests that lead to further connections being spawned). The CORBA client uses the `socket()` routine to create a TCP/IP socket that eventually connects to the CORBA server.

Default `bind()` Routine

The `bind()` routine is invoked by a CORBA server to bind itself to an address on which it can subsequently listen for connection requests from clients.

Default `listen()` Routine

The `listen()` routine is invoked by a CORBA server to indicate its willingness to listen for connection requests from clients.

Default `accept()` Routine

The `accept()` routine is invoked by a CORBA server to wait for a client to send a connection request. The `accept()` routine takes the first enqueued connection request, and establishes a new TCP/IP socket with the client that initiated the connection request. The CORBA server uses this new socket descriptor to exchange IIOP messages with the CORBA client at the other end of the connection.

Default `connect()` Routine

The `connect()` routine is invoked by a CORBA client to establish a connection with a CORBA server.

Default `setsockopt()` Routine

The `setsockopt()` routine is invoked by the ORB, on behalf of the CORBA client or server, to set options that affect the performance or efficiency of the socket. Because the

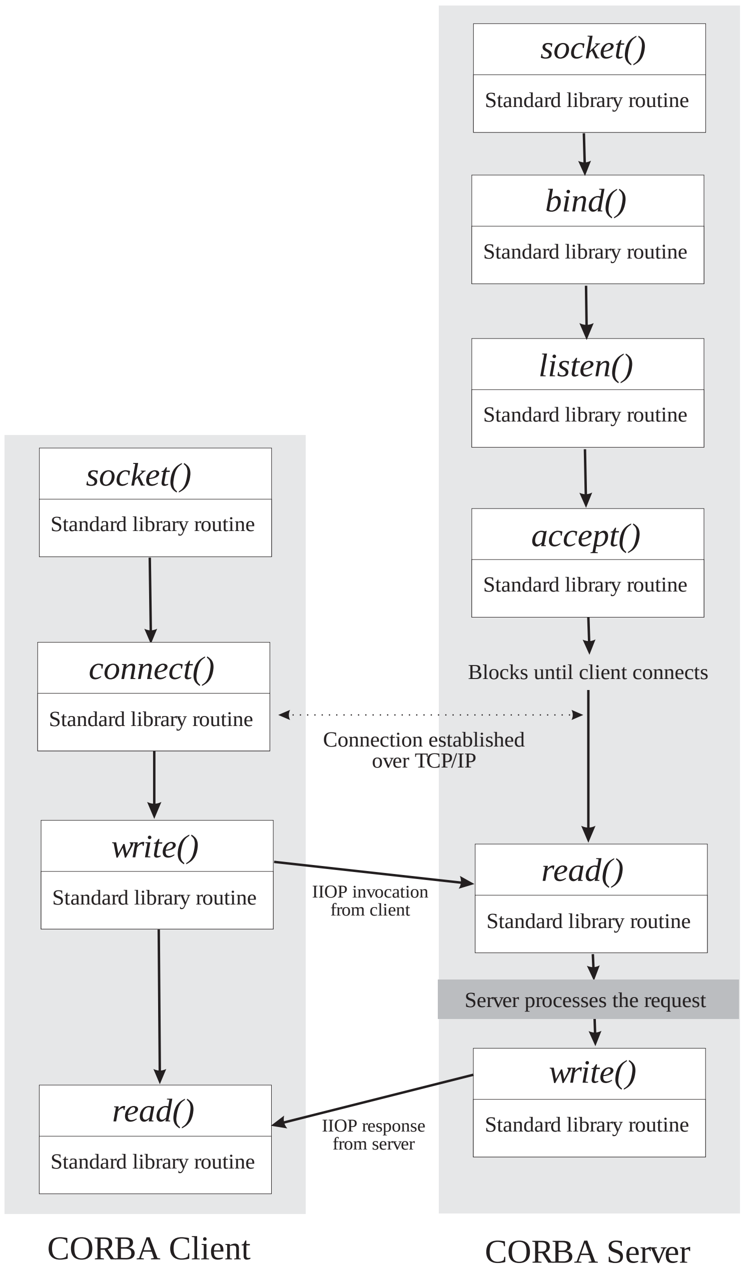


Figure 2.5: Sequence of steps for connection establishment and the communication of IIOP messages between an unreplicated CORBA client and an unreplicated CORBA server using the standard socket library routines.

CORBA client and server communicate over TCP/IP, some of these socket options may be particular to TCP/IP. For instance, the TCP_NODELAY socket option used by ORBs such as VisiBroker (from Inprise Corporation) [22] and TAO (from Washington University, St. Louis) to allow TCP/IP clients to send small messages as soon as possible, without any buffering delays.

2.2.1.2 Interposed Behavior

In order for the Eternal system to ensure that the inter-object communication occurs over a reliable totally ordered multicast group communication protocol (instead of over TCP/IP), both the client and the server must establish the socket to the Replication Mechanisms instead. To achieve this without modifying the application, and without either the client or the server ever being aware of it, a socket library interposer redirects all TCP/IP communication to the Replication Mechanisms. The socket library interposer "replaces" the socket library routines that CORBA objects use to connect and communicate over TCP/IP. It is not necessary for the socket library interposer to completely replace all of the symbol definitions within a Unix socket library such as `libsocket.so`.

The sequence of operations using the socket library interposer is shown in Figure 2.6. The TCP/IP socket between the client and the server is now converted to a Unix domain socket to the Replication Mechanisms. Because the socket library interposer preserves the semantics of the socket operation, including valid return values, the client (server) continues to believe that the peer endpoint of the socket is the server (client). This ensures that both the client and the server continue to behave in a normal manner, and communicate with each other using IIOP messages. The Replication Mechanisms, the real recipient of the IIOP messages, convey them over the underlying multicast group communication system.

The fact that the socket library interposer maintains the illusion of a TCP/IP connection to the CORBA client and server also implies that, once the socket is established, and valid socket descriptors are returned to the client and the server, the IIOP messages are automatically sent to the Replication Mechanisms using the socket descriptors. This implies that the `read()` and the `write()` routines do not need to be interposed. They will simply send and receive IIOP messaes through a valid socket descriptor, which the application and the ORB "believe" refers to a TCP/IP socket but which, in fact, refers to the Unix domain socket between the application and the Replication Mechanisms.

The socket library interposer does not need to interfere in the actual path of the IIOP messages once the socket has been established. This is a tremendous advantage in terms of performance because the Interceptor is absent in the application's invocation-response path, which is the critical path for determining an application's performance.

While library interpositioning overrides the original definitions of the library routines, it does allow for access to the original uninterposed definitions of the library routines, should they be required. For instance, when the TCP/IP connection is converted into a Unix domain connection to the Replication Mechanisms, while `socket()` is overridden to convert an AF_INET socket to an AF_UNIX socket, this nevertheless requires access to the uninterposed `socket()` routine to form the Unix domain socket.

The socket library interposer consists, in fact, of 140 lines of code written in C (because the real socket library is written in C) that overlays only specific socket library routines, including `socket()`, `bind()`, `listen()`, `accept()`, `connect()` and `setsockopt()`. Once the socket is established, `close()` on the socket file descriptor (which now represents a Unix domain socket instead of a TCP/IP-based socket) uses the standard library routine.

`socket()` Interposer

Both the client and the server are made to assume a client role with respect to the local Replication Mechanisms on their machines. The reason for this is that the Replication Mechanisms serve as the conduit for IIOP messages which are, in fact, conveyed through the underlying multicast group communication protocol.

Thus, the `socket()` interposer invokes the real `socket()` routine to create an unconnected Unix socket instead of the TCP/IP socket that the CORBA object intended. The `socket()` interposer then invokes the real `connect()` routine to establish a connection between the object and the Replication Mechanisms. The valid socket descriptor corresponding to the Unix socket is then returned to the CORBA object. Thus, any operation that the CORBA object performs using this socket descriptor directly affects its connection with the Replication Mechanisms.

`bind()` Interposer

With socket library interpositioning, the Replication Mechanisms handle all connection establishment between CORBA client groups and server groups, thereby relieving a CORBA server of its listener role. Thus, the `bind()` interposer is not required to do anything. However, in order for the CORBA server not to receive errors from the invocation of the `bind()` routine, the `bind()` interposer returns 0, indicating that the `bind()` was successful.

`listen()` Interposer

Because this functionality is now provided by the Replication Mechanisms, the `listen()` interposer does nothing, but returns 0 to the CORBA server, indicating that the `listen()` did not result in errors.

`accept()` Interposer

With library interpositioning, the `accept()` interposer "listens" for client connection requests by executing a blocking read on the Unix socket that connects the CORBA server to the Replication Mechanisms.

On receiving a connection request from a client object group, the Replication Mechanisms write to this socket, thereby causing the read to unblock. The return from the read call causes the `accept()` interposer to establish a connected Unix socket with the Replication Mechanisms for the purpose of communicating with the client group. The CORBA server is completely unaware of the actions of the `accept()` interposer, and continues to believe that the `accept()` routine that it invoked is still blocked. The `accept` interposer returns the socket descriptor of the newly established Unix socket that represents a virtual connection with the CORBA client (through Eternal). The server uses this descriptor to exchange IIOP messages over the socket, whose peer endpoint it believes is the client.

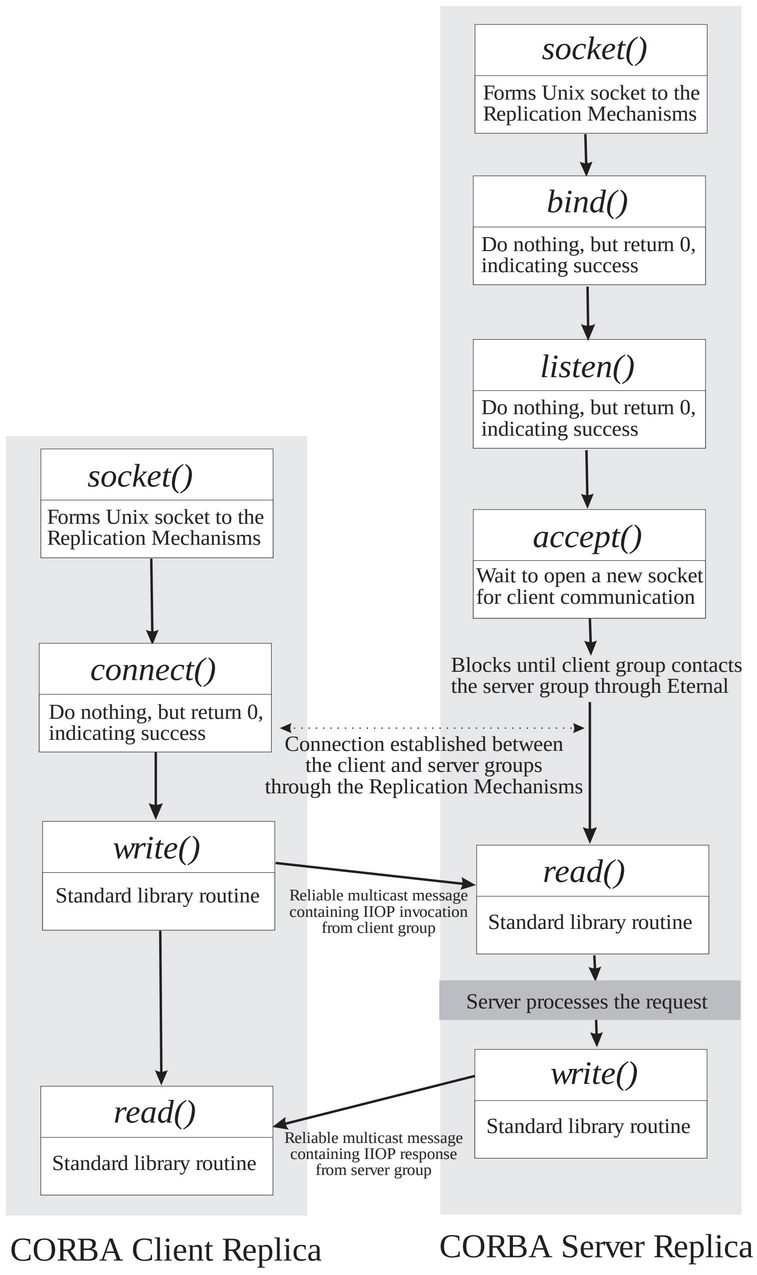

socket()
Forms Unix socket to the Replication Mechanisms
bind()
Do nothing, but return 0, indicating success
listen()
Do nothing, but return 0, indicating success
accept()
Wait to open a new socket for client communication
Blocks until client group contacts the server group through Eternal
socket()
Forms Unix socket to the Replication Mechanisms
connect()
Do nothing, but return 0, indicating success
Connection established between the client and server groups through the Replication Mechanisms
write()
Standard library routine
Reliable multicast message containing IIOP invocation from client group
read()
Standard library routine
Server processes the request
write()
Standard library routine
read()
Standard library routine
Reliable multicast message containing IIOP response from server group
CORBA Client Replica
CORBA Server Replica

Figure 2.6: Sequence of steps for connection establishment and the communication of IIOP messages between a CORBA client replica and a CORBA server replica using the Eternal Interceptor's socket library interposer, in conjunction with the Eternal Replication Mechanisms.

`connect()` Interposer

Because this functionality is now provided by the Replication Mechanisms, the `connect()` interposer does nothing, but returns 0 to the CORBA server, indicating that the `connect()` did not result in errors.

`setsockopt()` Interposer

Because the CORBA client and server intended to communicate over TCP/IP, some of these socket options are particular to TCP/IP, and not suitable for the Unix sockets created by the `socket()` interposer. These options need to be disabled through the `setsockopt()` interposer.

2.2.2 Thread Library Interposer

The specification of multithreading within the CORBA standard provides no guarantees about the order in which the ORB dispatches requests across the threads. The order of execution of threads within an object determines the state of the object. Because the ORB does not guarantee the deterministic dispatch of threads, the order of operations in two replicas of the same object might be different and, thus, their states might be inconsistent at the end of a sequence of thread executions.

To preserve replica consistency for multithreaded objects, or for processes containing multiple objects that share data, the interceptor uses its thread library interposer to introduce a scheduling component [48] to govern the order in which the threads and the operations are dispatched, over and above the total order in which the messages containing the operations are delivered to the ORB. The thread scheduler is described in detail in Chapter 6.

Thread library interposers "replace" the thread library routines that multithreaded ORBs and CORBA objects use to create and control threads. The thread library interposer does not need to replace all of the symbol definitions, for instance, within the Solaris thread library, `libthread.so`, or the POSIX thread library, `libpthread.so` [29] but to interpose only on the symbols of interest.

Of course, not all of the threads that the MT-domain or the ORB creates need to be controlled. For instance, an MT-domain that assumes the role of a server must necessarily "listen" for potential clients on a separate thread, and must spawn additional threads for every new client. The listening thread that an MT-domain server first spawns must not be prevented from running because, otherwise, the MT-domain would not be able to function in its role as a server. However, the additional threads that are dispatched to handle client invocations must be controlled because they might modify the state of the MT-domain.

The thread scheduler can interwork with any multithreading model adopted by the ORB. Moreover, the scheduler does not need to be in the path of outgoing messages from objects, but only in the path of incoming messages, before they are delivered to the application. Thus, the protocol adapter passes incoming IIOP messages (that it extracts from the group communication messages received from the replication management component) to the scheduler, which then determines the point in time at which the IIOP messages are delivered to the target objects.

Because the protocol adaptation, consistency management, and thread scheduling of the Eternal Replication Mechanisms operate deterministically, in concert with the underlying reliable totally ordered multicast group communication system, the deterministic behavior of the replicated objects is achieved, even in the presence of multithreading.

2.2.3 Other Library Interposers

2.2.3.1 Overcoming Sources of Nondeterminism

The use of replication for fault tolerance requires replica determinism, to ensure that no undesirable or unforeseen side-effects cause the states of the replicas of an object to become inconsistent. The Delta-4 project [57] employs a primary-backup, or passive, replication approach to overcome the problems associated with nondeterministic replicas. This solution can overcome the problem of nondeterminism only when the application is two-tiered, *i.e*, there is a client object invoking a server object. Using the primary-backup approach does not solve the problem of nondeterminism for arbitrarily tiered applications, *i.e.*, where a client invokes a server, which, in turn, acts as a client for a different server, and so on.

In the SCEPTRE 2 real-time system [6], nondeterministic behavior of the replicas also arises from preemptive scheduling. The developers of SCEPTRE 2 acknowledge the limitations of both active and passive replication of nondeterministic "capsules" for the purposes of ensuring replica consistency.

In the interests of strong replica consistency, it is necessary to "sanitize" nondeterministic library routines or system calls used by the application. The Transparent Fault Tolerance (TFT) system [8] enforces deterministic computation on replicas at the level of the operating system interface. TFT sanitizes nondeterministic system calls by interposing a software layer between the application and the operating system.

The thread library interposer within the Interceptor of the Eternal system handles one specific source of nondeterminism, namely, multithreading. This is addressed in detail in Chapter 6. Other sources of nondeterminism include library routines or system calls that return processor-specific information to the application. One such example is `gettimeofday()`; we describe below the sanitization of this library routine.

`gettimeofday()` Interposer

When this C library routine is invoked by a replicated object, different replicas may obtain different results, depending on their processor's value for the time of day. Unfortunately, the replicated object may use this information to update its internal state, and also to invoke other replicated objects in the system.

Clearly, inconsistency of the replicas ensues if the states of the different replicas are affected by different values for the time of day. A `gettimeofday()` interposer can use a central, deterministic time-issuing authority to supply the value of the time of day to be returned to the replicas that requested it. Of course, to prevent this time-issuing authority from being a single point of failure, its state is checkpointed to stable storage each time it is invoked, or the time-issuing authority is itself actively replicated.

Chapter 3

Replication Management

Fault tolerance in an object-oriented framework is provided by replicating objects. The purpose of replication is to provide redundant, identical copies of an object so that the object can continue to provide useful services, even though some of its replicas have failed, or the processors hosting some of its replicas have failed. It is therefore crucial that all of the replicas of the object have consistent state and deterministic behavior.

Eternal ensures strongly consistent replication both under fault-free operation and during recovery. The Eternal Replication Mechanisms described in this chapter exploit the Totem reliable totally ordered multicast group communication system to provide the necessary support for replication under normal operation, *i.e.*, when no faults occur in the system. In the event of a fault, the Replication Mechanisms coordinate with the Logging-Recovery Mechanisms described in Chapter 4 to maintain strong replica consistency.

3.1 Strong Replica Consistency

For ensuring strong replica consistency of the application, the Replication Mechanisms require that the application objects are *deterministic* in their behavior so that if two replicas of an object start from the same initial state, and have the same sequence of messages applied to them, in the same order, the two replicas will reach the same final state.

The mechanisms required for consistent replication vary with the replication style. For an actively replicated server (client) object, each replica responds to (invokes) every operation. It is relatively easy to mask the failure of a single active replica owing to the presence of the other active replicas which are also performing the operation. For a passively replicated server (client) object, only one of the replicas, designated the primary replica, responds to (invokes) every operation. In the event that the primary replica fails, one of the non-primary replicas is elected to be the new primary replica, which then resumes the operation of the failed primary.

Eternal ensures *strong replica consistency* for all replication styles by providing Replication Mechanisms and Logging-Recovery Mechanisms that address:

- **Ordering of operations.** All of the replicas of each replicated object must perform the same sequence of operations in the same order to achieve replica consistency. As described in Section 3.2, the Replication Mechanisms achieve this by receiving the IIOP messages of all of the objects on its processor, and then exploiting the underlying reliable totally ordered multicast group communication system for conveying the IIOP invocations (responses) to the replicas of a CORBA server (client). Eternal's use of reliable totally ordered multicast communication of the application's messages facilitates replica consistency under both fault-free and recovery conditions.

- **Duplicate operations.** Owing to the nature of replication, the potential for duplicate messages exists, *e.g.*, when every replica of a three-way actively replicated client object invokes an operation on a replicated server object, every server replica will receive three copies, or duplicates, of each invocation, one from each of the client replicas. Clearly, such duplicate invocations must never be delivered to, or performed on, server objects, and similarly, duplicate responses must never be returned to the client objects. The mechanisms used to perform duplicate detection and suppression are described in Section 3.4 of this chapter.

- **Recovery.** When a new replica is activated, or a failed replica is recovered, *before* it issues an invocation, performs an operation, or issues a response, the new or recovered replica must have the consistent state that the other replicas of the same object already possess. The Replication Mechanisms transfer the state of one of the replicas to any new replica in the case of active replication. In the case of passive replication, the Replication Mechanisms transfer the state of the primary replica to the non-primary replicas in accordance with the type (warm, cold) of passive replication that is employed. The mechanisms for state transfer are described in Chapter 4.

- **Multithreading.** Unfortunately, many commercial ORBs and CORBA applications employ multithreading, a significant source of non-determinism. For multithreaded ORBs which allow for an object to execute multiple different operations simultaneously, the Replication Mechanisms exploit the Interceptor to provide additional mechanisms, described in Chapter 6, to ensure replica consistency, regardless of the ORB's or the application's multithreading.

3.2 Reliable Totally Ordered Multicast

Replica consistency implies that all of the replicas of an object that execute an operation must have the same, or consistent, state at the end of that operation. One way of ensuring this, given that the replicated object is deterministic, is for all of the operational replicas of an object to "see" the *same* sequence of operations in the *same* order, thereby resulting in the same state at the end of each operation. This can be achieved by using reliable totally ordered multicast messages to convey the invocations to the replicas of an object.

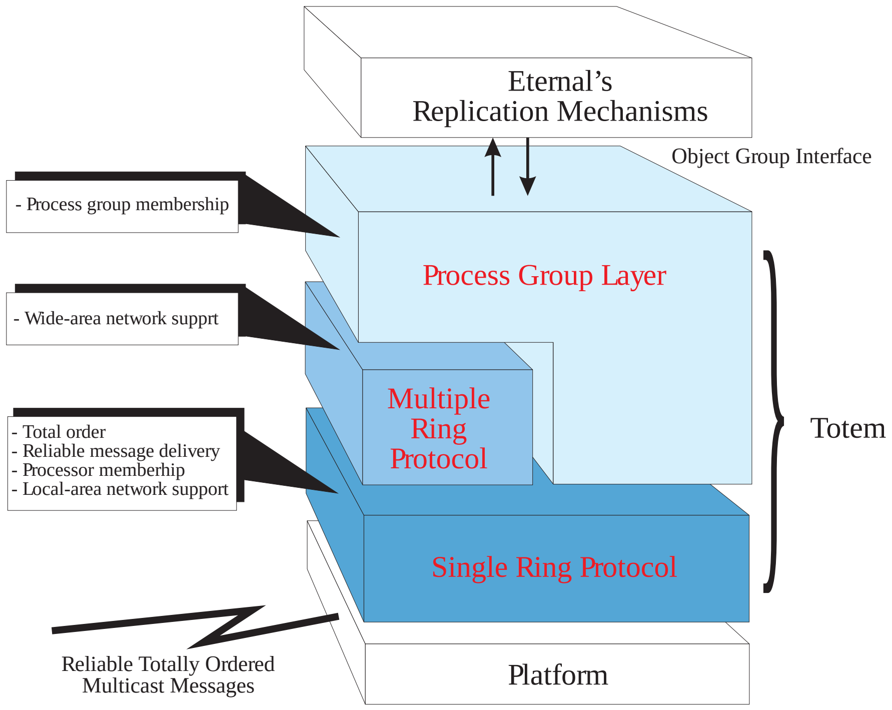


Figure 3.1: The Totem group communication system.

Unfortunately, the current CORBA standard provides no support for reliable totally ordered multicast protocols. Considerable modification to the transport layers of the ORB would be required before an existing ORB could use such a protocol. Eternal's Interceptor enables the Replication Mechanisms to exploit the services of a reliable multicast protocol without requiring the modification of either the ORB or the application. The Interceptor captures the IIOP messages from the application and passes them to the Replication Mechanisms, which then use the Totem reliable totally ordered group communication system to convey the messages across the network.

3.2.1 The Totem System

Typical applications consist of processes that cooperate or share information to perform a task. Such a collection of processes is referred to as a *process group*, and can be considered abstractly as a single unit. The membership of a process group is simply the list of all of its constituents; the members of a process group can be distributed across multiple processors in the network.

The Totem system [1, 40] shown in Figure 3.1 is a suite of group communication protocols that provide reliable totally ordered multicasting of messages to processors operating in a single local area-network, or in multiple local-area networks interconnected by gateways.

Each message has a unique timestamp assigned to it by the originator of the message. These timestamps are used to deliver messages in a single system-wide total order that respects Lamport's causal order [33]. The Totem system also provides membership and topology change services to handle the addition of new and recovered processors and processes, the deletion of faulty processors and processes, and the partitioning and remerging of the system.

The virtual synchrony model of Isis [7] orders group membership changes along with the regular messages. It ensures that failures do not result in incomplete delivery of multicast messages or holes in the causal delivery order. It also ensures that, if two processors proceed together from one view of the group membership to the next, then they deliver the same messages in the first view. The extended virtual synchrony model [41] of Totem extends the model of virtual synchrony to systems in which the network can partition and remerge, and in which processors can fail and recover. Processors in different components of a partitioned network may deliver the same messages, and yet the order in which the messages are delivered within each component of the partitioned network is consistent.

Totem provides a process group interface [35] that allows applications to be structured into process groups, with Totem maintaining information about the current memberships of all of the process groups that it supports across the system. The process group interface exploits the services and guarantees of the underlying Totem protocols to provide similar guarantees within and across process groups. Thus, Totem provides a system-wide total order within and across all of the process groups within the system.

Through this interface, application processes can join a process group to perform tasks of the application. A processor can support many process groups, and a process can belong simultaneously to multiple process groups. The services of a process group can be invoked transparently, with no knowledge of its exact membership or of the location of its member processes. A process in the system can thus address all of the members of a process group (including its own) as a whole, using Totem. A process can send messages to one or more process groups, regardless of whether the process belongs to these groups. The multicast messages are totally ordered within and across all receiving process groups.

The Eternal Replication Mechanisms employ the reliable totally ordered multicasts of Totem to convey the IIOP messages exchanged between replicated objects, thereby facilitating replica consistency.

3.2.2 Object Groups

Analogous to the notion of a process group, an *object group* represents a collection of objects that cooperate to provide some useful service. The members of such a collection of objects could be identical or dissimilar, and could be hosted on the same processor or on different processors. If all of the members of the group are identical in interface and implementation, the object group is *homogeneous*.[1] This is a useful representation of a replicated object, where the replicas of the object correspond to the members of the homogeneous object group. In Eternal, an object group is equivalent to a replicated object in the system.

[1] On the other hand, a *heterogeneous object group*, whose member objects are dissimilar, can be used as an abstraction for load balancing and management applications. For the remainder of this text, however, the term "object group" refers to a homogeneous object group.

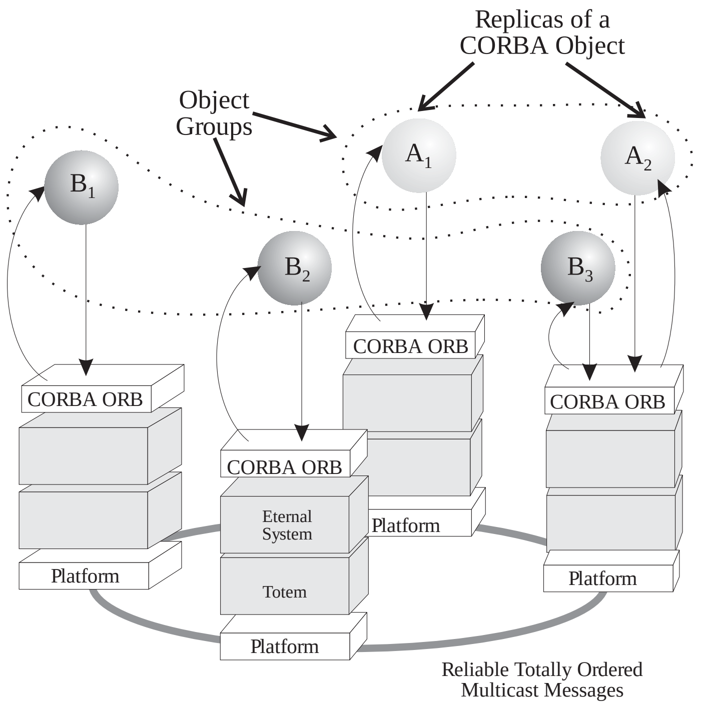

Replicas of a CORBA Object
Object Groups
A1
A2
B1
B2
B3
CORBA ORB
Eternal System
Totem
Platform
Reliable Totally Ordered Multicast Messages

Figure 3.2: Object groups in the Eternal system.

The utility of the object group abstraction lies in replication transparency and failure transparency. While all of the details of the coordination and interaction between the group members (replicas), the number of group members (degree of replication), the replication style (active, cold passive, warm passive) and their exact location (distribution of replicas) within the system are necessarily visible to the Replication Mechanisms, these details are hidden from all of the clients of the object. This transparency implies that, from the client's perspective, the invocation of a replicated object is no different from that of a single unreplicated object. Invocation of the group (replicated object) is transparently translated into the invocation of its members (replicas). The object group membership mechanisms enable the addition and the removal of replicas from an object group in a manner that is transparent to the application.

In Eternal, both client and server objects can be replicated and can, thus, can be represented as object groups. Eternal exploits the reliable totally ordered multicasts of Totem to communicate the invocations to, and the responses from, every object group. At the client replica, the Interceptor captures the client's IIOP invocation and passes it to the Replication Mechanisms. The Replication Mechanisms, in turn, encapsulate the IIOP invocation into a group communication message to be multicast by the underlying Totem system. This is done for every client replica that sends the invocation. At the server end, the receiving Replication Mechanisms extract the IIOP invocation from the incoming reliable multicast message. The Replication Mechanisms then deliver the IIOP invocation to the destination server replica through the Interceptor. This is done for every server replica that is to receive the invocation. A similar procedure is repeated for the IIOP response from the replicated server to the replicated client. Because the IIOP invocations and responses are conveyed through reliable totally ordered multicast messages, all of the server (client) replicas of an object receive the same invocations (responses) in the same order, thereby ensuring consistency of the states of the server (client) replicas.

3.3 Replication Styles

Eternal incorporates support for active replication, cold passive replication and warm passive replication styles. The Eternal system allows the user to dictate the choice of replication style for every application object (that is to be replicated) at system configuration time. Equipped with the knowledge of system resources (processors, memory, etc.), the user can also select the appropriate processors on which to locate the replicas, in the interests of reliability and performance.

However, regardless of the replication style used, each replicated client (server) object is unaware of its own replication, *i.e.*, each client (server) replica is unaware that there are other replicas of the same object. Eternal allows the application programmer to write the application without worrying about replication; from the application's perspective, communication with a replicated object looks no different from communication with a single unreplicated object. A replicated client (server) invokes (responds to) a replicated server (client) object just as if it were invoking (responding to) a single unreplicated server (client) object.

3.3.1 Passive Replication

For a passively replicated object, there exists a single designated replica, known as the *primary replica*, that performs all of the operations for the replicated object. All of the other non-primary (backup) replicas do not issue, or receive, invocations and responses. The backup replicas do not perform any operations while the primary replica is operational. Their sole purpose is to provide a pool of replicas from which a new primary replica can be chosen, should the current primary replica fail.

When a passively replicated client object invokes an operation on a server object, only the primary client replica issues the invocation. When a passively replicated server object receives an invocation, only the primary server replica performs the operation, and returns the response. To support passive replication, the Replication Mechanisms of Eternal multicast the invocation (response) from the primary client (server) replica to the server (client) object group via the underlying Totem reliable totally ordered multicast group communication system. The total ordering of messages ensures that the state of the passively replicated object after recovery is consistent with that of the passively replicated object before the fault occurred. While the Replication Mechanisms at the primary replica deliver all incoming invocations and responses to the primary replica, the Replication Mechanisms at every backup replica receive, and record the incoming messages, but do not deliver them

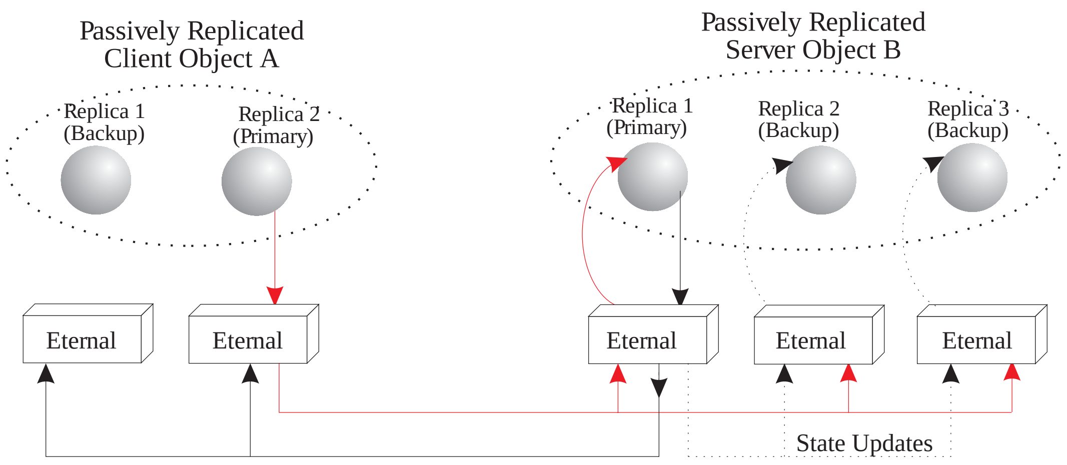


Figure 3.3: Warm passive replication, with state updates being transferred at the end of each operation.

to the backup replica. This recording of messages is essential for duplicate detection and recovery, in the event that the primary replica fails.

There exist different styles of passive replication that differ in the degree to which the states of the backup replicas "lag" behind the state of the primary replica. The Eternal system provides support for cold and warm passive replication. Eternal allows the user to specify properties particular to passive replication at system configuration time. For instance, on deciding to use passive replication for an object, the user can provide a preferred location (processor) for the primary replica, the frequency of checkpointing (for cold or warm passive replication) or the frequency of state transfer (for warm passive replication). The user can also dictate a recovery sequence, *i.e.*, the order in which backups are to be selected for the role of the new primary replica, should the existing primary replica fail.

3.3.1.1 Cold Passive Replication

In the case of a cold passively replicated server object, the backup replicas are not even loaded into memory, and thus do not come into existence until the primary replica fails. However, using the user's input at system configuration time, Eternal "knows" the locations (processors) where the cold passive replicas must be created, when required.

Since there is only one replica, the primary replica, that exists at any point in time, to provide fault tolerance, the primary replica's state must be captured for use in the event that it fails. At a frequency dictated by the user at system configuration time, Eternal's Logging-Recovery Mechanisms retrieve the state of the primary replica, and record this in a log. The log will contain not only the last checkpointed state, but also the IIOP invocations that have been delivered to the primary replica since the last checkpoint.

In the event that the primary replica fails, one of the cold backup replicas is loaded into memory, and assumes the role of the new primary replica. Clearly, for the new primary replica to "take over" from the old primary replica without violating the replica consistency, the new replica's state must be identical to the state of the old primary replica before the old primary failed.

Thus, before the new primary can fully assume the role of the primary replica, its state is initialized using the last checkpoint recorded previously by the Logging-Recovery Mechanisms for the old primary replica. After this checkpoint is applied to the new primary, the IIOP invocations that were logged after the previous checkpoint are also applied in the order that they were received and subsequently logged. Only when this recovery process is complete can the new primary start to issue or to receive any current invocations or responses, or to perform operations that might affect its state. The recovery mechanisms are described in detail in Section 4.1.2.

3.3.1.2 Warm Passive Replication

In warm passive replication, all of the backup replicas are created and initialized, and the state of the primary replica is retrieved and transferred to all of the backup replicas at a frequency that the user specifies at system configuration time. Hot passive replication is a variant of warm passive replication, with the state transfer occurring at the end of every operation on the primary replica. Thus, while the states of the backup and primary replicas may differ while the primary replica performs an operation, their states are consistent at the end of each state transfer.

Unlike cold passive replication, the warm passive backup replicas are loaded into memory and are running. However, just as with cold passive replication, the Replication Mechanisms do not deliver any incoming invocations or responses to the backup replicas. Instead, at a user-specified frequency, Eternal's Logging-Recovery Mechanisms retrieve the state of the primary replica, and transfer this state to the backup replicas. As long as the primary replica is running, the only messages that the Replication Mechanisms deliver to the backup replicas are the messages that contain the state of the primary replica. If the primary replica fails, a new primary replica is chosen from the backup replicas. The "lag" in the state of the new primary replica (formerly a backup replica) and the old primary replica depends on the frequency of state transfer. With warm passive replication, the states of the backups may be obsolete far more often than if hot passive replication were used because the primary replica's state is transferred to warm backup replicas at a lower frequency than to hot backup replicas. The state transfer mechanisms Logging-Recovery Mechanisms reconcile this "lag" through the mechanisms described in Section 4.1.2.

Figure 3.3 shows a two-way passively replicated client object A interacting with a three-way hot passively replicated server object B. Only the primary client replica issues the invocation to the server object. The primary server replica performs the operation corresponding to the invocation, and returns the response. Clearly, in the fault-free case of this example, there is no potential for duplicate invocations (responses) because only a single client (server) replica, the primary replica, issues the invocation (response).

Because hot passive replication is employed for server object B, when the primary server replica completes the operation (which may update the state of the primary), the Logging-Recovery Mechanisms at the primary server replica retrieve the primary replica's updated state, and transfer this state to the Logging-Recovery Mechanisms at the backup replicas. The receiving Logging-Recovery Mechanisms deliver the state to the backup replicas. Thus,

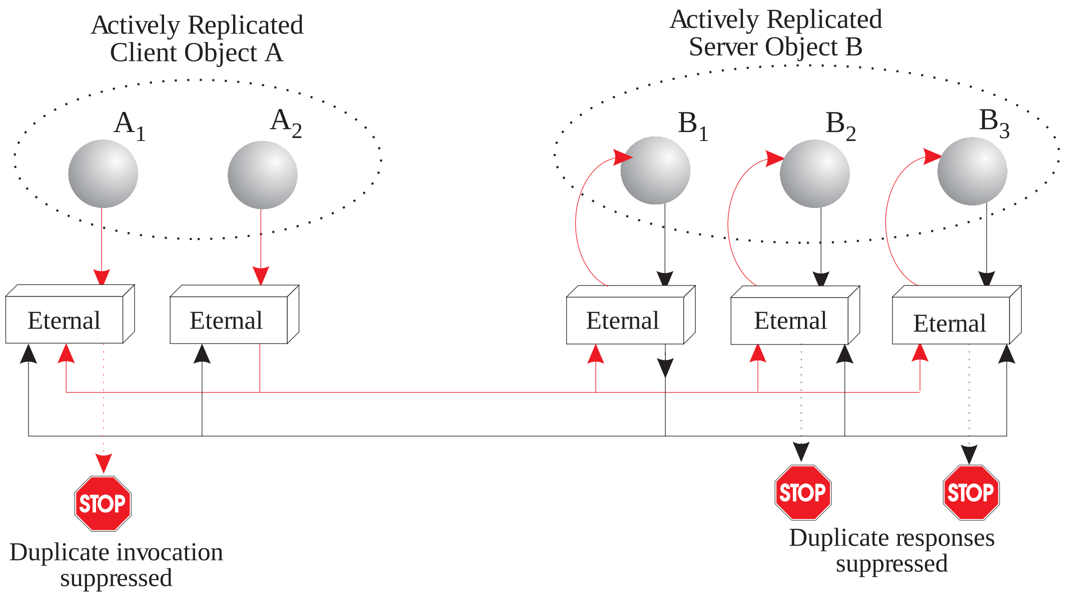


Figure 3.4: Active replication.

while the primary replica of object B performs the operation, the states of B's backup replicas may differ from that of the primary replica; however, the update operations at the end of the operation ensure that object B's replicas are consistent in state.

3.3.2 Active Replication

When an actively replicated client object invokes an operation on a server object, every client replica issues the invocation. Similarly, when an actively replicated server object receives an invocation, every server replica performs the operation, and every server replica returns the response.

To enable active replication, Eternal's Replication Mechanisms multicast the invocation (response) from every client (server) replica to the server (client) object group via the Totem underlying reliable totally ordered multicast system. The underlying Totem system ensures that all of the active server replicas of an object receive the same invocations in the same order. Thus, the server replicas will perform the operations in the same order, and all of the active client replicas will receive the responses in the same order. This ordering of operations ensures that the states of both the client and the server replicas are consistent at the end of the operation.

Figure 3.4 shows a two-way actively replicated client object A interacting with a three-way actively replicated server object B. The two active client replicas issue the invocation to the replicated server object. Each of the three server replicas performs the operation corresponding to the invocation, and each of them returns the response. Because both client replicas issue the same invocation, every server replica will receive two duplicate invocations and, thus, will perform the operation twice, instead of only once (as the client had intended). Similarly, there exists the danger that each client replica receives three duplicate responses to its single invocation, resulting in its state being corrupted if all three responses are delivered. In the interests of replica consistency, such duplicate invocations and duplicate responses must be detected and suppressed; only one copy of every distinct invocation or response must be delivered to the application.

The Logging-Recovery Mechanisms provides mechanisms, described in Section 3.4, that detect and suppress duplicate invocations and duplicate responses, thereby preventing inconsistencies that might otherwise arise. Because duplicate message detection at the destination is wasteful of network bandwidth, Eternal provides for duplicate detection at the source as well.

Figure 3.4 shows source-side duplicate detection and suppression through the use of "outgoing message loopback." At every client replica, the Replication Mechanisms multicast the invocation not only to the server replicas, but also to all of the client replicas. At the Replication Mechanisms hosting every server replica, these received invocations are intended for delivery to the server replicas. At the Replication Mechanisms hosting every client replica, these invocations aid in duplicate detection and suppression. For instance, in the figure, replica A_2's invocation happens to be multicast by A_2's Replication Mechanisms before A_1's Replication Mechanisms can multicast A_1's invocation. A_2's invocation is multicast to the server object B as well as to the Replication Mechanisms hosting the other replicas (in this case, only A_1) of client object A. On "seeing" this invocation from a fellow replica (A_2) of the same client object (A), the Replication Mechanisms hosting A_1 can refrain from multicasting the invocation from its own replica (A_1), which is a duplicate of A_2's invocation. Similarly, the duplicate responses from the server replicas B_1, B_2 and B_3 can be suppressed by the Replication Mechanisms hosting the server replicas.

In an asynchronous distributed system, source-side duplicate suppression can never be made completely effective. Thus, Eternal ensures that the duplicate messages that escape detection at the source are detected and suppressed at the destination. Of course, duplicate suppression at either the source or the destination hinges on the Replication Mechanisms's ability to detect that any two given invocations (*e.g.*, A_1's and A_2's invocations in Figure 3.4) or any two given responses (*e.g.*, B_1's and B_3's responses in Figure 3.4) are, in fact, duplicates of each other. The mechanisms that Eternal uses to detect duplicate messages are described in Section 3.4.

3.3.3 Comparison of Replication Styles

In the case of cold passive replication, under normal operation, the cost of checkpointing the primary replica's state to a log must be considered. If the state of the object is large, this checkpointing could become quite expensive. In the case of warm passive replication, if the state of the primary is large, transferring this state to the backup replicas, even if it is done periodically, could become quite expensive. The state transfer cost is incurred for active replication only during recovery, and never during normal operation.

On the other hand, cold passive replication requires only one replica to be operational and, thus, conserves processing power. While warm passive replication requires more replicas to be operational, these backups do not perform any operations (other than receiving the primary replica's state periodically), and also conserve processing power. With active replication, every replica performs every operation, and therefore consumes an equal amount of computational resources of the processor that hosts it. Thus, passive replication has the advantage that it is less consuming of processing power, *i.e.*, it does not require the operation to be performed by each of the replicas. If the operation is computationally expensive, the cost of passive replication can be lower (in the fault-free case) than that of active replication.

For every operation invoked on an actively replicated object, a multicast message is required to issue the operation to each target replica. This can lead to increased usage of network bandwidth because each operation may itself generate further multicast messages (as is the case with nested operations). For passive replication, because only one replica, the primary client (server) replica, invokes (responds to) every operation, passive replication may require fewer multicast messages. However, if the state of the primary replica is large, the state transfer or checkpointing may require many multicast messages.

With active replication, recovery time is faster in the event that a replica fails. In fact, because all of the replicas of an actively replicated object perform every operation, even if one of the replicas fails, the other operational replicas can continue processing and perform the operation. This is also true of warm passive replication if a backup replica fails.

However, if the primary replica fails, recovery time may be significant. For cold passive replication, recovery requires the reelection of a new primary, the transfer of the last checkpoint, and the application of all of the invocations that the old primary received since its last checkpoint. If the state of the object is large, retrieving the checkpoint from the log may be time-consuming. For warm passive replication, recovery may be faster because the warm backup replicas already have their states initialized to the last checkpoint of the primary owing to the periodic state transfers.

The cost of using active replication is dictated by application-specific issues, such as the number of replicas and the depth of nesting of operations. Active replication is favored if the cost of multicast messages and the cost of replicated processing is less than the cost of transmitting the object's state to every replica at the end of the operation. Hybrid active-passive replication schemes [21] have been considered, with the aim of addresing the reduction of multicast overhead in active replication styles, as well as of achieving the best of the active and passive replication styles.

3.3.4 Interactions between Replication Styles

While the Eternal system allows the user to choose between active and passive replication styles for an application object, it ensures that this choice of replication style is transparent to all of the other objects, as well as all of the replicas of the object itself. The replication transparency that Eternal provides implies that actively replicated objects and passively replicated objects are invoked by clients in exactly the same manner, although the Replication Mechanisms handle these invocations differently in each case, in a manner appropriate to the replication style.

The most interesting of the interactions between replicated objects with different replication style are shown in Figure 3.5. Here, the client object A is actively replicated and the server object B is hot passively replicated. The figure shows the sequence of steps in the

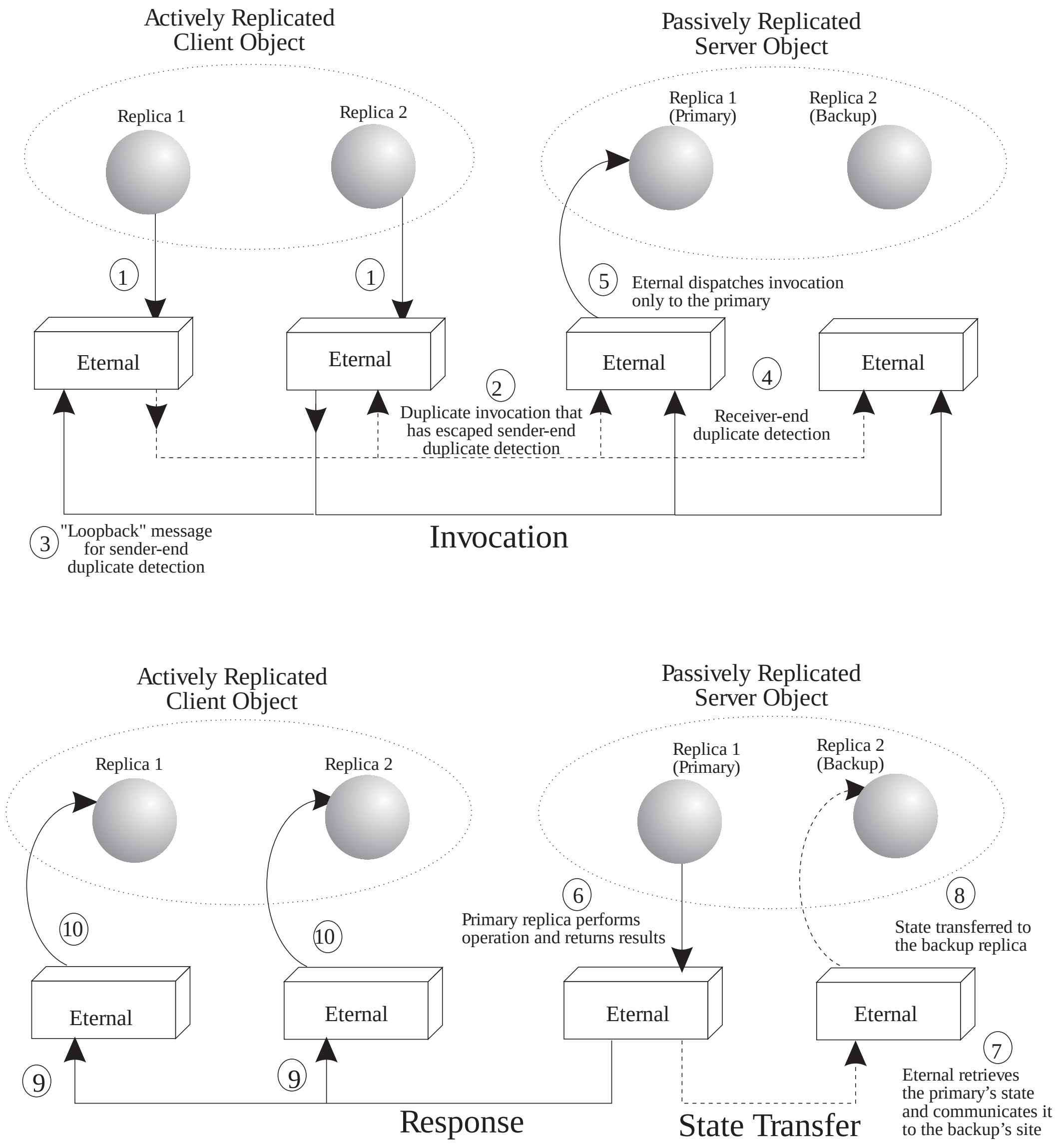

Actively Replicated Client Object
Replica 1
Replica 2
Passively Replicated Server Object
Replica 1 (Primary)
Replica 2 (Backup)
1
1
5 Eternal dispatches invocation only to the primary
Eternal
Eternal
Eternal
Eternal
2 Duplicate invocation that has escaped sender-end duplicate detection
4 Receiver-end duplicate detection
3 "Loopback" message for sender-end duplicate detection
Invocation
Actively Replicated Client Object
Replica 1
Replica 2
Passively Replicated Server Object
Replica 1 (Primary)
Replica 2 (Backup)
10
10
6 Primary replica performs operation and returns results
8 State transferred to the backup replica
Eternal
Eternal
Eternal
Eternal
9
9
7 Eternal retrieves the primary's state and communicates it to the backup's site
Response
State Transfer

Figure 3.5: Sequence of steps in the interaction between an actively replication client object with a passively replicated server object.

interaction between the two replicated objects. Eternal's Replication Mechanisms manage the invocations in such a way that the client (server) replicas of object A (B) are never aware of the passive (active) of object B (A), or even the fact that object B is replicated. Also, by providing fault transparency, Eternal ensures that the client (server) replicas of object A (B) are never aware of the failure of a server (client) replica.

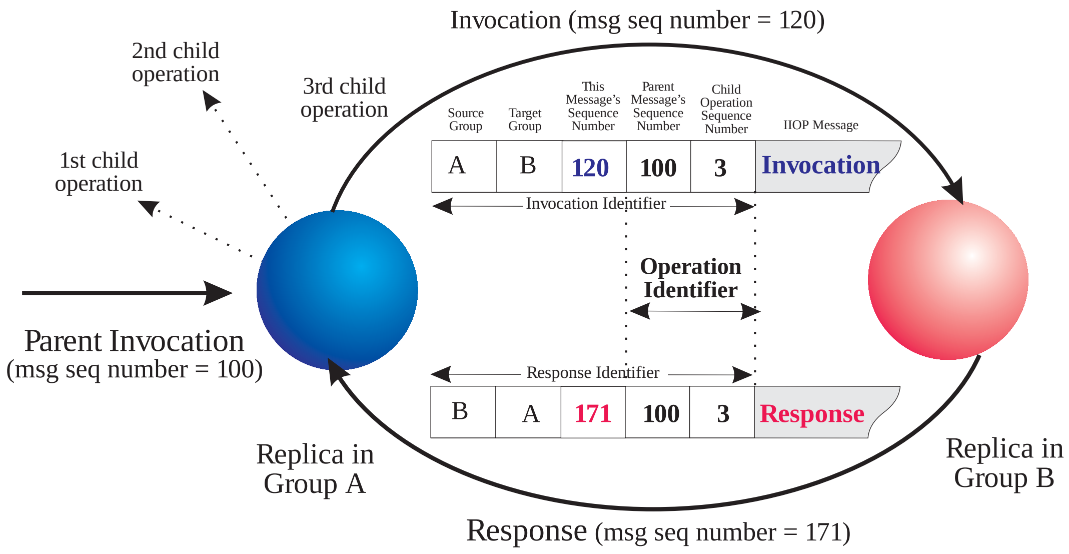


Figure 3.6: Assignment of invocation, response and operation identifiers.

3.4 Duplicate Detection and Suppression

In addition to the information that CORBA packages with an invocation, Eternal supplies unique *operation identifiers* that simplify the detection and suppression of duplicate invocations and duplicate responses.

The Replication Mechanisms attach an Eternal-specific header to every IIOP message that they multicast. The IIOP message, along with the Eternal-specific header, is encapsulated into a multicast message, formatted appropriately for Totem. Neither the multicast header nor the Eternal-specific header is intended to be delivered, or "seen", by the application objects. Their primary intent is to convey information essential for the proper operation of the infrastructure consisting of the Replication Mechanisms, the Logging-Recovery Mechanisms and the reliable multicast protocol.

The Eternal-specific header contains various pieces of information that are useful to the Replication Mechanisms in routing received messages. An important part of the Eternal-specific header is the information that the Replication Mechanisms use for detecting duplicate invocations and duplicate responses.

3.4.1 Operation Identifiers

To enable incoming response messages to be matched with their corresponding invocations, the Logging-Recovery Manager inserts an invocation (response) identifier into the Eternal-specific header for each outgoing IIOP invocation (response) message. For an outgoing invocation at the client end, the invocation identifier is derived as shown in Figure 3.6. For an outgoing response at the server, the Logging-Recovery Manager "remembers" and reuses a portion of the invocation identifier for the invocation that resulted in this response. The portion of the invocation identifier that is reused in its counterpart response identifier is known as the *operation identifier.*

The operation identifier uniquely represents the entire operation that consists of the invocation-response pair. Its uniqueness is derived from the use of the totally ordered message sequence numbers of the underlying Totem protocol.

Consider the invocation A_{inv} on a replicated object A, as a result of which object A dispatches multiple nested invocations. One of these nested invocations, say, the $S_{A_{inv}}$th nested invocation, denoted by B_{inv}, is issued by object A on another replicated object B. Assume that B performs the operation corresponding to this invocation, and returns a response B_{res} to object A. The invocation B_{inv} and its counterpart response B_{res} together constitute a single operation.

The invocation identifier that forms part of the Eternal-specific header for the invocation sent from A to B has the form

$$(T_{B_{inv}}, (T_{A_{inv}}, S_{A_{inv}})),$$

and the response identifier that forms part of the Eternal-specific header for the response sent from B to A has the form

$$(T_{B_{res}}, (T_{A_{inv}}, S_{A_{inv}})),$$

where

$T_{A_{inv}}$ = the totally ordered sequence number of the message containing A_{inv}

$T_{B_{inv}}$ = the totally ordered sequence number of the message containing B_{inv}

$T_{B_{res}}$ = the totally ordered sequence number of the message containing B_{res}

$S_{A_{inv}}$ = the sequence number of B_{inv} in the sequence of nested invocations by A.

The invocation and response identifiers are common in the last two fields $(T_{A_{inv}}, S_{A_{inv}})$; these fields together constitute the operation identifier. The first part of the invocation identifier $(T_{B_{inv}})$ may differ for the same invocation from the different replicas of A. Similarly, the first part of the response identifier $(T_{B_{res}})$ may differ for the same response from the different replicas of B. However, the operation identifier field of the invocation and response identifiers is identically generated by the Replication Mechanisms at every replica of A and of B.

The fields $T_{A_{inv}}$, $T_{B_{inv}}$ and $T_{B_{res}}$ are derived from the totally ordered message sequence numbers assigned by the Totem multicast group communication system. The system-wide uniqueness of these timestamps (as a result of the total ordering) contributes to the uniqueness of the operation identifiers, and thus, to the detection of duplicate messages.

In the example of Figure 3.6, $T_{A_{inv}}$ corresponds to 100, $S_{A_{inv}}$ corresponds to 3, $T_{B_{inv}}$ corresponds to 120 and $T_{B_{res}}$ corresponds to 171. While only one replica of A and one replica of B are shown, the operation identifier (100, 3) is identically generated by the Replication Mechanisms at every replica of A and B. Thus, the invocation and response identifiers for the operation contain the same unique operation identifier.

Operation identifiers assist the Replication Mechanisms in associating the invocation and the corresponding response that constitute the operation, because both the invocation identifier and its counterpart response identifier have the same operation identifier. Eternal records, for each source group on its processor, the invocation identifiers of all outgoing invocations for which responses are expected. When a response arrives, Eternal delivers the response only if the operation identifier field of the received response identifier corresponds to the operation identifier field of the invocation identifier of an outstanding invocation. Also, by using a transitive sequence of these operation identifiers, a complete sequence of nested operations can be traced.

Operation identifiers are also useful in discarding duplicate invocations and duplicate responses so that only non-duplicate messages are delivered to the destination group. By the rules outlined above, the Replication Mechanisms hosting a client (server) replica assign to each *distinct* invocation (response), an operation identifier that is *unique* to the operation, but has the *identical* value at the Replication Mechanisms hosting *every* replica of the client (server). Thus, for every incoming message, the Replication Mechanisms can compare the operation identifier fields of that message with others that it has received. If the operation identifiers match, the receiving Replication Mechanisms can safely discard the duplicate message, because it has a record of having "seen" the message previously, probably from a different replica of the same sender object.

3.4.2 Example

Eternal provides support for *nested operations* as well. By a nested operation, we mean an operation that results in the invocation of yet another operation or, in the terminology of Eternal, the "parent" invocation of one replicated object giving rise to a "child" invocation of another replicated object. The operation identifiers for duplicate detection, as described in Section 3.4, serve the additional purpose of identifying, and associating parent operations with, child operations.

Figure 3.7 shows an example of the use of operation identifiers in the detection of duplicate invocations and duplicate responses under fault-free conditions. With a nested operation such as that shown in the figure, Eternal takes great care to ensure that duplicate invocations and responses are suppressed and that the states of the replicas of *all* of the objects involved in the nested operation are consistent after the entire operation, even in the presence of faults.

Here, the actively replicated object A has three replicas and the passively replicated object B has three replicas. Some client object (not shown in the figure) invokes a method with invocation identifier $(100,(75,5))$ in object group A. Each of the replicas in A executes the method corresponding to this parent invocation, resulting in the invocations of other methods of other objects. One such child invocation (in fact, the fourth such child invocation) is issued by A on B.

The timestamp of the parent invocation that resulted in the subsequent child invocations is 100. Because the method invocation on replicated object B is the fourth in the sequence of invocations triggered by the execution of the parent invocation, at each replica in A, the operation identifier for the invocation of B by A is $(100,4)$.

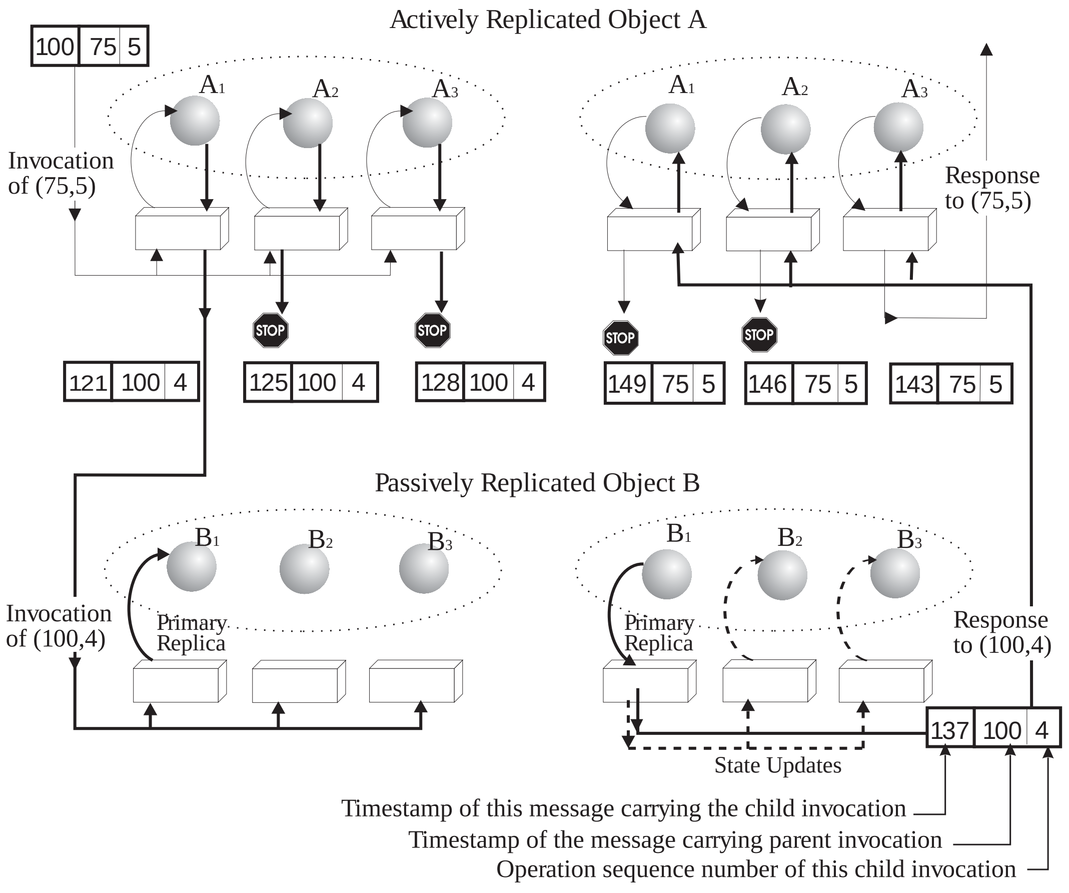

Actively Replicated Object A
100 75 5
A1 A2 A3
Invocation of (75,5)
STOP
121 100 4
125 100 4
128 100 4
149 75 5
146 75 5
143 75 5
Response to (75,5)
Passively Replicated Object B
B1 B2 B3
Invocation of (100,4)
Primary Replica
Response to (100,4)
137 100 4
State Updates
Timestamp of this message carrying the child invocation
Timestamp of the message carrying parent invocation
Operation sequence number of this child invocation

Figure 3.7: Use of operation identifiers in duplicate detection and suppression under fault-free conditions.

Because each of the three active replicas issues the invocation, the first of these in the total order is the one to be delivered to B, in this case, the message with invocation identifier $(121, (100, 4))$. The other two duplicate invocations are suppressed because the Replication Mechanisms for replica A_1 multicast a "loopback" message containing the invocation identifier $(121, (100, 4))$. In this case, the Replication Mechanisms for replica A_2 (A_3) receives the "loopback" message *before* multicasting A_2's (A_3's) invocation with the same operation identifier $(100, 4)$ as contained in the "loopback" message. Because these operation identifiers are identical, the Logging-Recovery Manager for A_2 (A_3) suppresses A_2's (A_3's) duplicate invocation.

It must be emphasized that the "loopback" messages are never intended for delivery to the application, but are "seen" only by Eternal for duplicate detection and suppression at the source. Of course, in an asynchronous distributed system, the "loopback" mechanism cannot be made completely effective, and some duplicate messages may escape detection at the source. However, Eternal's Mechanisms also detect and suppress such duplicate messages at the destination.

The primary replica in object group B then executes the method, after which the Logging-Recovery Mechanisms coordinate the transfer of the state of the primary replica to the backup replicas in object group B, and the Replication Mechanisms multicast the response to object group A using the response identifier $(137, (100, 4))$. Note that the invocation identifier $(121, (100, 4))$ and the corresponding response identifier $(137, (100, 4))$ refer to the same operation and thus have the same operation identifier $(100, 4)$.

At the end of the operation, the Replication Mechanisms at one of the replicas in object group A multicast the response (to the client that issued the original parent invocation) using the response identifier $(143, (75, 5))$. When the Replication Mechanisms at one of the other replicas in object group A "sees" this message, it suppresses its own replica's response for $(75, 5)$.

Of course, the most interesting problems in nested operations arise when a replica in the "chain of invocations" fails, and subsequently recovers. The mechanisms of Eternal that handle these problems are described in Chapter 4.

Chapter 4

Logging and Recovery Management

In addition to providing object replication which allows the application to continue to provide useful services even if the replicas fail, an important part of fault tolerance involves providing recovery to the failed replicas. Recovery [49] typically requires the activation of a new replica (to replace the failed one), and the synchronization of the state of the new replica with that of the other replicas of the object.

The Logging-Recovery Mechanisms that Eternal provides are responsible for recording incoming invocations, incoming responses and checkpoints of the replicas hosted on a processor. Typically, many replicas of different objects may be hosted on a processor and, thus, the Logging-Recovery Mechanisms maintain a single physical log per processor, where the log is indexed by the object group identifier.

4.1 Recovery for Different Replication Styles

4.1.1 Failure of an Active Replica

For an actively replicated object, fault recovery is relatively simple. Totem's reliable totally ordered multicast mechanisms ensure that an invocation or response has been dispatched either to all of the remaining replicas of an object or to none of them. Consequently, the operation will be performed by all of the remaining replicas or by none of them.

If an active replica fails while performing an operation, the remaining active replicas in the object group continue to perform the operation and return the result. The failure is thus transparent to the other replicated objects involved in the nested operation. Thus, active replication yields substantially more rapid recovery from faults.

When a failed active replica is recovered, the state of the new or recovering replica must be initialized with the state of an existing replica of the object. However, because

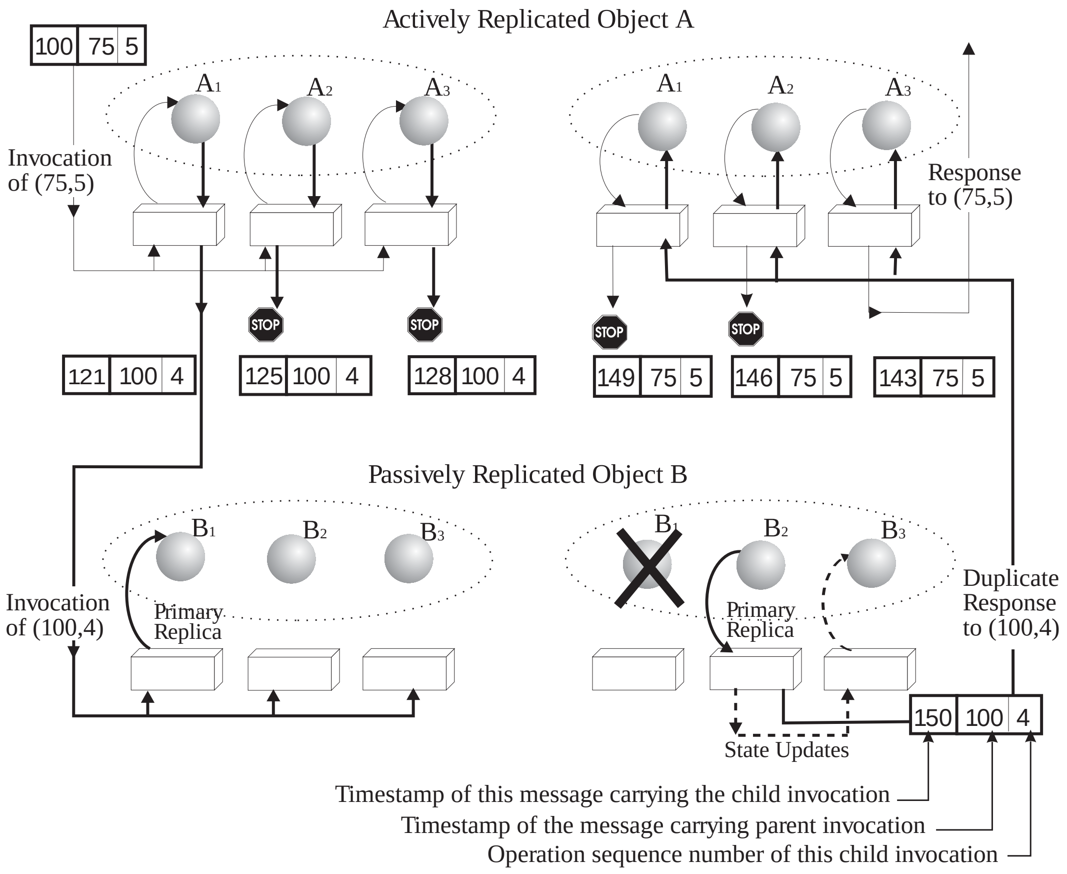

Actively Replicated Object A
100 75 5
A1
A2
A3
Invocation of (75,5)
Response to (75,5)
STOP
121 100 4
125 100 4
128 100 4
149 75 5
146 75 5
143 75 5
Passively Replicated Object B
B1
B2
B3
Invocation of (100,4)
Primary Replica
Duplicate Response to (100,4)
150 100 4
State Updates
Timestamp of this message carrying the child invocation
Timestamp of the message carrying parent invocation
Operation sequence number of this child invocation

Figure 4.1: Use of operation identifiers in duplicate detection and suppression under recovery conditions.

a recovering replica may continue to receive normal invocations and responses during the state transfer, such invocations and responses must be enqueued during the state transfer, and applied to the recovered replica after the state transfer is complete.

4.1.2 Failure of a Passive Replica

In the case of passive replication, the effect of the failure of a replica depends on whether the failed replica is a primary or a backup. If a backup replica fails, it is simply removed from the group by the object group membership mechanisms while the operation continues to be performed. Thus, the failure of a backup replica is transparent to the other object groups involved in the nested operation.

If the primary replica fails, one of the backup replicas is chosen to be the new primary replica and, thus, must be restored to the state that the old primary replica had just before it failed. Because the old primary replica is no longer available once it has failed, the primary's state (while it is operational) must be continuously or periodically captured and stored so that it is available for recovery if the primary replica fails. Just as with active replication, invocations and responses that arrive during recovery must be enqueued for delivery to the new primary replica only after its state has been initialized.

Consider the warm passively replicated object B in Figure 4.1. In keeping with warm passive replication, the state of the backup replicas is identical to that of the primary before A's invocation of B. The operation identifiers are assigned by the Mechanisms using the rules described in Section 3.4. The primary replica B_1 receives A's invocation $(121, (100, 4))$ with operation identifier $(100, 4)$, and performs the operation corresponding to the invocation. Because B is passively replicated, the Logging-Recovery Mechanisms at the backup replicas receive and store, but do not deliver, the invocation to their respective backup replicas.

Suppose that the primary replica B_1 fails after it returns a response to A's invocation, but before its state can be transferred to the backup replicas. Eternal elects a new primary replica from among the backup replicas. Through infrastructure state that it maintains for the replicated object B, the Logging-Recovery Manager realizes that, while object A's invocation was delivered to the old primary it never captured, the updated state of object B after the operation.

Thus, the Logging-Recovery Mechanisms at the new primary replica B_2 deliver the "incomplete" invocation $(121, (100, 4))$ (that the Logging-Recovery Manager had stored on receipt) with operation identifier $(100, 4)$ to B_2. Being deterministic, the new primary replica B_2 issues a response that has the same content and the same operation identifier (but a different message timestamp) as the one that the replicas of object A might have received from the old primary. The operation identifier $(100, 4)$ contained in the response to object A from both the old and the new primaries enables the Logging-Recovery Managers at the replicas of object A to detect and suppress these duplicate responses. Thus, while duplicate detection is not essential in passive replication under normal operation, it is required for ensuring strong replica consistency for passive replication under recovery.

4.2 Structure of the Log

The Logging-Recovery Mechanisms log messages, checkpoints and operation identifiers. The log also contains an enqueuing facility for the synchronization of state transfer.

4.2.1 Storing Checkpoints and Messages

The messages that are logged are those that arrive at the Logging-Recovery Mechanisms, through the totally ordered message sequence provided by the underlying Totem group communication system. The messages are verified to be non-duplicate messages before they are logged. Each entry in the log contains an incoming non-duplicate IIOP invocation or response along with its Eternal-specific header. The Eternal-specific header contains the operation identifer essential for duplicate detection. The operation identifiers are stored separately, and garbage collected independent of the logged messages, to enable faster duplicate detection.

For all replication styles, the log must store the operation identifiers to enable duplicate detection. However, for actively replicated objects, the log does not need to store any checkpoints or messages until recovery is initiated. At the point of recovery for an active replica, the mechanisms for synchronizing state transfer handle the logging of the checkpoints and the messages in such a way that replica consistency is guaranteed.

For a warm passively replicated object, the backup replicas already have their states initialized to the last checkpoint of the primary. However, in the interests of bringing up a new backup replica, this last checkpoint must be stored. Also, in the event that the primary replica fails, the messages following the last checkpoint must also be stored.

4.2.2 Storing Operation Identifiers

The duplicate detection scheme described in Section 3.4 requires the Logging-Recovery Mechanisms to store information about the operation identifiers of messages that it has "seen" previously. Fortunately, because the operation identifier is derived from the total order of messages, the fields of the operation identifier are monotonically increasing. If $(T_{A_{inv}}, S_{A_{inv}})$ is the operation identifier of a message from object group A to object group B, the next message from object group A to object group B will have an operation identifier $(T'_{A_{inv}}, S'_{A_{inv}})$, such that one of the following two conditions holds: either $\{T'_{A_{inv}} = T_{A_{inv}}, S'_{A_{inv}} > S_{A_{inv}}\}$ or $\{T'_{A_{inv}} > T_{A_{inv}}, S'_{A_{inv}} \geq 1\}$.

In either case, the Replication Mechanisms do not need to store the entire history of operation identifiers that it has ever "seen" for messages from object group A to object group B; only the last operation identifier that the Replication Mechanisms have received for a specific destination group, from a specific source group, needs to be stored. Messages from object group A to object group B that contain operation identifiers that occur earlier in the sequence than this last-seen operation identifier can be safely discarded as duplicate messages. Thus, for every source object group A that sends messages to a destination object group B, the Replication Mechanisms hosting every replica in B must record the operation identifier associated with the last-received non-duplicate message from A to B. Figure 4.2 shows the list of operation identifiers that the Logging-Recovery Mechanisms maintain.

4.3 Consistent State

Maintaining strong replica consistency involves ensuring that the replicated object has consistent state, even as its replicas perform operations that update their states, and even as replicas fail and recover. The deterministic behavior of an application object ensures that its replicas will have the same state after applying the same sequence of operations in the same order, starting from the same initial state. However, this is just one aspect of strong replica consistency – the recovery of a failed replica introduces additional complications.

Every replicated CORBA object can be regarded as having three kinds of state: *application-level state*, known to, and programmed into the application object by, the application programmer, *ORB-level state*, maintained by the ORB that hosts a replica of the object, and *infrastructure-level state*, invisible to the application programmer and maintained for the replicated object by Eternal's infrastructure.

Message	Direction	Logging at Replicas of A	Garbage Collection Point
Synchronous invocation	On object A	Operation identifier logged to detect duplicate invocations from the sender replicas	Receipt of next invocation (synchronous/asynchronous) from the same sender group
Asynchronous invocation	On object A	Operation identifier logged to detect duplicate invocations from the sender replicas	Receipt of next invocation (synchronous/asynchronous) from the same sender group
Synchronous invocation	From object A	Operation identifier logged to detect "loopback" duplicates and to match the response	Receipt of the response from the invoked group
Asynchronous invocation	From object A	Operation identifier logged to detect "loopback" duplicates	Next invocation from object A
Synchronous response	To object A	Operation identifier logged to detect duplicate responses from the sender replicas	Receipt of next response from the same sender group
Synchronous response	From object A	Operation identifier logged to detect "loopback" duplicates	Next response from object A

Figure 4.2: The logging and the garbage collection of operation identifiers by the Logging-Recovery Mechanisms hosting replicas of an object A for different types of communication (synchronous, asynchronous, invocation, response) involving object A.

For strong replica consistency, however, a fault-tolerant system must identify and maintain consistent ORB-level state and consistent infrastructure-level state across all of the replicas of a CORBA object, in addition to consistent application-level state.

Eternal's Logging-Recovery Mechanisms ensure that all of the replicas of an object are consistent in the three kinds of state. State transfer to a new or recovering replica includes the transfer of application-level state from an existing replica to the new replica, the transfer of ORB-level state from the ORB hosting an existing replica to the ORB that hosts the new replica, and the transfer of the infrastructure-level state from the Logging-Recovery Mechanisms managing an existing replica to the Logging-Recovery Mechanisms that manage the new or recovered replica.

4.3.1 Application-Level State

Application-level state is represented by the values of the data structures of the replicated object. Typically, the application-level state is visible to, and completely determined by, the application programmer. Of the three kinds of state, the application-level state is possibly the easiest to retrieve and to restore.

To enable application-specific state to be captured, the application programmer must ensure that every replicated CORBA object inherits the `Checkpointable` interface, shown in Figure 4.3. Because it is not possible to anticipate the structure, or the contents, of the application-level state of every conceivable application object, the application-level state is defined in a generic way to be of the type sequence<octet>.

This inherited IDL interface has two methods, (*get_state*()) and (*set_state*()), both of which are intended to be implemented by the application programmer. The *get_state*()

```
// Generic definition of application-level state
typedef sequence<octet> State;

// Exceptions associated with application-level state transfer
exception NoStateAvailable {};
exception InvalidState {};

// Interface to be inherited by every replicated object
interface Checkpointable
{
   // Retrieves application-level state
   State get_state() raises(NoStateAvailable);

   // Assigns application-level state
   void set_state(in State s) raises(InvalidState);
};
```

Figure 4.3: The `Checkpointable` IDL interface that must be inherited by every CORBA object in the application to enable the checkpointing and transfer of application-level state.

method, when invoked on a CORBA object, returns the current application-level state of the object. The *set_state*() method, when invoked on a CORBA object, with the application-level state (that has been retrieved by an earlier *get_state*() invocation) as a parameter, overwrites the current application-level state of the invoked object with the value of this parameter. Alternatively, an interface with methods for incremental state retrieval and assignment could be used; this is discussed briefly in Section 4.5.3.

In the case of an actively replicated object, the application-level state must be retrieved through a *get_state*() invocation on an existing active replica, and transferred to a new or recovering active replica through a *set_state*() invocation. In the case of passive replication, the application-level state of the primary replica must be retrieved using a *get_state*() invocation, and the state thus obtained must be either stored in a log (cold passive replication), or transferred to the backup replicas after every invocation (hot passive replication). The three phases of the recovery – the retrieval (*get_state*()), the transfer, and the assignment (*set_state*()) of application-level state are essential to replica consistency, and must be initiated through the totally ordered message sequence.

4.3.2 ORB-Level State

When an ORB hosts a CORBA object, the ORB provides location transparency to the object, and manages all connection-level and transport-level information on behalf of the object. This requires that the ORB maintain some information, at runtime, on behalf of the object. This information allows the ORB to control the dispatch of incoming invocations to the object, the creation of threads within the object, the creation of sockets to support client connections, and the dispatch of outgoing requests. Although specific to the CORBA object, this information is, however, completely hidden from the object, due to the nature of the ORB's mediating role in CORBA. Furthermore, this ORB-level information is not a part of the CORBA standard, and no standard interfaces exist for extracting this information from the ORB, or assigning values to it.

When a CORBA object is replicated, each replica of the object is hosted by a different copy of the same ORB on a different processor. For an actively replicated object, if the object and the ORB are both deterministic, and if every replica processes the same sequence of invocations in the same order, both the application-level state and the ORB-level will be consistent across all replicas, at the end of every operation.

However, in the case of recovery, consistent state is more difficult to achieve. Even if the application-level state of the new or recovering replica is initialized by a transfer of application-level state from an existing active replica, the two replicas (the existing replica and the recovering replica) will not be consistent in state because the respective ORB-level information will differ. Similarly, in the case of passive replication, if the primary replica fails and a backup replica takes over as the new primary replica, consistent replication cannot be ensured through the transfer of application-level state (from the old primary replica's logged checkpoints to the new primary replica) alone; the ORB-level state of the old primary must be checkpointed and transferred to the new primary replica's ORB to ensure strong replica consistency.

The ORB-level state consists of the values of the data structures (last-seen request identifier, threading policy, *etc.*) stored by the ORB, at runtime, on behalf of the object. For the strongly consistent replication of a CORBA object, the values of these different "pieces" of the ORB-level state must be identical at every replica of the object, as replicas perform operations, and as replicas fail and recover.

Unfortunately, these "pieces" of ORB state are hidden within the data structures of the ORB. This internal representation of the ORB-level state is not standardized (and indeed, such standardization would be contrary to the OMG's philosophy of standardizing interfaces, and not their implememtations), and thus, not sufficiently identical across different ORBs. Furthermore, the ORB-level state for a replicated object may be represented in a form that is specific to the ORB vendor or to the implementation of CORBA that is used. Thus, in the current state of the CORBA standard and the current commercial implementations of CORBA, it is infeasible to maintain strongly consistent replication of an object if its replicas are hosted by different ORBs. For all practical purposes and for the rest of the text, it is assumed that a strongly consistent replicated object has all of its replicas running over the same ORB.

4.3.2.1 Request Identifiers

One of the most basic "pieces" of ORB-level state is the request identifier that the ORB hosting an object inserts into every outgoing IIOP request message from the object. The ORB generates this request identifier on a per-connection basis for each client object that it hosts, and this request identifier is inserted into the corresponding IIOP reply message by the target server object when it responds to the invocation. On receiving the IIOP response, the client-side ORB first compares the request identifier embedded in the IIOP response message from the server with the value that it assigned for the the client's IIOP invocation of the server. Only on verifying that the returned request identifier matches the expected (transmitted) request identifier will the client-side ORB deliver the response to the client that issued the corresponding request.

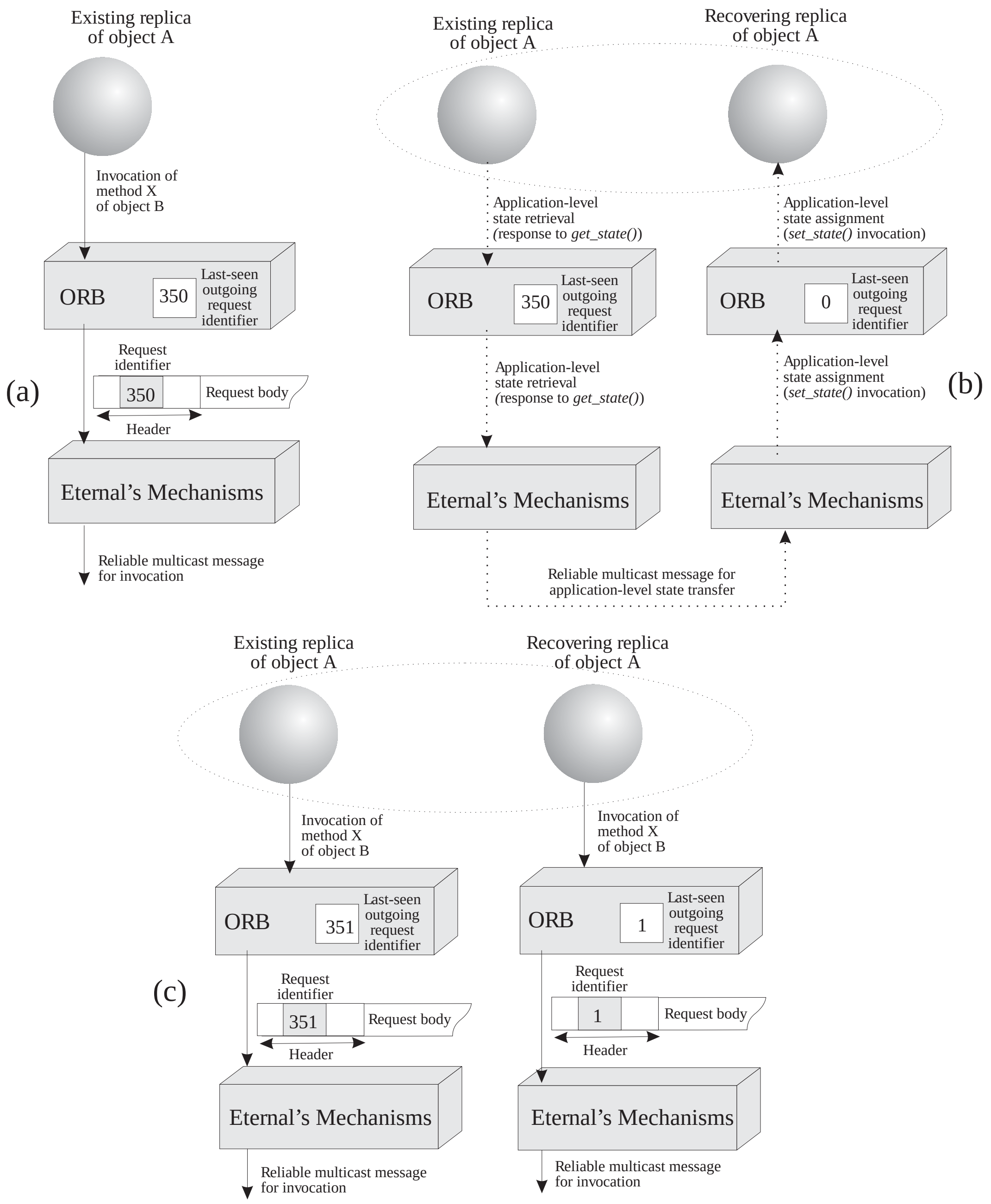

Existing replica of object A
Invocation of method X of object B
ORB
350
Last-seen outgoing request identifier
Request identifier
350
Request body
Header
Eternal's Mechanisms
Reliable multicast message for invocation
(a)
Existing replica of object A
Recovering replica of object A
Application-level state retrieval (response to get_state())
ORB
350
Last-seen outgoing request identifier
Application-level state assignment (set_state() invocation)
ORB
0
Last-seen outgoing request identifier
Application-level state retrieval (response to get_state())
Application-level state assignment (set_state() invocation)
(b)
Eternal's Mechanisms
Eternal's Mechanisms
Reliable multicast message for application-level state transfer
Existing replica of object A
Recovering replica of object A
Invocation of method X of object B
Invocation of method X of object B
ORB
351
Last-seen outgoing request identifier
ORB
1
Last-seen outgoing request identifier
(c)
Request identifier
351
Request body
Header
Request identifier
1
Request body
Header
Eternal's Mechanisms
Eternal's Mechanisms
Reliable multicast message for invocation
Reliable multicast message for invocation

Figure 4.4: Replica inconsistency due to different request identifiers from existing and recovering replicas. In this example, only application-level state is being retrieved and transferred.

Eternal provides mechanisms to handle the consistency of "pieces" of ORB-level state such as request identifiers. However, to demonstrate how replica consistency could be violated if request identifiers were not handled during recovery, consider the example of Figure 4.4. For the purpose of the example, assume that all of Eternal's Mechanisms, but for those that ensure consistent ORB-level state, are present.

Figure 4.4(a) shows an existing replica of an actively replicated object A that issues an invocation (say, of method X of object B). This request carries the request identifier 350, as issued by the ORB hosting this replica. Assume that the replica receives the response to this invocation, and that a new replica of the object A is now launched. Before delivering any further invocations, assume that Eternal's Mechanisms retrieve the application-level state from the existing replica (by invoking the *get_state*() method), multicast this state to the Mechanisms hosting the new replica. The receiving Mechanisms assign this state to the new replica (by invoking the *set_state*() method). Although Eternal, in fact, transfers ORB-level state as well as application-level state, for the purposes of this example, we will assume that the ORB-level state is not being handled.

The last outgoing invocation from the existing replica carried the request identifier 350, which its ORB "remembers." Unfortunately, because the ORB hosting the new replica does not "know" about this piece of information which it must store in its data structures, the ORB assigns the initial value 0 for the last-seen request identifier. Once the new replica is recovered, assume that both replicas now dispatch the same invocation (again, invocation of method X of object B), as shown in Figure 4.4(c). The ORB hosting the existing replica assigns the request identifier 351 to its replica's outgoing invocation, and the ORB that hosts the recovering replica assigns the request identifier 1 to its replica's outgoing invocation. The two invocations are identical in intent, and in the body of the request; they differ only in the request identifiers that they carry.

The duplicate detection that Eternal performs ensures that only one of these invocations will be delivered to the target server object B. If the delivered invocation happens to be the one with the request identifier 1, then the server object B will return an IIOP response that encapsulates this request identifier 1. Unfortunately, when this response is delivered to the ORBs hosting the two client replicas of object A, only the ORB that assigned this request identifier 1 (which happens to be the ORB hosting the recovered replica) will deliver the message to its replica. The ORB that hosts the existing replica will detect a mismatch between the request identifier (351) that it sent and the request identifier (1) that it received. Thus, the ORB will not deliver the response to the existing replica, which will be blocked, waiting forever for the server object to return a response.

Thus, if a new replica of the object is launched, the ORB that hosts the new replica must store the same request identifier as that stored by ORBs hosting existing replicas of the same object. Otherwise, even if a response is received by the ORB hosting a new client replica, the response will never reach the client replica because of a mismatch between the returned request identifier and the request identifier that the new client's ORB (incorrectly) expects.

The request identifier information is buried within the ORB. With the use of vendor-specific "hooks" into the ORB, the Eternal system could extract the current request identifier from the ORB. However, it may not always be possible to obtain such "hooks" from an ORB vendor; even if those "hooks" were provided, using them may not always be easy. Fortunately, the request identifier information is visible from outside the ORB, in the messages that are sent by the ORB. By parsing every outgoing message from the ORB, Eternal's Replication Mechanisms can discover, and store, the ORB's setting for the current request identifier. Furthermore, by ensuring that the stored value is transferred from the Replication Mechanisms hosting an existing replica to the Replication Mechanisms hosting a new replica, at the point of recovery, the Eternal system can ensure that all outgoing messages from the new replica are appropriately "patched" with the correct request identifier.

4.3.2.2 Socket Connections

The ORB handles all of the connection establishment and management on behalf of every object that it hosts. The information that it must manage includes the identifiers and the states of the sockets that an application object is currently using, the order in which these sockets were established, and the buffered messages at these sockets. In addition, the ORB must store information about the two endpoints of each socket, and the options that may be assigned to the socket (such as the TCP_NODELAY option).

An existing replica of an object will have many established socket connections that the replica uses to communicate with other objects in the system. When a new replica of the same object is launched, because the new replica has not yet communicated with other objects in the system, and because the new replica does not "know" about the connection history of the existing repica, the new replica is not likely to have the established connections of an existing replica. It is particularly important that the connections of the existing replica be set up identically for the new replica. Otherwise, it may not be possible for the ORB hosting the new replica to deliver incoming messages to the replica, or to establish connections for the new replica to communicate with other objects at a later point.

Fortunately, all of the socket establishment is handled by Eternal's Interceptor on behalf of the ORB. Thus, although the ORB "believes" that it handles all of the connections, Eternal manages connections transparently to the ORB. This enables the Replication Mechanisms hosting an existing replica to "probe" the Interceptor for the connection history of the replica. Eternal can then transfer this information to the Replication Mechanisms, and thus to the Interceptor, hosting a new replica. The Interceptor that receives this information can then mimic the connection history of the existing replica for the new replica, and establish the appropriate connections in the correct order.

As in the case of the request identifier, with the provision of vendor-specific "hooks" into the ORB, it is entirely possible to extract the connection information from the ORB itself, rather than from the Interceptor.

4.3.3 Infrastructure-Level State

Infrastructure-level state is independent of, and invisible to, the replicated object as well as to the ORB, and involves only information that Eternal deems necessary for maintaining consistent replication.

The infrastructure-level state is independent of the application-level state and the ORB-level state, and consists of the list of operation identifiers for the outstanding invocations for which the existing replicas (in the same group as the new or recovering replica) are awaiting responses. It also contains information essential for duplicate detection and garbage collection of the log, including the list of last-seen operation identifiers from every sender group. Much of this information is stored by the Logging-Recovery Mechanisms on behalf of every replica, as outlined in Section 4.2.

Because this infrastructure-level state is completely visible to Eternal (being part of what Eternal itself maintains), it is relatively easy for the Logging-Recovery Mechanisms to piggyback this state onto the application-level state and the ORB-level state that they must transfer to the Logging-Recovery Mechanisms hosting a new replica. The Logging-Recovery Mechanisms that receive the three kinds of state will assign the application-level state first, the ORB-level state next, and finally, the infrastructure-level state *before* allowing the new replica to receive or process any regular incoming invocations or responses. The retrieval, as well as the assignment, of the three different kinds of state must appear to be a single atomic action so that the state transfer of the three kinds of state occurs at a single logical point in time.

4.4 Object Quiescence

Eternal's Logging-Recovery Mechanisms ensure that all of the replicas of an object are consistent in application-level, ORB-level and infrastructure-level state. The frequency of state retrieval or checkpointing is determined on a per-replicated-object basis, by the user, at the time of deploying the application, when all other fault tolerance properties (replication style, number of replicas, location of replicas, etc) for the replicated object are also determined.

The user-specified checkpointing frequency serves only an advisory purpose. It informs Eternal's infrastructure how often the Logging-Recovery Mechanisms can issue a *get_state*() invocation on the replicated object. By no means does this guarantee that the replicated object will perform this operation immediately, and will therefore respect the checkpointing frequency. For instance, the replicated object may be in the middle of another operation, or it may be blocked waiting for a response, *etc.* In such cases, the *get_state*() will be enqueued and delivered to the replicated object in the course of time. To decide on the appropriate time to deliver the *get_state*() invocation, the Eternal system must determine the moment that the object is quiescent, and capable of receiving a new invocation.

4.4.1 Conditions for Quiescence

In general, if an object is non-quiescent, it is performing an operation after having received an invocation, and thus, its state may be undergoing modifications. If a non-quiescent object were to be checkpointed, then different replicas of the object might be at different points of the execution, and different checkpoints may be obtained for the replicated object. Thus, state *must* be checkpointed for an object only when the object is quiescent.

Additional complications arise due to *shared data between objects that are collocated within the same process.* Ideally, in the interests of replica consistency, the application should be programmed such that objects collocated within the same process are independent, self-contained entities that never share data structures that are outside of any of the objects, *e.g.*, global variables that are not member variables of any of the objects. Otherwise, even as the state (which will necessarily include the shared data) of an object within the process is being retrieved, another object collocated within the same process could be modifying the shared data.

Unfortunately, it may not always be possible to enforce such "clean" application programming. Thus, without the assurance that the application will never contain collocated application objects that share data, the Eternal system must assume that shared data may exist in the application, and must enforce strong replica consistency taking this into account.

Effectively, if an object shares data with other objects within the same process, even if the object is currently not performing any operations, it does not immediately qualify for quiescence. An object can be regarded as quiescent for the purposes of state transfer if:

Condition 1: No threads are actively executing within the object

Condition 2: No threads are actively executing that may modify data that the object shares (*e.g.*, global variables) with other collocated objects within the process

Condition 3: No synchronous invocations have been delivered to the object for which the object has not returned a response

Condition 4: All oneway invocations that have been delivered to the object have completed execution

Condition 5: The object is not blocked from receiving new invocations due to any synchronous invocation that it has issued for which it is awaiting a response.

Asynchronous one-way calls that the object has issued do not matter because the object is not waiting for a response from the call.

These conditions for quiescence are not specific only to the delivery of the *get_state*() invocation – they must be verified both for the transfer of state to and from the object and for the delivery of every new incoming invocation to the object. For strong replica consistency, *every replicated object must be quiescent at the beginning of each invocation.* With the assurance of "clean" application programming, condition 2 above can be safely eliminated because the object's state will never be dependent on shared data. However, conditions 1, 3, 4 and 5 must be enforced, regardless of the existence of shared data.

4.4.2 Implications of Quiescence Conditions

The Logging-Recovery Mechanisms cannot retrieve the state from an existing replica of an object unless the conditions in Section 4.4.1 are met. Thus, checkpoints must be coordinated among the different objects collocated within the same process. For instance, when two different objects are collocated in the same address space, *i.e.*, within the same process, and the two objects share data, either object may update the shared state independently. As a consequence of this shared state, state cannot be retrieved until both objects are quiescent according to the conditions stated above. Unfortunately, the shared state enforces some sort of dependency between the two objects, and, thus, the need for coordinated checkpointing [12, 43].

To achieve replica consistency, the moment of quiescence must be determined identically for all of the replicas of an object, regardless of the process in which these replicas are located, and regardless of the other objects with which the replicas are collocated. The implication of this is that the unit of quiescence must be identical for every replicated object for the purposes of state retrieval or assignment. Thus, the replicas of an object must be located

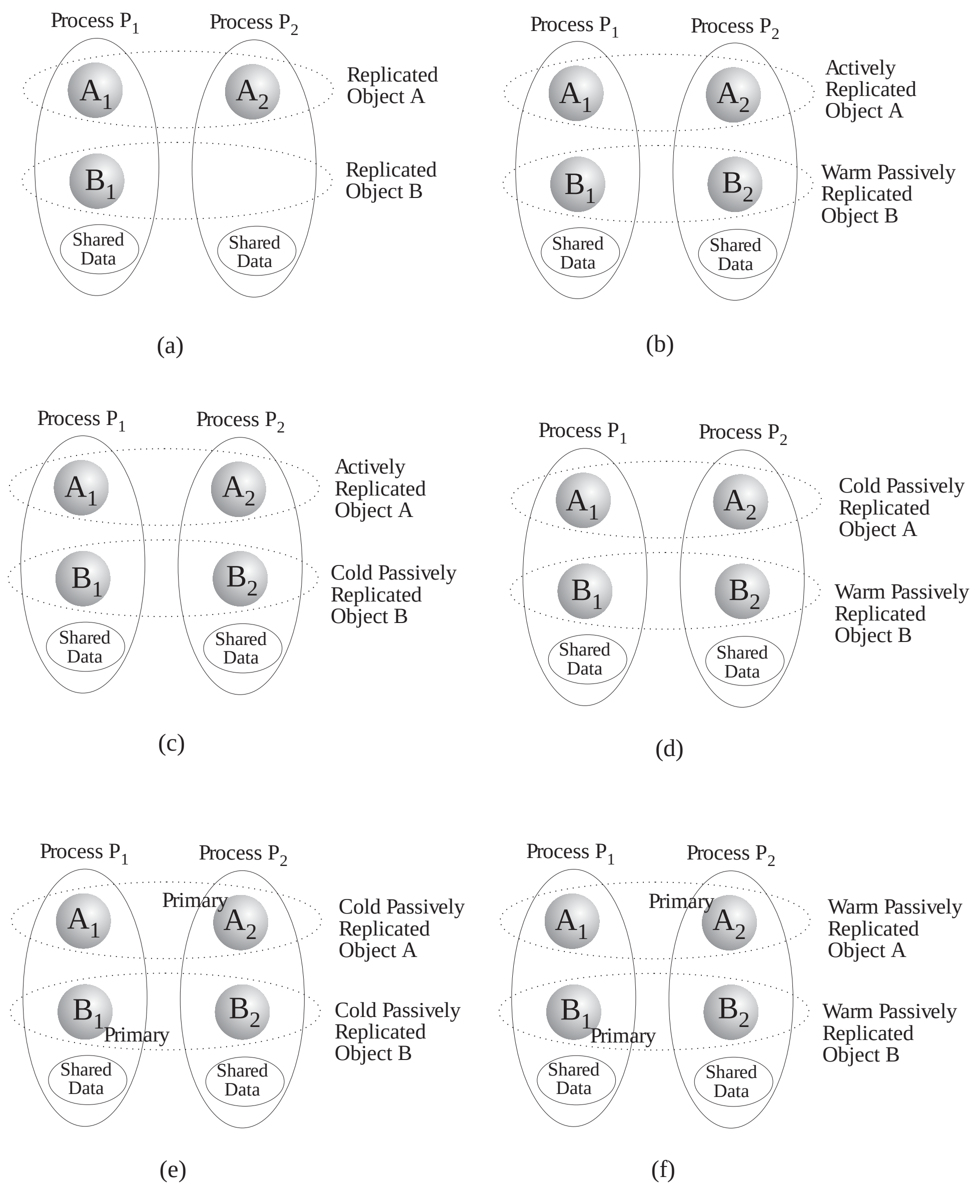

Process P1
Process P2
A1
A2
B1
B2
Shared Data
Replicated Object A
Replicated Object B
(a)
Actively Replicated Object A
Warm Passively Replicated Object B
(b)
Actively Replicated Object A
Cold Passively Replicated Object B
(c)
Cold Passively Replicated Object A
Warm Passively Replicated Object B
(d)
Primary
Cold Passively Replicated Object A
Cold Passively Replicated Object B
(e)
Warm Passively Replicated Object A
Warm Passively Replicated Object B
(f)

Figure 4.5: Combinations of replicas and replication styles that could lead to inconsistent replication when replicas collocated within the same process share data.

in identical processes, and collocated with identical replicas of other objects within those processes. As shown in Figure 4.5(a), it is not possible for a strongly consistent replicated object A to have one of its replicas A_1 collocated with a replica B_1 of object B (where B_1

shares data with A_1) in a process P_1, and another of its replicas A_2 by itself in another process P_2. If this were allowed, the moment of quiescence would be determined differently for the replicas of object A in the different processes P_1 and P_2. The differing moments of quiescence would mean that, if a $get\_state()$ invocation were dispatched to object A, replica A_1 would actually execute the invocation at a different moment in time from replica A_2. This would mean that the single $get\_state()$ invocation issued on the replicated object would result in different responses (with potentially different states contained in them) from the different replicas of the object.

Figures 4.5(b)-(f) show other combinations of collocated replicated objects with different replication styles that can result in inconsistent replication when the collocated objects share data. For instance, in Figure 4.5(e) with two collocated cold passively replicated objects with shared data, clearly the backup replicas A_1 and B_2 are not even loaded into memory. Thus, although they are shown in the figure, these backup replicas do not exist, and only the primary replicas A_2 and B_1 are performing every operation, and updating, separately in the two processes P_1 and P_2, the data shared common to both A and B. The value of this shared data is likely to differ across the two processes so that, if the primary replica A_2 fails, and A_1 must now take over as the new primary replica, the different values of the shared data in the processes P_1 and P_2 will lead to inconsistency.

Another consequence of the conditions for quiescence is that different replication styles (active replication, cold passive replication, warm passive replication) for objects collocated within the same process cannot be mixed. All objects located within the same process that share data must have the same replication style.

Thus, the replicas of object A must necessarily be located in identical processes P_1 and P_2, and collocated with the same set of objects, and with objects with the same replication style. Figures 4.6(a), (b) and (c) indicate valid possibilities for P_1 and P_2 that would enable consistent replication in the presence of data shared between collocated objects. Thus, although replication, consistency and recovery are managed on a per-replicated-object basis, in the presence of shared data, the effective unit of replication must be the process. "Peer" processes are those that contain the same set of object replicas with the same replication style; thus, the quiescence and state of peer processes will be determined identically.

Of course, the notion of peer processes must be continued to be maintained as replicas within these processes fails. Particular care must be taken in the case that a replica located within a process fails. If the failed replica is collocated with replicas of other objects within the same process address space, there is likely to be shared state. In this case, the entire process must be terminated, and all of the replicas within the process must be killed. While this may seem drastic, if such action is not taken, quiescence might be determined differently for this process than for its peer processes. Thus, the states of the operational object replicas within a process may become different from the states of the corresponding object replicas within the peers of the process.

In the interests of strong replica consistency, and without the assurance of "clean" application programming, the process is the effective unit of quiescence, state retrieval, state assignment, failure and activation.

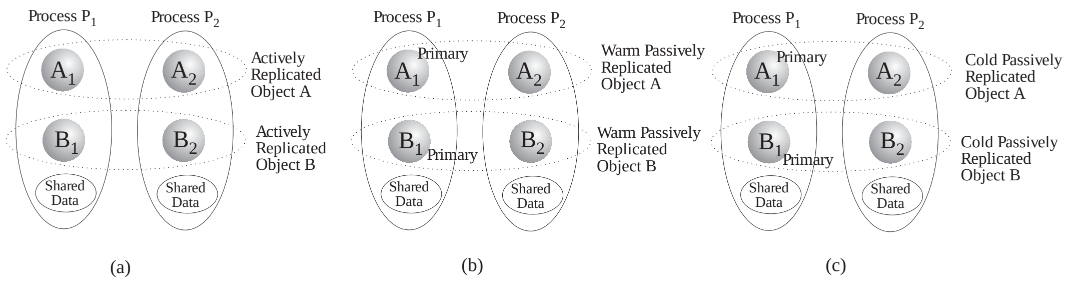

Process P_1
Process P_2
A_1
A_2
B_1
B_2
Shared Data
Actively Replicated Object A
Actively Replicated Object B
Primary
Warm Passively Replicated Object A
Warm Passively Replicated Object B
Cold Passively Replicated Object A
Cold Passively Replicated Object B
(a)
(b)
(c)

Figure 4.6: Combinations of replicas and replication styles that can ensure replica consistency when replicas collocated within the same process share data.

4.5 State Transfer

The Replication Manager (shown *above* the ORB in Figure 1.3) is responsible for the creation and the activation of new replicas of application objects. The activation of a new replica of an object must trigger the transfer of state to the new replica.

For reasons of efficient and consistent recovery, the state transfer is performed by the Logging-Recovery Mechanisms (*underneath* the ORB), rather than by the application or by the Replication Manager above the ORB. Thus, the Replication Manager above the ORB only needs to communicate the activation of a new replica to the Logging-Recovery Mechanisms to initiate recovery.

4.5.1 Generating State Transfer Invocations

Because the Logging-Recovery Mechanisms assume responsibility for state retrieval and assignment, they are the sole invoker of the `Checkpointable` interface of the replicated objects that are *above* the ORB.

The Logging-Recovery Mechanisms are located underneath the ORB, and are not implemented as CORBA objects themselves. This means that the Mechanisms do not have access to the ORB, and to a direct means of communicating with CORBA objects. However, to allow the invocation of the `Checkpointable` interface, there must be some means of communication between the application objects above the ORB and the Mechanisms underneath the ORB, without requiring the mediation of the ORB. To do this, the Eternal system exploits the Interceptor, as well as a simple IIOP Message Engine located within the Mechanisms.

Figure 4.7 shows two replicas of a replicated object A, where A_1 is an existing replica, and A_2 is a new replica. For every replica hosted by Eternal, the Interceptor first establishes a separate Unix socket from each CORBA object to Eternal's Mechanisms underneath the ORB. This socket, labelled $Sock_{ST}$ in the figure, is not used for normal IIOP message communication between replicated CORBA clients and servers. Instead, $Sock_{ST}$ is dedicated for use by the Logging-Recovery Mechanisms for the purpose of dispatching the $get\_state()$ and the $set\_state()$ invocations to the object. The figure also shows a regular socket, $Sock_{AB}$, that is used for the normal exchange of IIOP messages between replicated object A (one of whose replicas is shown in the figure) and replicated object B (not shown in the figure).

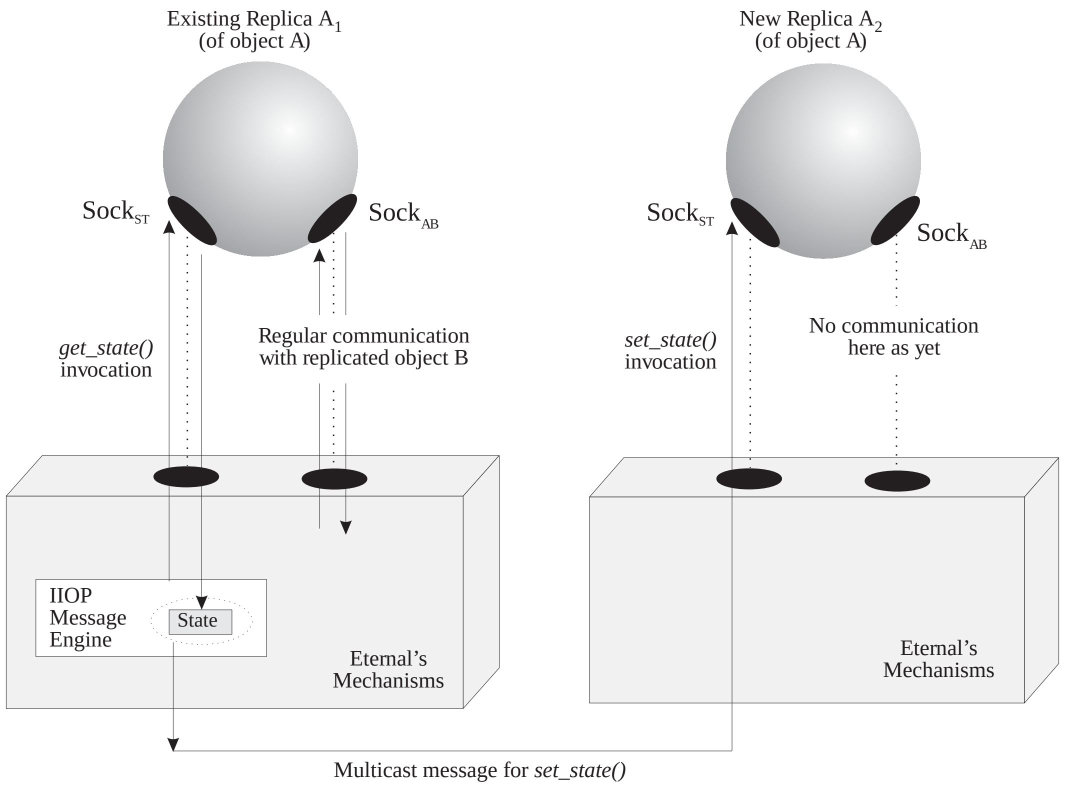

Existing Replica A1 (of object A)
New Replica A2 (of object A)
SockST
SockAB
SockST
SockAB
get_state() invocation
Regular communication with replicated object B
set_state() invocation
No communication here as yet
IIOP Message Engine
State
Eternal's Mechanisms
Eternal's Mechanisms
Multicast message for set_state()

Figure 4.7: Generating the $get\_state()$ and the $set\_state()$ invocations for state transfer.

The IIOP Message Engine that the Mechanisms use does not have any knowledge of ORBs, stubs or skeletons; it is a simple piece of code that is capable of fabricating valid IIOP invocations given the object key of the target object, the byte ordering of the system, the name of the method to be invoked, and the parameters for the method. The messages are fabricated according to the standard GIOP message formats.

For state retrieval from an object, given the object key of the target object, the IIOP Message Engine can generate a valid IIOP invocation for the $get\_state()$ invocation. By exploiting the special socket $Sock_{ST}$ that has been established by the Interceptor, the Logging-Recovery Mechanisms dispatch the Eternal-fabricated IIOP invocation to the object. The IIOP invocation can be delivered to the ORB hosting the target object which, in this case, happens to be the replica A_1 of object A. The ORB hosting A_1 receives this invocation, and because the invocation has been properly formatted according to the GIOP specifications, the ORB will deliver this to replica A_1. All that the ORB requires is that the replica A_1 actually exist, which is indeed the case because replica A_1 already exists.

The return value contained in the IIOP response to the $get\_state()$ invocation on the existing replica A_1 is the application-level state, which is received by the Mechanisms that host replica A_1. This return value must be turned into the parameter for the *set_state*() invocation on the new replica A_2. The IIOP Message Engine in the Mechanisms is capable of parsing the response to the *get_state*() invocation, extracting the application-state embedded in the body of the response, and fabricating a *set_state*() IIOP invocation with this extracted application-level state as the parameter.

In addition, the Logging-Recovery Mechanisms at A_1 piggybacks the ORB-level state and the infrastructure-level state corresponding to the existing replica A_1 onto this Eternal-fabricated *set_state*() IIOP invocation. The resulting message, containing the three kinds of state, is multicast to all of the Logging-Recovery Mechanisms hosting replicas of the object.

The Logging-Recovery Mechanisms at the new replica A_2 receive this message, and use the embedded infrastructure-level state and the ORB-level state in the message to initialize the infrastructure-level state and the ORB-level state, respectively, that the Mechanisms start to maintain for A_2. The Eternal-fabricated IIOP message for *set_state*() invocation is delivered to $Sock_{ST}$ that the Interceptor at A_2 has established for communication with the Mechanisms. When the ORB hosting A_2 receives this properly-formatted invocation message, it delivers the message to replica A_2. This is possible because the Replication Manager has already created replica A_2.

4.5.2 Synchronization of State Transfer

During recovery, the application-level state must first be retrieved from an existing replica or a log before it can be assigned to a new replica. Because of this necessary order of operations, the *get_state*() invocation must be issued to the replicated object before the *set_state*() invocation can be issued. Nevertheless, although these invocations have this order imposed on them, the point in the incoming invocation sequence that the *get_state*() appears for the existing replicas must correspond to the point in the incoming invocation sequence that the *set_state*() invocation appears from the new replica. Otherwise, the state retrieved by the *get_state*() invocation will not, in fact, be the state assigned by the *set_state*() invocation.

Thus, in the transfer of state from an existing replica to a new or recovering replica, it is particularly important that the retrieval of state from the existing replica and the assignment of the retrieved state at the new or recovering replica occur at the same *logical* point in time. The tricky issues of synchronizing the transfer of state are handled by the Logging-Recovery Mechanisms.

The *get_state*() invocation must be delivered only to the existing replicas that have the current consistent state of the replicated object; the *set_state*() invocation must be delivered only to the new replica. Nevertheless, both invocations are received, in the totally ordered sequence of multicast messages, by the Mechanisms hosting every replica of the object. However, the receipt of the invocations results in different actions, depending on whether the receiving Logging-Recovery Mechanisms host an existing or a new replica.

Figure 4.8 shows two replicas of a replicated object A, where A_1 is an existing replica, and A_2 is a new replica, and the sequence of steps in synchronizing the state transfer of the replicated object A.

At the existing replica A_1, the Logging-Recovery Mechanisms deliver the *get_state*() invocation as shown in Figure 4.8(i). However, because the new or recovering replica A_2 has not

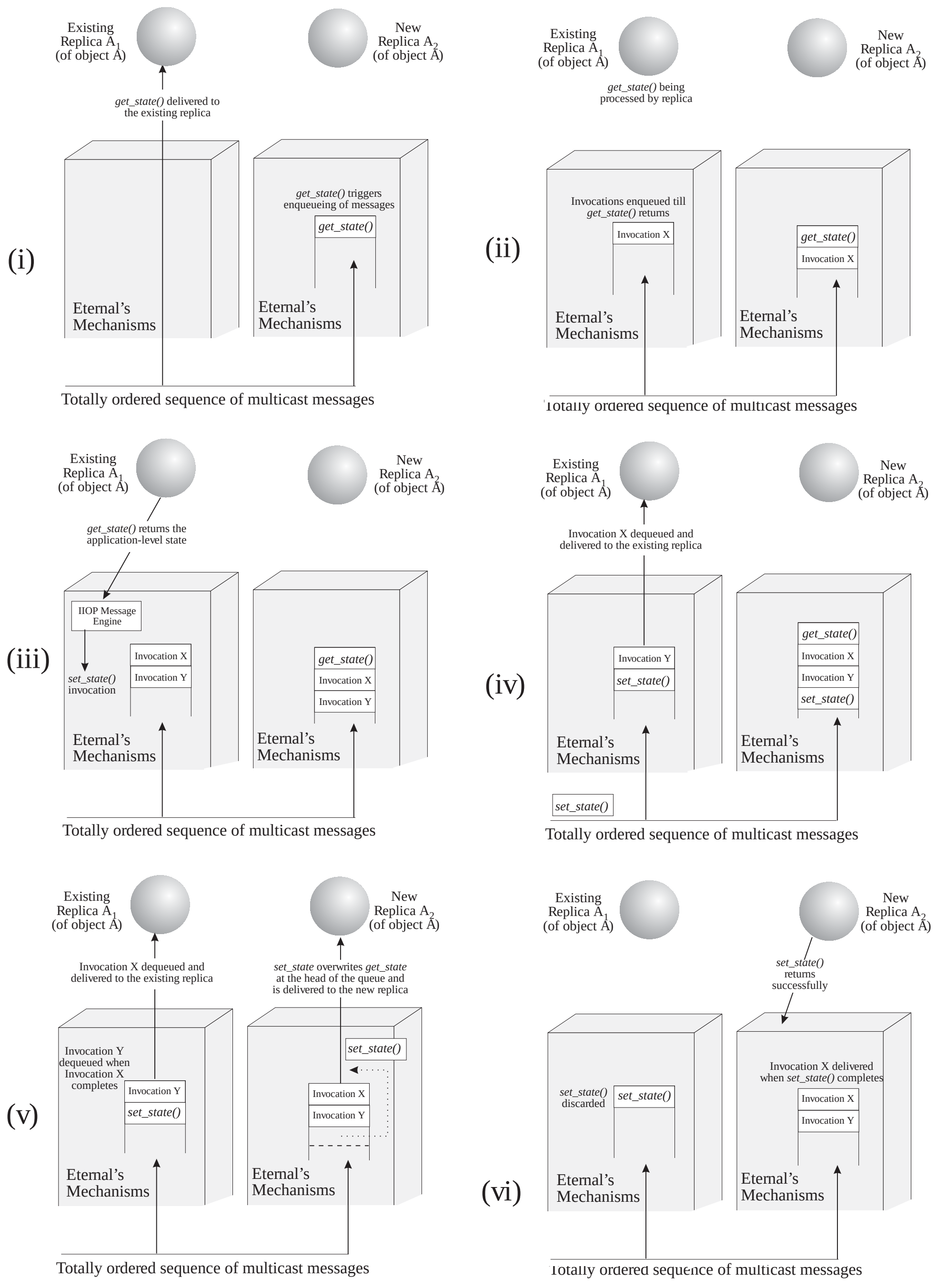

(i)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
get_state() delivered to the existing replica
get_state() triggers enqueueing of messages
get_state()
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages
(ii)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
get_state() being processed by replica
Invocations enqueued till *get_state()* returns
Invocation X
get_state()
Invocation X
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages
(iii)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
get_state() returns the application-level state
IIOP Message Engine
set_state() invocation
Invocation X
Invocation Y
get_state()
Invocation X
Invocation Y
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages
(iv)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
Invocation X dequeued and delivered to the existing replica
Invocation Y
set_state()
get_state()
Invocation X
Invocation Y
set_state()
set_state()
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages
(v)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
Invocation X dequeued and delivered to the existing replica
set_state overwrites *get_state* at the head of the queue and is delivered to the new replica
Invocation Y dequeued when Invocation X completes
Invocation Y
set_state()
set_state()
Invocation X
Invocation Y
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages
(vi)
Existing Replica A_1 (of object A)
New Replica A_2 (of object A)
set_state() returns successfully
set_state() discarded
set_state()
Invocation X delivered when *set_state()* completes
Invocation X
Invocation Y
Eternal's Mechanisms
Eternal's Mechanisms
Totally ordered sequence of multicast messages

Figure 4.8: Synchronization of state retrieval and state assignment for consistent replication.

yet been initialized with the correct consistent state of the replicated object, the *get_state*() operation is not delivered to the new replica. Instead, the receipt of the *get_state*() invocation triggers the enqueueing of incoming IIOP messages by the Logging-Recovery Mechanisms at the new or recovering replica.

While the existing replica A_1 is performing the *get_state*() operation, regular invocations (such as Invocation X shown in Figure 4.8(ii)) might arrive for the replicated object A. Because A_1 is in the middle of an operation, these incoming invocations will not be delivered to A_1. Because A_2 has not yet been initialized with the correct state, these invocations are not delivered to A_2. The Logging-Recovery Mechanisms at both replicas enqueue the regular incoming invocations.

The *get_state*() invocation completes, as shown in Figure 4.8(iii), and the IIOP Message Engine within the Mechanisms converts the response of the *get_state*() into a *set_state*() invocation, as described in Section 4.5.1. The *set_state*() invocation is multicast, along with the ORB-level state and the infrastructure-level state. Once replica A_1 returns the response to the *get_state*(), it is free to process normal invocations, and the Logging-Recovery Mechanisms that host A_1 will deliver, in the order of their arrival, any invocations (such as Invocations X and Y) that were enqueued while A_1 was processing the *get_state*().

When the *set_state*() invocation is received by the Logging-Recovery Mechanisms at the new or recovering replica, as shown in Figure 4.8(v), this invocation moves to the head of the incoming message queue (a position previously occupied by the *get_state*() message), and is delivered to A_2. The remaining enqueued messages are applied after the application-level state is transferred to the new or recovering replica, as shown in Figure 4.8(v). The *set_state*() invocation, when received by the Logging-Recovery Mechanisms hosting the existing replica A_1, will simply be discarded because A_1 already has the correct consistent state of the replicated object A.

Thus, the logged *get_state*() invocation itself is never applied to the new or recovering replica A_2; instead, its receipt by the Logging-Recovery Mechanisms is used to represent the *state synchronization point*, in the totally ordered message sequence, at which the state assignment must occur through its counterpart *set_state*() message.

4.5.3 Incremental State Transfer

The simplest mechanism for the transfer of large states between replicas of an object is to suspend operations on the object, transfer the state, and then resume operations on the object. This solution is appropriate when the state is not too large and can be transferred quickly. A drawback of this scheme is the need to stop all operations on the object until the state transfer is accomplished. More refined, though more complex, schemes allow operations to be performed on the object while a large state is being transferred [38]. Such a scheme is described below.

For active replication, one of the replicas is designated to perform the transfer. This replica does not stop processing further operations while transferring the state. Rather, it logs a *preimage* (the values of the updated parts of the state before the update) of each update that it performs. First the existing state is transferred, and then the preimages are transferred. The state initially transferred to the new replica may be incomplete, since the state may have been partially updated after the transfer, but the new replica can reconstruct

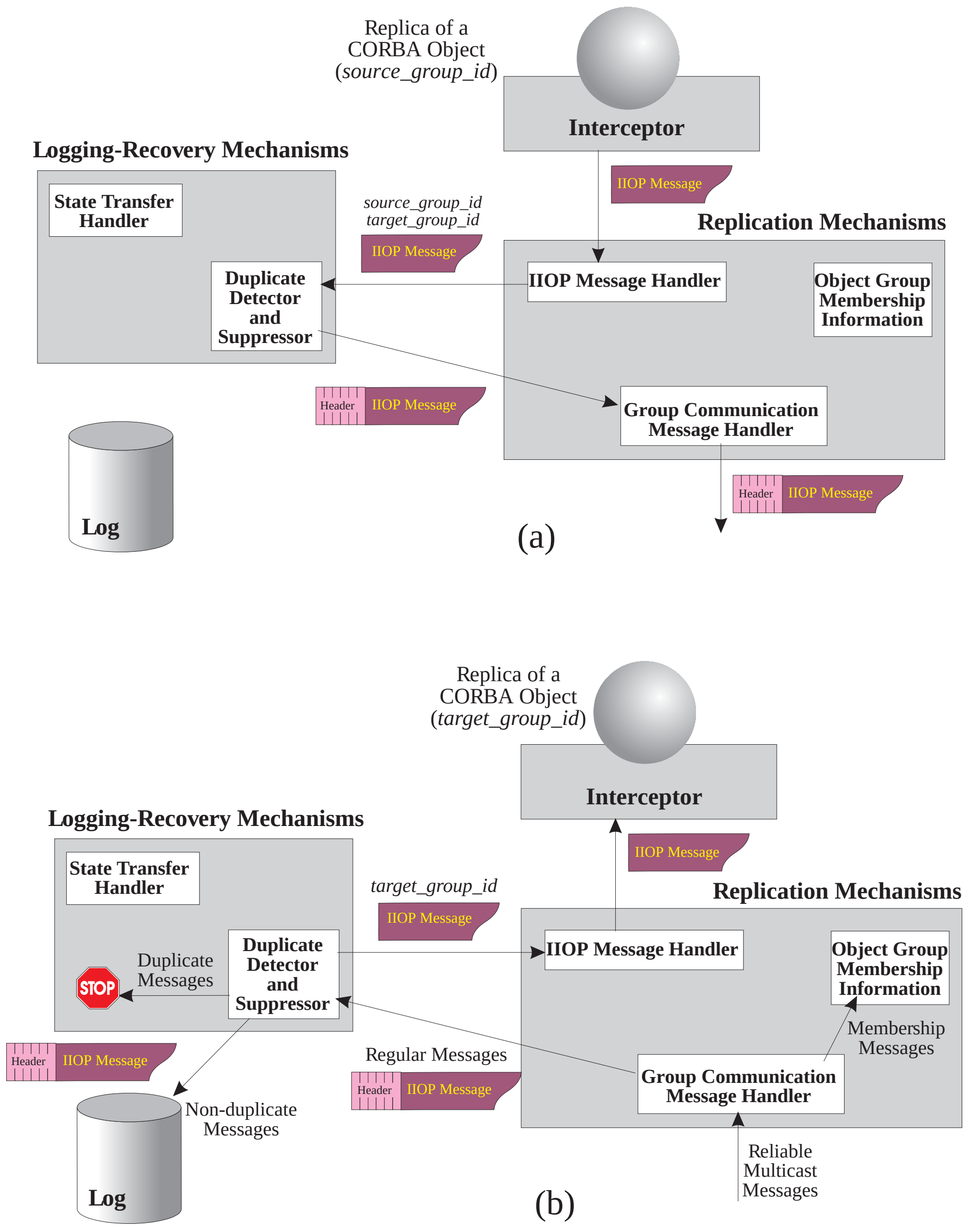


Figure 4.9: Interaction between the Replication Mechanisms and the Logging-Recovery Mechanisms of the Eternal system for (a) outgoing messages, and (b) incoming messages.

the state by applying the preimages. During the transfer, the new replica performs no operations, but rather logs all of the operations. Once the state transfer is completed, the new replica processes the operations it has logged in order to bring its state into consistency with that of the other replicas.

For passive replication, the procedure is similar, except that the *postimages* (the values of the updated parts of the state after the update) are logged and transferred, instead of the preimages, and the new or backup replicas do not log and process operations.

The advantage of this scheme is that a primary replica does not have to stop processing its messages while transferring its state. However, extra load is imposed on the primary replica since it continues processing and transfers its state simultaneously.

4.6 Interaction between the Mechanisms

The Logging-Recovery Mechanisms, operating in concert with the Replication Mechanisms, provide for the consistent object replication of the CORBA application. The interaction between Eternal's Replication and Logging-Recovery Mechanisms is shown in Figure 4.9.

For every outgoing IIOP message that it receives from the Replication Mechanisms, the Logging-Recovery Mechanisms insert, but do not record, a unique operation identifer into the Eternal-specific header of the encapsulated message. For every incoming encapsulated IIOP message, the Logging-Recovery Mechanisms use the information in the Eternal-specific header to detect and suppress duplicate messages, and pass only non-duplicate messages (along with sufficient information about the target object group) to the Replication Mechanisms for delivery to the application.

Chapter 5

Majority Voting

Applications such as those in the telecommunications industry, the medical field, and the financial sector, place stringent requirements on the reliability and security of distributed systems. These requirements arise primarily because such applications must provide continuous service and, thus, cannot be shut down. Unfortunately, it is precisely the critical nature of these applications that makes them highly vulnerable to malicious attacks.

CORBA lacks the intrinsic support to meet the requirements of such critical applications. The Eternal system extends CORBA to protect an existing CORBA application against intrusions or accidents that damage some portion of the distributed system, or faults that occur within the system [44].

5.1 Secure Totally Ordered Reliable Multicast

To support critical applications that must tolerate arbitrary faults, including those that arise from the incorrect behavior of the application itself, the Eternal system employs *active replication with majority voting* for every client and server object of the application. Unfortunately, the guarantees of the Totem protocols are insufficient to ensure effective majority voting. Instead, Eternal exploits the SecureRing Protocols [27, 28] to communicate the operations to the replicas, to maintain the consistency of the states of the replicas of each application object and, in particular, to provide the higher levels of fault tolerance required for majority voting.

5.1.1 The SecureRing System

The SecureRing Protocols provide the properties necessary for majority voting, and allow for detection and removal of faulty processors. The protocols consist of a message delivery protocol, a processor membership protocol, and a Byzantine fault detector as shown in Figure 5.1. As with the Totem protocols, the services of the underlying SecureRing Protocols are made available to Eternal's Mechanisms through the object group interface, which hides the complexity of the protocols from the Eternal system.

Imposed on the communication medium is a *logical* ring with a token that controls the multicasting of messages. To multicast a message on the ring, a processor must hold the token. The processor membership consists of all of the processors that participate in the sending and receiving of messages on this logical ring.

SecureRing's message delivery protocol provides secure reliable totally ordered delivery of messages that are multicast by the processors in the system. This ensures that all of the non-faulty processors in the system deliver the same messages in the same total order and that, if any non-faulty processor delivers a message, then, no non-faulty processor delivers a *mutant* message having the same identifier but different contents.

SecureRing's processor membership protocol reconfigures the system when one or more processors exhibit faulty behavior. The membership protocol exchanges information via special processor membership messages, and reaches agreement on and installs a new membership consisting of apparently non-faulty processors that are able to communicate with each other.

SecureRing provides a local Byzantine-fault detector on each processor that monitors the messages sent by the message delivery and processor membership protocols, and provides its output to the membership protocol in the form of a list that contains the processors currently suspected as being faulty by the local Byzantine fault detector. Such faults are classified as omission faults and commission faults. An *omission* fault occurs when a process omits to send a required message to one or more non-faulty processes, or repeatedly fails to acknowledge the receipt of a required message. A *commission* fault occurs when a processor sends mutant tokens, or sends a token that is improperly formed.

To ensure that a malicious processor cannot masquerade as another processor, the token-holding processor digitally signs the token before multicasting it. In addition, to tolerate faults involving message corruption during transit, the token transmitted by a processor contains the message digests of the messages that the processor multicasts while it holds the token.

The SecureRing protocols employ a public key cryptosystem such as RSA [59] in which each processor possesses a private key known only to itself, and with which it can digitally sign messages. Each processor can obtain the public keys of other processors to verify the signed tokens that it receives. The protocols also employ a message digest function such as MD4 [58] in which an arbitrary length input message is mapped to a fixed length message digest.

Tolerating malicious faults comes with a cost, primarily due to the computation time associated with the generation and the verification of the digital signatures on the token. However, because only tokens (and not messages) are digitally signed, and because multiple messages can be multicast with every acquisition of the token by a processor, a single digital signature on the token can cover multiple messages.

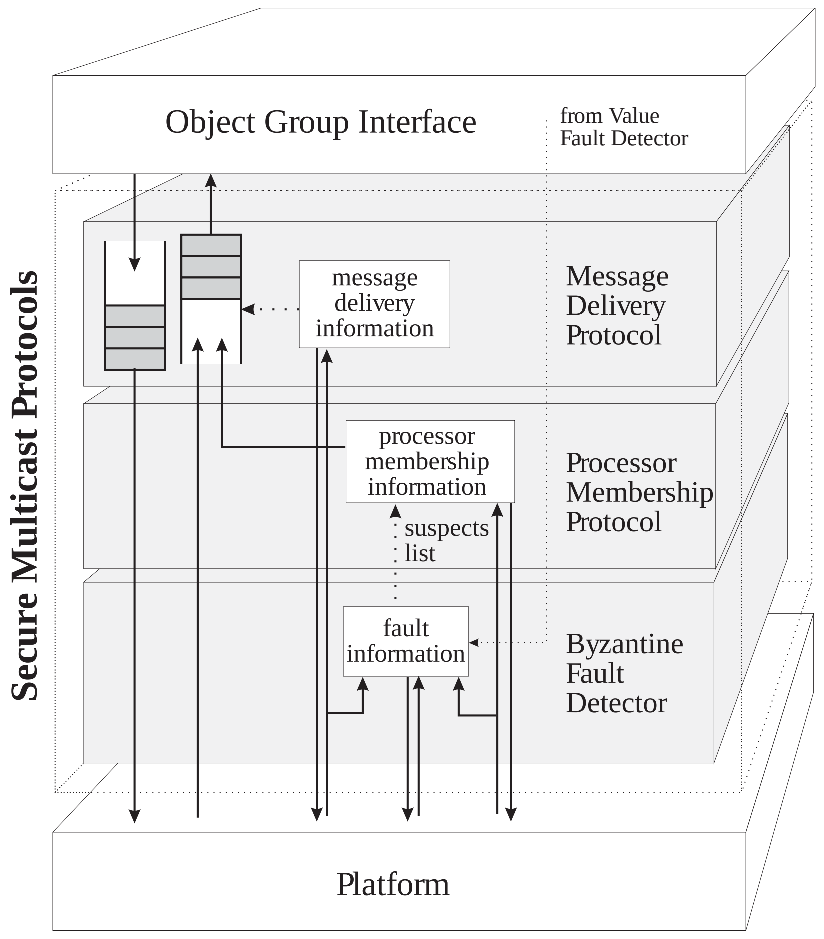

Secure Multicast Protocols
Object Group Interface
from Value Fault Detector
message delivery information
Message Delivery Protocol
processor membership information
Processor Membership Protocol
suspects list
fault information
Byzantine Fault Detector
Platform

Figure 5.1: The SecureRing group communication system.

5.2 Active Replication with Majority Voting

Critical applications that must tolerate value faults, in addition to crash faults, require the use of majority voting on all messages to be delivered to the application objects. Because the Eternal system aims to support applications that must tolerate arbitrary faults, the Replication Mechanisms employ majority voting on the invocations and responses delivered to each application object.

To enable majority voting, a number of “copies” of the same invocation (response) must serve as inputs to the voting algorithm, where each “copy” is sent by a different replica of the same client (server) object. The voting algorithm uses these “copies” as inputs to produce a single output “copy” of the invocation (response) for delivery to the application. To collect these “copies”, Eternal employs active replication for every object of the application.

In an environment where value faults are absent, these "copies" correspond to the duplicate messages discussed in Section 3.4, and the first non-duplicate receiveed message can be safely delivered without the need for majority voting. However, in an application subject to arbitrary faults, any one of multiple client (server) replicas could send a corrupted invocation (response). Thus, while the "copies" of the invocation (response) received at the target object represent the same operation, they may differ in value or content, and may not, in fact, be duplicates of each other.

5.2.1 Support for Majority Voting

To send messages to a target replicated object, the Replication Mechanisms do not need to know the number, or the location, of the target replicas. However, to perform majority voting on the copies of an invocation or response, the Replication Mechanisms need to know how many copies of the invocation or response to expect. Thus, the voter within the Replication Mechanisms hosting a target server (client) replica needs to know the current number of replicas (the current degree of replication) of the actively replicated sender client (server).

One way that the voters could obtain this information, is by having the Mechanisms on every processor join a special *base group* that serves to disseminate membership changes and membership information. For every new replica that is created for an object, or for every replica of an object that dies, the new degree of replication (after the addition or removal of the replica is effected) is reported to the base group, along with the identity (in the form of the object group identifier that is uniquely assigned to the replicated object) of the replicated object whose degree of replication has changed. Thus, the Replication Mechanisms on every processor are always aware of the current number of replicas in every replicated object in the system.

In the Eternal system, as shown in Figure 5.2, object group membership messages (generated by the object group interface due to the addition or removal of a replica) are received, but not forwarded to the application, by every member of the base group. On receipt of such messages, the Replication Mechanisms update the membership information that they must maintain to perform majority voting. The base group poses some difficulties in terms of scalability; it requires the Replication Mechanisms on every processor to belong to a group. An alternative to the base group is to have the Eternal-specific header of every multicast message contain the degree of replication of the replicated sender object. For the rest of the text, however, we will discuss the base group approach.

The voting algorithm is deterministic and produces the same result for each invocation (response) at every server (client) replica. For each invocation (response) from a client (server) object, when the voter receives a majority of copies that it verifies as being identical in value, the voting algorithm produces a single result, which the Replication Mechanisms then deliver as an invocation (response) of (to) the target server (client) replica. For every output result that the voter produces, the Replication Mechanisms discard all subsequently received messages that are copies of the delivered invocation (response).

Figure 5.2 shows the interaction between the object group interface and the voting mechanisms. The Replication Mechanisms on a processor maintain a single voter for every object group (replicated object) one of whose members (replicas) it hosts. The Replication

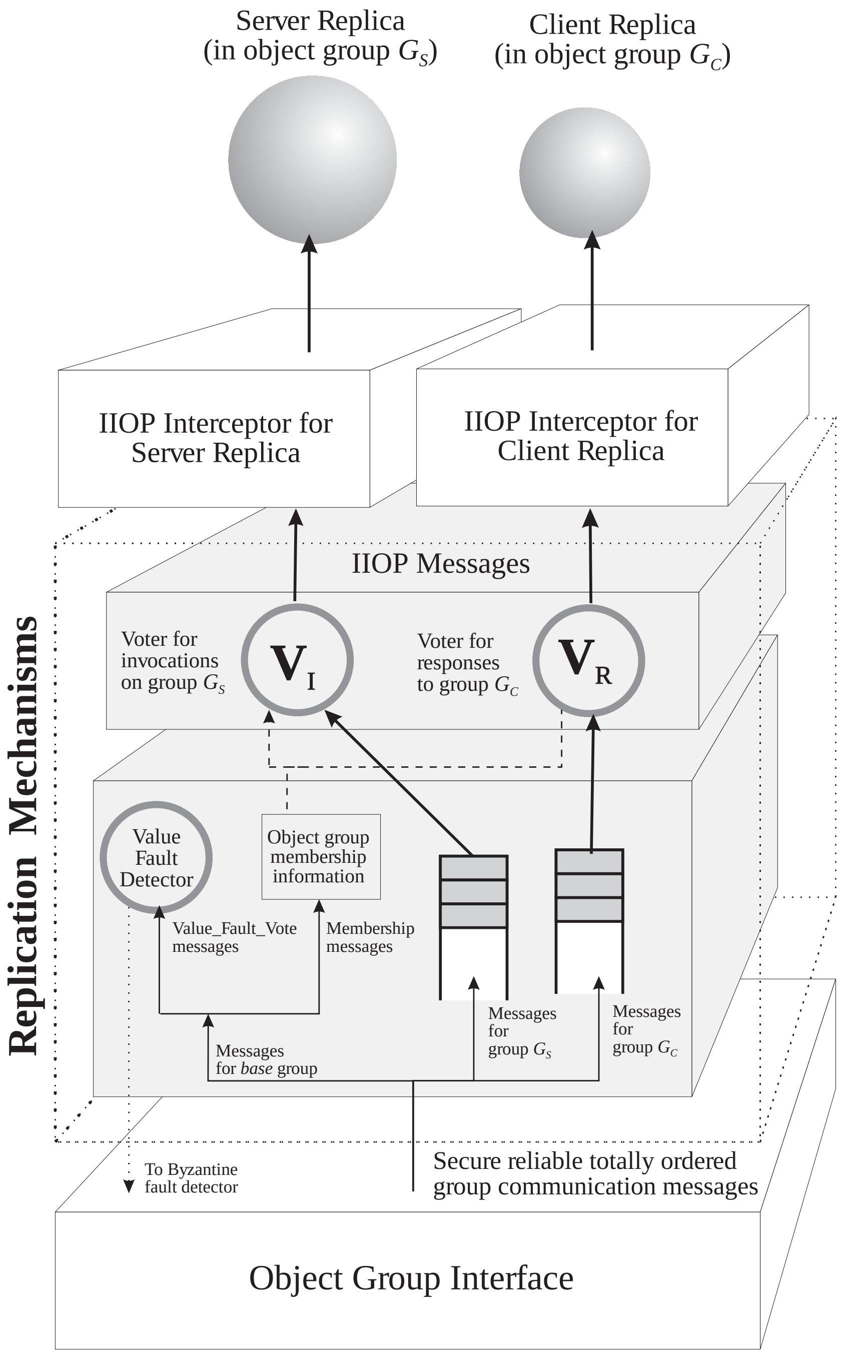

Server Replica
(in object group G_S)
Client Replica
(in object group G_C)
IIOP Interceptor for Server Replica
IIOP Interceptor for Client Replica
IIOP Messages
Voter for invocations on group G_S
V_I
Voter for responses to group G_C
V_R
Replication Mechanisms
Value Fault Detector
Object group membership information
Value_Fault_Vote messages
Membership messages
Messages for *base* group
Messages for group G_S
Messages for group G_C
To Byzantine fault detector
Secure reliable totally ordered group communication messages
Object Group Interface

Figure 5.2: Eternal's Replication Mechanisms enhanced with support for majority voting on received invocations and responses.

Mechanisms receive all of the secure reliable totally ordered multicast messages destined for the replicas that it hosts, and filters the messages based on their object group identifiers of the target replicas. The Mechanisms then pass on to the voters only those messages that are destined for the target replica with whose group identifier the voter is associated. The voters then execute the voting algorithm to decide on the delivery, to their designated replicas, of the messages that they receive.

5.2.2 Input Majority Voting

Input majority voting occurs on incoming invocations to replicated server objects. As shown in Figure 5.3, in the case of input majority voting, the Replication Mechanisms at each of the server replicas detect the copies of each invocation using the invocation identifiers in the messages. The mechanisms for the detection of these copies and the assignment of the invocation identifiers follow the rules outlined in Section 3.4 for duplicate detection. In this case, these mechanisms serve to collect the copies of every distinct invocation to be fed to the invocation, or input, voter.

By virtue of its membership in the base group, the Replication Mechanisms at each of the server replicas know the current degree of replication, r_c, of the replicated client object or, alternatively, the number of copies of each invocation to expect from the replicated client object. From its knowledge of r_c, the Replication Mechanisms can determine the majority number of client replicas.

As shown in Figure 5.2, when the Replication Mechanisms hosting a replica of a server object (with object group identifier G_S) receive the copies of an invocation destined for group G_S, the Mechanisms dispatch these copies to the local voter V_I assigned for voting on invocations destined for group G_S. The voter V_I collects these copies, votes on them, and produces a single invocation that is then delivered to the server replica through the Interceptor.

5.2.3 Output Majority Voting

Output majority voting occurs on incoming responses to replicated client objects. As shown in Figure 5.4, in the case of output majority voting, the Replication Mechanisms at each of the client replicas detect the copies of each response using the response identifiers in the messages. The mechanisms for the detection of these copies and the assignment of the response identifiers follow the rules outlined in Section 3.4 for duplicate detection. In this case, these mechanisms serve to collect the copies of every distinct response to be fed to the response, or output, voter.

By virtue of its membership in the base group, the Replication Mechanisms at each of the client replicas know the current degree of replication, r_s, of the replicated server object or, alternatively, the number of copies of each response to expect from the replicated server object. From its knowledge of r_s, the Replication Mechanisms can determine the majority number of server replicas.

As shown in Figure 5.2, when the Replication Mechanisms hosting a replica of a client object (with object group identifier G_C) receive the copies of a response destined for group G_C, the Mechanisms dispatch these copies to the local voter V_R assigned for voting on

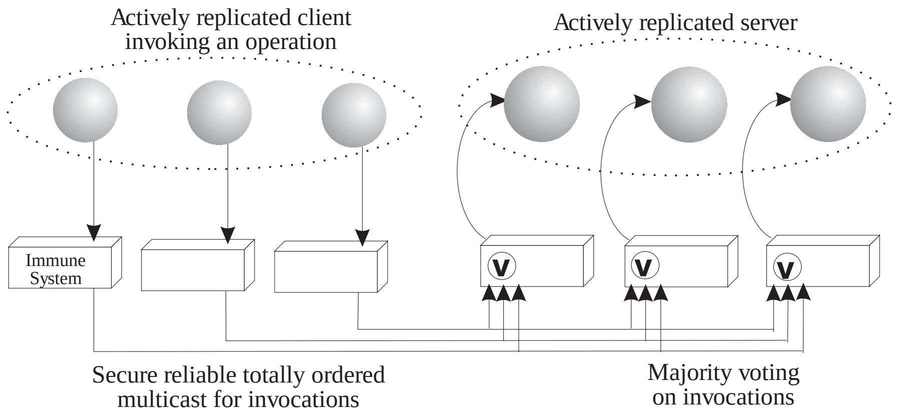

Actively replicated client invoking an operation
Actively replicated server
Immune System
V
V
V
Secure reliable totally ordered multicast for invocations
Majority voting on invocations

Figure 5.3: Active replication with input majority voting on invocations.

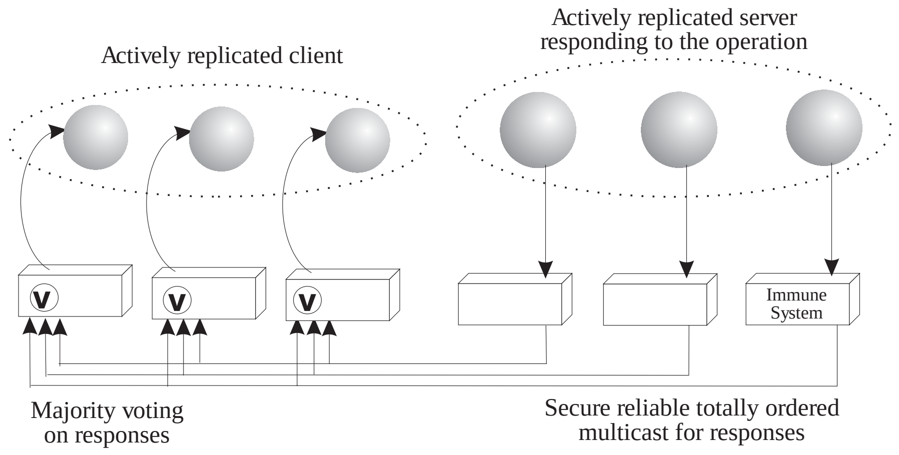

Actively replicated client
Actively replicated server responding to the operation
V
V
V
Immune System
Majority voting on responses
Secure reliable totally ordered multicast for responses

Figure 5.4: Active replication with output majority voting on responses.

responses destined for group G_C. The voter V_R collects these copies, votes on them, and produces a single response that is then delivered to the client replica through the Interceptor.

5.2.4 Value Fault Detection

As shown in Figure 5.2, the voter V_I (V_R) within the Replication Mechanisms can detect an incorrect value of an invocation (response) sent by a corrupted client (server) replica. To ensure that a value fault due to a corrupt replica is handled as a malicious processor fault, the Replication Mechanisms on *every* processor within the system (and not merely those that first detected the value fault through their voters V_I or V_R) must vote *locally* on the same set of invocations and independently reach the same conclusion.

To achieve this, each Replication Mechanisms use a *value fault detector*, as shown in Figure 5.2. A voter V_I (V_R), on detecting an incorrect value of an invocation (response), multicasts to the base group, a `Value_Fault_Vote` message encapsulating the set of copies of the invocation (response) on which it voted. This special message is delivered to the value fault detector within the Replication Mechanisms on every processor. Each of these value fault detectors compares this set of copies to determine the identity of the corrupt client (server) replica and, thus, the identity of the processor that hosts that replica.

The value fault detector on every processor then communicates the identity of the corrupt processor to the local Byzantine fault detector (within the SecureRing Protocols on the same processor) through a `Value_Fault_Suspect` message. This special message is not intended to be transmitted over the network, and is used solely for private notifications by the value fault detector to the local Byzantine fault detector. Thus, every Byzantine fault detector in the system will receive the same local notifications. The mechanisms of the SecureRing protocols can employ these notifications to decide on, and effect, the removal of the corrupt processor. The performance of the Eternal system for active replication with majority voting is given in Chapter 8.

Chapter 6

Multithreading

Some of the issues surrounding replica consistency and multithreading have been addressed for fault-tolerant systems that are not based on CORBA. In [64], a technique is employed to track and record the nondeterminism due to asynchronous events and multithreading. While the nondeterminism is not eliminated, the nondeterministic executions are recorded so that they can be replayed to restore replica consistency in the event of rollback.

Many commercial ORBs and CORBA applications are multithreaded, primarily because multithreading can yield substantial performance advantages and concurrency. However, the specification of multithreading in the CORBA 2.2 standard [54] does not provide any guarantees on the order in which a multithreaded ORB dispatches incoming operations. In particular, the specification of the Portable Object Adapter (POA), which is a key component of the CORBA standard, provides no ordering guarantees on the mechanisms that the ORB or the POA use to dispatch requests across threads. The following is an excerpt from the specification of the POA ([54], pages 9-12) in the CORBA standard:

> "There are no guarantees that an ORB or POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads. There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind."

Thus, the ORB or the POA may dispatch several requests for the same object within multiple threads at the same time, and these threads may not necessarily execute in the order in which the requests were received by the ORB or the POA.

In addition to ORB-level threads, the CORBA application may itself be multithreaded, with the application-level thread scheduling having been determined by the application programmer. The application programmer must ensure the correct sequencing of operations, and must prevent thread hazards. Careful application programming can ensure thread-safe operations within a single replica of an object, without leading to race conditions or deadlocks within that replica; however, it does not guarantee that threads and operations will be dispatched, and will execute, in the same order *across all of the replicas* of the object. The application programmer cannot be expected to understand the detailed issues surrounding fault tolerance. Making the application programmer responsible for concurrency control and ordering of dispatched operations and threads in replicated objects is unacceptable for maintaining strong replica consistency in a fault-tolerant system.

This chapter describes how the Eternal system addresses these challenges through the mechanisms [48] that it provides for guaranteeing consistent replication in the face of one specific source of nondeterminism, namely, multithreading in the ORB or the application.

6.1 Replication of Multithreaded Objects

Several different concurrency models [61] are supported by current commercial ORBs. These include thread-per-request, where a separate thread is spawned for each new invocation on an object, and thread-per-object, where a single thread executes all invocations on an object. Most practical CORBA applications consist of processes that contain multiple objects, each having possibly multiple threads. Objects that are collocated within the same process may access and update shared data; thus, irrespective of the threading model used by the ORB, multiple threads may exist within each process. Because a server object may itself assume the role of a client, we do not distinguish between the problem of multithreading for client objects and server objects.

We use the term *MT-domain* to refer to any CORBA client or server that supports multiple (application-level or ORB-level) threads which may access shared data, and that contains one or more CORBA objects. The MT-domain abstraction is independent of the concurrency model of the ORB, of the role of the MT-domain as a client or as a server, as well as of the commercial ORB that hosts the MT-domain.

In Sections 6.1.1 and 6.1.2, we provide examples that illustrate how replica inconsistency can arise in the active and the passive replication of MT-domains, respectively. In addressing the specific problems posed by multithreading with regard to replica consistency, we assume that all of the mechanisms of Section 3.1 that guarantee replica consistency for single-threaded objects are already present, *i.e.*, messages are delivered to the ORB in a totally ordered sequence, duplicate operations are detected and suppressed, and state transfers are provided for recovering replicas.

While these mechanisms suffice to guarantee replica consistency in the absence of multithreading, they serve only to facilitate, but not to guarantee, replica consistency when either the ORB or the application is multithreaded. In particular, the totally ordered multicasts ensure only that the ORBs that host the various replicas receive the same messages in the same order. They do not guarantee that the ORBs will dispatch these incoming messages onto the threads of the replicas in the same order.

We assume further that other sources of nondeterminism, *e.g.*, system calls (such as local timers) that return processor-specific information, are handled by other mechanisms, such as the additional sanitizing interposers described in Section 2.2.3.1. We also assume

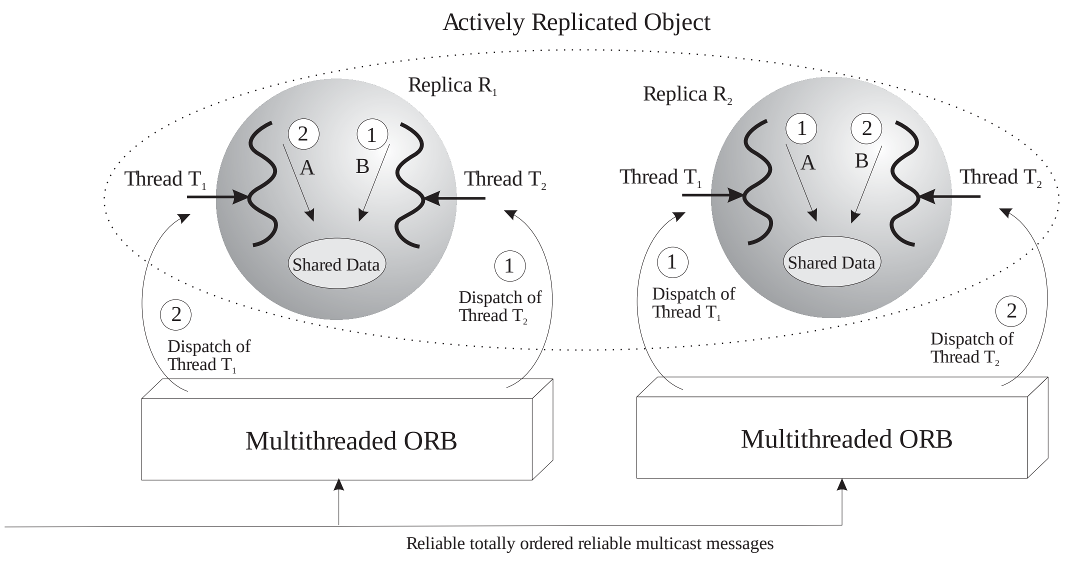


Figure 6.1: Inconsistency with active replication of multithreaded objects.

that all replicas of the application are located on the same type of platform, i.e., use the same vendor's ORB,[1] run over the same operating system with identical processing speed, memory, etc. In addition, we assume the deterministic behavior of the operating system. With these assumptions, any replica inconsistency that arises in the examples discussed in this chapter can be attributed solely to the multithreading of the ORB or the application.

6.1.1 Inconsistent Active Replication

For active replication, strong replica consistency means that, at the end of each operation invoked on the replicas of an object, each of the replicas of the object has the same state. The example shown in Figure 6.1 illustrates one instance of the replica inconsistency that can arise when only the mechanisms for consistent replication for single-threaded ORBs have been provided.

In the figure, R_1 and R_2 are active replicas of the same MT-domain. The ORB at each replica receives messages in the same order, but dispatches two threads T_1 and T_2 to perform different incoming operations on the replicas. The two threads are simultaneously active within each replica, and can access and update shared data. Suppose that threads T_1 and T_2 issue update operations A and B, respectively, on the shared data within the process, and that operation B (A) is executed before operation A (B) in replica R_1 (R_2). Even if the MT-domain is programmed to avoid race conditions and other thread hazards, the order of operations in the two replicas of the MT-domain differs in this case, and, thus, their states

[1]In the current state of the art, although ORBs are implemented according to the same set of specifications, their implementations differ sufficiently that replicas of an object cannot run on different ORBs without introducing some degree of non-determinism.

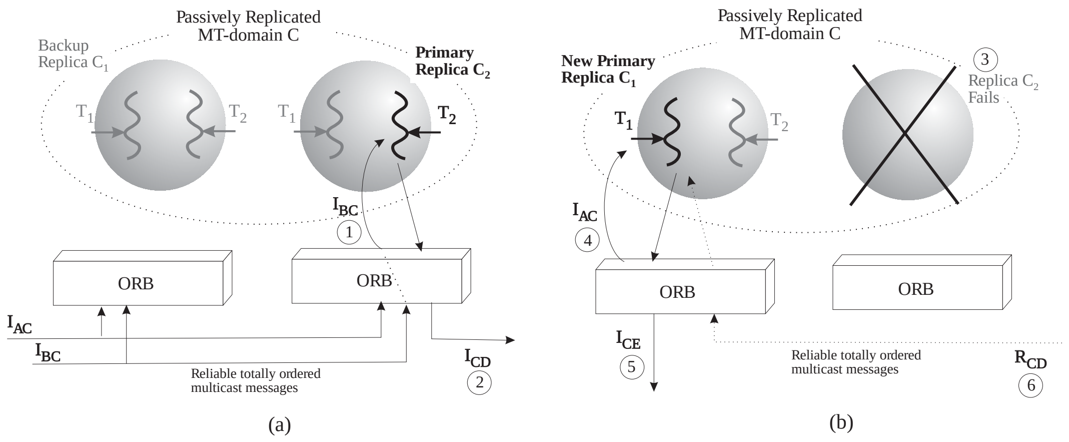

Passively Replicated MT-domain C
Backup Replica C1
Primary Replica C2
T1
T2
IBC
1
ORB
ORB
IAC
IBC
Reliable totally ordered multicast messages
ICD
2
(a)
Passively Replicated MT-domain C
New Primary Replica C1
3
Replica C2 Fails
T1
T2
IAC
4
ORB
ORB
ICE
5
Reliable totally ordered multicast messages
RCD
6
(b)

Figure 6.2: Inconsistency with passive replication of multithreaded objects when (a) the primary replica is initially operational, and (b) the primary later fails and the backup replica becomes the new primary replica.

are likely to be inconsistent at the end of the update operations. Such replica inconsistency can also arise when the ORB initially dispatches only a single thread, which later spawns other threads within the same replica.

Of course, in a typical distributed application, the inconsistent state of a replicated MT-domain may affect all other objects that it communicates with, even if these other objects are inherently single-threaded. Thus, the replica inconsistency of one replicated object can propagate to affect the consistency of other replicated objects.

6.1.2 Inconsistent Passive Replication

For passive (primary-backup) replication, strong replica consistency means that, at the end of each state transfer, each of the replicas of an object has the same state. Passive replication does not suffer from the problems caused by multithreading only in the very restrictive case of a single unreplicated client invoking a primary-backup MT-domain that itself does not invoke any other objects. In all other cases where a MT-domain is passively replicated, the potential for inconsistency in the states of the replicas of the MT-domain exists. The example shown in Figure 6.2 illustrates the problem of replica inconsistency for passive replication.

In the figure MT-domain C is passively replicated, with primary replica C_2 and backup replica C_1. Objects A, B, D and E (not shown) with which C interacts are not necessarily multithreaded or replicated, for that matter. The example focusses on the passive replication of MT-domain C, and how C's multithreading results in the inconsistency of its replicas.

The invocation I_{AC} (I_{BC}) of object A (B) on MT-domain C requires thread T_1 (T_2) to be dispatched. If thread T_1 (T_2) is dispatched, MT-domain C will issue the nested invocation I_{CE} (I_{CD}) to object E (D). Thus, MT-domain C invokes two different nested operations through its two threads, and must obtain results from both operations.

Consider the following sequence shown in Figure 6.2:

1. The primary replica C_2 is initially operational. The ORB hosting C_2 dispatches thread T_2 to handle invocation I_{BC} first.
2. Thread T_2 issues a nested invocation I_{CD} on object D.
3. The primary replica fails before handling invocation I_{AC}. The backup replica C_1 becomes the new primary replica for MT-domain C.
4. Multithreaded ORBs can dispatch operations and threads in any order, not necessarily the received total order of operations. The ORB hosting the new primary replica C_1 dispatches thread T_1 to handle invocation I_{AC} first.
5. Thread T_1 issues a nested invocation I_{CE} on object E.
6. Before the new primary's ORB handles invocation I_{BC}, object D returns the response R_{CD} to the old primary's nested invocation I_{CD}. The receiving ORB delivers this response to C_1, which is unable to handle this response to the nested invocation (I_{CD}) that it has no knowledge of having issued.

The inconsistency arises precisely because of the nondeterministic behavior of multithreaded ORBs, and the lack of specification of the order of dispatch of the operations that such ORBs receive.

6.2 Enforcing Determinism

A MT-domain is the basic unit of replication for multithreaded applications in the Eternal system. To preserve replica consistency for MT-domains, the Eternal system provides mechanisms that govern the order in which the threads (and operations) are dispatched within each replica of a MT-domain, over and above the total order in which the messages containing the operations are delivered to the ORB.

The Eternal system *enforces* deterministic behavior within a MT-domain by allowing only a *single logical thread of control*, at any point in time, within each replica of the MT-domain. Although multiple threads may exist in a MT-domain, all of them must be related to (and required for the completion of) the single operation that "holds" the logical thread-of-control. Furthermore, at most one of these threads can be actively executing; all of the other threads must be suspended or awaiting a response.

The Eternal system controls the dispatching of threads and operations within every replicated MT-domain, transparently both to the objects and threads within the MT-domain, and to the multithreaded ORB that hosts the MT-domain. To achieve this, the Eternal system employs a deterministic *operation scheduler*[2] that is inserted into the address space of every replica of a MT-domain, and that maps incoming invocations to the thread-of-control within the replica, or enqueues unrelated invocations for later dispatch.

[2]The scheduling of operations for replica consistency is orthogonal to the real-time scheduling of operations. The operation scheduler described here does not factor in any considerations for real-time operation.

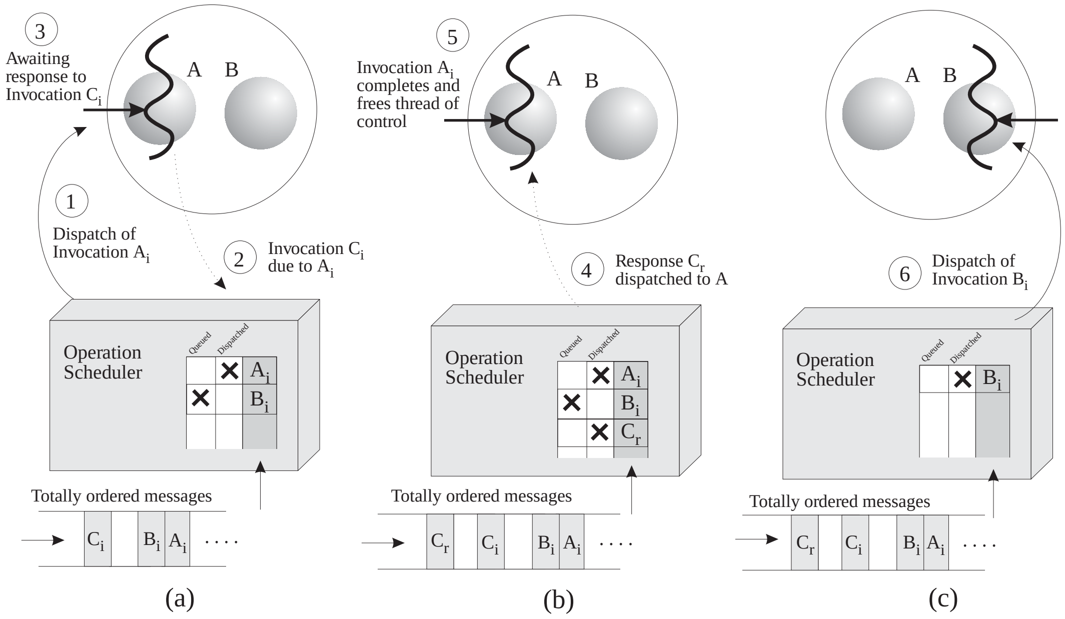

3
Awaiting response to Invocation C_i
A B
1
Dispatch of Invocation A_i
2
Invocation C_i due to A_i
Operation Scheduler
Queued
Dispatched
A_i
B_i
Totally ordered messages
C_i B_i A_i ....
(a)
5
Invocation A_i completes and frees thread of control
A B
4
Response C_r dispatched to A
Operation Scheduler
Queued
Dispatched
A_i
B_i
C_r
Totally ordered messages
C_r C_i B_i A_i ....
(b)
A B
6
Dispatch of Invocation B_i
Operation Scheduler
Queued
Dispatched
B_i
Totally ordered messages
C_r C_i B_i A_i ....
(c)

Figure 6.3: Sequence of actions of the operation scheduler at a replica of a MT-domain.

The scheduler dictates the creation, activation, deactivation and destruction of threads, within the replica of a MT-domain, as required for the execution of the current operation "holding" the logical thread-of-control. The scheduler is inserted into a position such that it can override any thread or operation scheduling performed by either the nondeterministic multithreaded ORB within the replica, or by the replica itself.

Operations are mapped identically onto the logical thread-of-control at all of the replicas of a MT-domain, thereby ensuring that the same operation "holds" the thread-of-control at each replica at any logical point in time. To enable this, the Eternal system ensures that the scheduler at each replica of a MT-domain receives the same sequence of totally ordered messages containing invocations and responses destined for the MT-domain. Based on this incoming sequence of messages, the scheduler at each replica decides on the immediate delivery, or the delayed delivery, of the messages to that replica. At each replica of a MT-domain, the scheduler's decisions are identical and, thus, operations and threads within the MT-domain are dispatched identically at each replica. Consequently, all of the replicas of a MT-domain associate the same operation with their logical thread-of-control at any point in time. Thus, the logical thread-of-control is identically determined for the replicated object as a whole.

6.3 Scheduling for Consistency

While the MT-domain model may seem somewhat restrictive in terms of the effective concurrency achieved in the application, those restrictions are necessary to achieve replica consistency for replicated multithreaded CORBA applications. To ensure a single logical thread-of-control within the MT-domain, the scheduler may delay or reschedule invocations and responses on a MT-domain. This is necessary because another operation can assume the MT-domain's logical thread-of-control only when the current operation within the MT-domain completes.

Thus, the scheduler enqueues, in the order of their arrival, all incoming invocations and responses that are unrelated to the current operation or the logical thread-of-control. The next operation to be scheduled on the MT-domain, upon release of the thread-of-control, is the first operation that has been enqueued or, in the absence of enqueued operations, the next operation that the scheduler receives.

Figure 6.3 shows a replica of a MT-domain, along with its operation scheduler, and the sequence of actions of the MT-domain's scheduler for the given totally ordered messages. All of the replicas of the MT-domain are forced to behave identically, as the example illustrates.

The MT-domain in this example consists of two objects A and B, each capable of supporting a thread. Invocations A_i, B_i and C_i are destined for objects A, B and C, respectively (object C is in some other MT-domain not shown in the figure). The invocation A_i gives rise to a nested invocation C_i; the invocation B_i is independent of both A_i and C_i.

At the start of the sequence of actions in Figure 6.3(a), there is no operation executing in the replica of the MT-domain, and the thread-of-control is free to be assumed by the next operation. Thus, the operation scheduler delivers the invocation A_i (which occurs first in the total order of messages), leading to a thread executing within object A. The scheduler assigns the MT-domain's thread-of-control to the logical operation represented by the invocation A_i until A_i completes.

The invocation A_i on object A leads to the subsequent invocation C_i on object C, and A_i can complete only when the response C_r to the invocation C_i is returned to object A. Because the thread-of-control has been assigned to A_i, the scheduler delivers to the MT-domain only those incoming invocations and responses that correspond to A_i. In this case, the only message in the total order that is related to A_i is C_r. Note that the invocation B_i is independent of A_i. Thus, although B_i has been received by the scheduler ahead of C_r in the total order of messages, it is not delivered to its target object B until the thread-of-control is released by A_i. To deliver only those invocations related to the thread-of-control, the scheduler requires some means of recognizing, and relating, the operations contained in the totally ordered messages. Identifiers for associating operations with the thread-of-control are discussed in Section 6.3.2.

Figure 6.3(b) shows the receipt of the response C_r by the MT-domain, and its delivery to object A, when the scheduler determines that its delivery is appropriate.

After processing the response C_r, object A completes the invocation A_i and the thread-of-control again becomes available. The scheduler also garbage collects threads that were used by the thread-of-control. The MT-domain's scheduler reassigns the thread-of-control to the next invocation B_i. The MT-domain's scheduler then delivers the invocation B_i, leading to a new thread of execution within the target object B, as shown in Figure 6.3(c).

This delaying of invocation B_i in favor of the response C_r (although B_i precedes C_r in the total order of operations) does not itself introduce any inconsistency between the replicas of the MT-domain. The reason is that the operation scheduler at each of the replicas of the MT-domain arrives at the same scheduling decision regarding the delivery of C_r before B_i.

Replica consistency is thus maintained as a result of the deterministic behavior *across all* of the replicas of a MT-domain through the totally ordered messages that they receive, as well as the deterministic behavior *within each* replica of the MT-domain through the identical scheduling of distinct operations onto a single thread-of-control.

Replica determinism, unfortunately but inevitably, reduces the degree of concurrency within the application. However, if objects are indeed independent of each other, and do not share data, they can be assigned to different processes and Eternal's operation scheduler will schedule them concurrently without restriction. The memory protection between processes, provided by the operating system, ensures that objects in different processes do not share data (unless they explicitly use shared memory techniques), and are indeed independent.

6.3.1 Remote Callbacks

A remote callback operation may lead to multiple nested remote invocations (as opposed to local procedure calls) on the same MT-domain, each of which must be delivered and executed in order for the operation to complete. Thus, for such a remote callback operation, the single logical thread-of-control is realized through multiple threads within the MT-domain, at most one of which is actively executing, while the others are suspended or awaiting a response. In the example of Figure 6.3, the operation A_i is not a remote callback on the MT-domain because the execution of A_i did not lead to further remote invocations on the *same* MT-domain.

Figure 6.4 shows the interaction between a replicated MT-domain X and a replicated MT-domain Y. Here, invocation I_1 on object A, when dispatched to the logical thread-of-control in X, results in the invocation I_2 of object C in the MT-domain Y. The invocation I_2 leads to a further nested invocation I_3 on object B within the MT-domain X. I_1 requires that I_2 completes, and I_2 requires that I_3 completes. Thus, the invocation I_3 must be allowed to proceed inside every replica of X and return a response to object C within every replica of Y. Because the scheduler ensures identical behavior at all of the replicas of a MT-domain, it suffices to consider replicas X_1 and Y_1 (shown in the figure) of the MT-domains X and Y, respectively.

To permit a second invocation I_3 on a MT-domain X_1 that already has the invocation I_1 pending a response, the scheduler for X_1 needs to verify that the second invocation is a *descendant* of the first (*parent*) operation and that the parent operation is suspended. A descendant of a particular parent operation is a nested invocation that arises from the execution of the parent operation, and that must be allowed to execute in order for the parent operation to complete. A descendant invocation may be a remote callback operation. In this example, invocations I_2 and I_3 are descendants of the parent invocation I_1, with I_3 being a remote callback on the MT-domain X_1.

After the scheduler for X_1 determines that I_3 is a descendant of I_1 and that the thread for I_1 is suspended, awaiting a response (in this case from object C in Y_1), it proceeds to activate a thread to handle invocation I_3. If the thread executing I_1 is not suspended

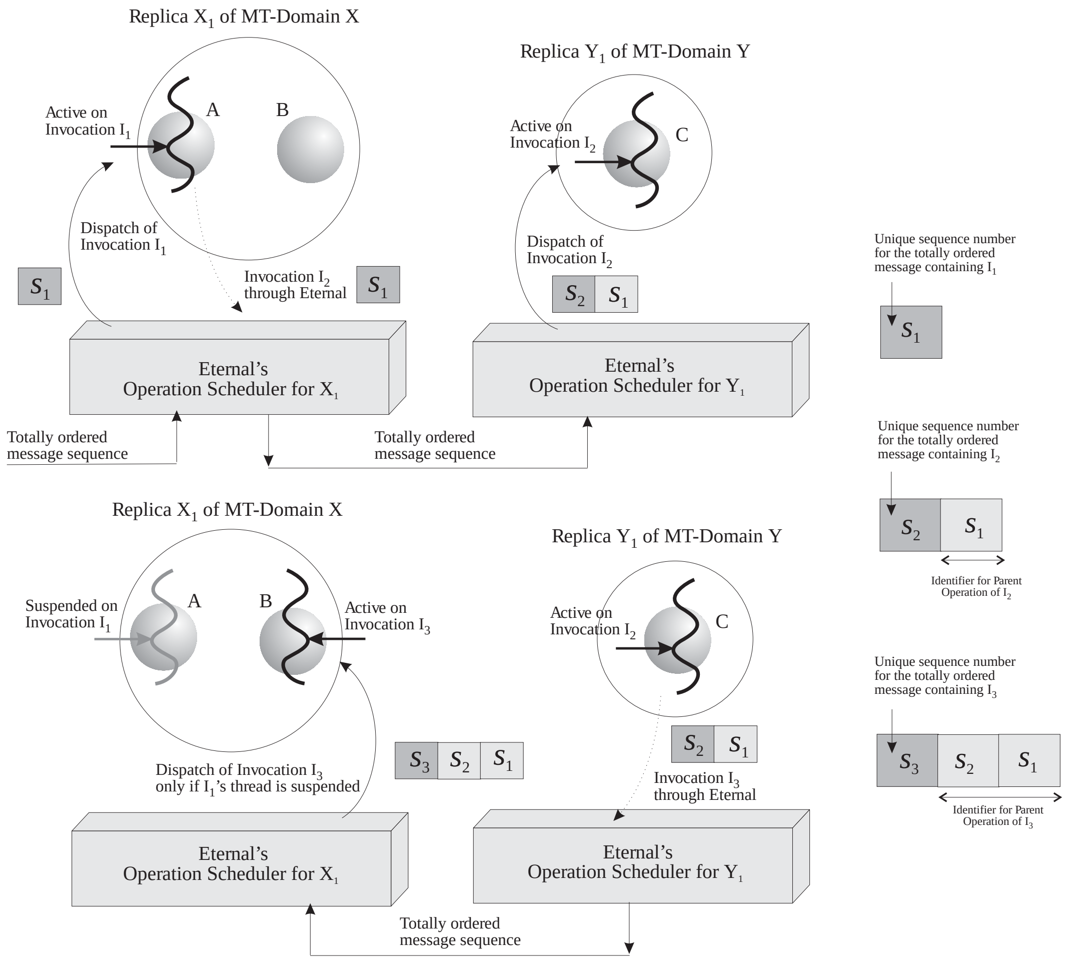

Replica X1 of MT-Domain X
Active on Invocation I1
A
B
Dispatch of Invocation I1
S1
Invocation I2 through Eternal
S1
Eternal's Operation Scheduler for X1
Totally ordered message sequence
Totally ordered message sequence
Replica Y1 of MT-Domain Y
Active on Invocation I2
C
Dispatch of Invocation I2
S2 S1
Eternal's Operation Scheduler for Y1
Unique sequence number for the totally ordered message containing I1
S1
Unique sequence number for the totally ordered message containing I2
S2 S1
Identifier for Parent Operation of I2
Replica X1 of MT-Domain X
Suspended on Invocation I1
A
B
Active on Invocation I3
Dispatch of Invocation I3 only if I1's thread is suspended
S3 S2 S1
Eternal's Operation Scheduler for X1
Replica Y1 of MT-Domain Y
Active on Invocation I2
C
S2 S1
Invocation I3 through Eternal
Eternal's Operation Scheduler for Y1
Totally ordered message sequence
Unique sequence number for the totally ordered message containing I3
S3 S2 S1
Identifier for Parent Operation of I3

Figure 6.4: Remote callback operations on a MT-domain. Scheduling identifiers assigned by the operation scheduler enable the detection of remote callbacks.

before I_3 is allowed to start executing, the states of the objects within X_1 may become inconsistent due to the multiple threads being active. The logical thread-of-control within X_1 is still associated with the first invocation I_1 because all other operations within the MT-domain are direct descendants of I_1. Once the invocation I_3 completes and B returns a response to the invoking object C, the scheduler for X_1 disposes of the thread for I_3.

A descendant remote callback on a MT-domain can generate further descendant remote callback invocations on the same MT-domain. Thus, the MT-domain scheduler must maintain a stack of invocations dispatched in the MT-domain. Every remote callback descendant invocation is "pushed" onto the stack when it is dispatched onto the MT-domain, and "popped" off the stack when it completes. The invocation at the top of the stack must complete before any of the others below it in the stack can complete.

6.3.2 Scheduling Identifiers

To handle nested remote callback operations, the operation scheduler uses *scheduling identifiers* that allow the scheduler to associate parent and descendant operations at the MT-domain level. These scheduling identifiers are internal to, and examined by, Eternal's operation schedulers, and are never seen by the CORBA application or the ORB.

At the point that it dispatches an invocation I_q onto the replica of the MT-domain that it controls, the operation scheduler assigns I_q the scheduling identifier $s_q s_p$, where s_q is the sequence number of the message containing I_q, and s_p is the scheduling identifier of the parent, if any, of I_q.

For the example of Figure 6.4, the MT-domain schedulers assign the identifiers s_1, $s_2 s_1$ and $s_3 s_2 s_1$ to the invocations I_1, I_2 and I_3, respectively. In this case, s_1 and s_2 (s_3) are (is) uniquely assigned by the scheduler at every replica of the MT-domain X_1 (Y_1). Furthermore, the same unique identifiers are generated within every replica of the MT-domain X_1 (Y_1) because these identifiers are derived, in an identical manner, from the totally ordered messages that the operation scheduler at each replica receives.

To detect an incoming remote callback, the operation scheduler uses the scheduling identifier to determine if the invocation is a descendant of any operation that has been invoked, and is awaiting a response within the MT-domain. For instance, the scheduler at X_1 detects that invocation I_3 is a descendant of I_1 due to the presence of I_1's identifier s_1 in I_3's identifier $s_3 s_2 s_1$. Once the scheduler verifies that an operation is indeed a descendant, it waits for the currently executing thread-of-control within the MT-domain to suspend itself, and then dispatches the descendant operation.

6.3.3 Scheduling Algorithm

To dispatch an operation to the thread-of-control, or to delay an operation that may lead to replica inconsistency, each operation scheduler for a MT-domain replica maintains:

- The scheduling identifier s_D, and semantics (synchronous or asynchronous), of the current operation I_D being executed by the logical thread-of-control T_D within the MT-domain. The scheduling identifier s_D is used by the operation scheduler to detect remote callback invocations. The scheduler compares the scheduling identifer of every incoming operation with s_D to determine any descendants of I_D. If an incoming message is a descendant of I_D, or a response to an invocation issued by the MT-domain, the operation scheduler dispatches it when all of the threads within the MT-domain are suspended, or awaiting a response.

- A dispatch queue Q_{op} of operations (invocations, responses and state transfer messages) waiting to be assigned to the thread of control when it becomes available. When the current operation I_D completes, the operation scheduler can dispatch a new operation from Q_{op}. Operations that are not related to the thread-of-control in the MT-domain are enqueued in the total order in which the operation scheduler receives them. Operations that are descendants of I_D are scheduled for execution, in the order of their arrival, ahead of all operations that are not descendants of I_D.

```
switch (Reason for Activation)

    // The thread-of-control for the MT-domain is
    // available to be assigned to a new operation.
    case THREAD_OF_CONTROL_RELEASED:
        if (operation queue $Q_{op}$ is not empty)
            $I_D$ = operation at the head of the dispatch queue $Q_{op}$
            $s_D$ = scheduling identifier for $I_D$
            if (thread pool $Q_{thr}$ is empty)
                Create new threads into $Q_{thr}$
            endif
            $T_D$ = first available thread in $Q_{thr}$
            Dispatch operation $I_D$ onto the thread $T_D$
            Push $I_D$ onto stack of re-entrant descendant invocations
        endif
        return
    endcase

    // A new operation intended for the MT-domain is
    // delivered in the totally ordered messages
    case INCOMING_INVOCATION_OR_RESPONSE:
        Insert incoming message at the end of the dispatch queue $Q_{op}$
        return
    endcase

    // The thread-of-control for the MT-domain is suspended on the
    // operation $I_D$. In addition to the original thread $T_D$,
    // numDesc threads could be suspended due to any uncompleted
    // remote callback descendant operations of $I_D$. All of these
    // threads are awaiting responses.
    case THREAD_OF_CONTROL_SUSPENDED:
        if (dispatch queue $Q_{op}$ is not empty)
            if (dispatch queue $Q_{op}$ has descendants of $I_D$)
                $I_{stackTop}$ = re-entrant descendant at the top of the stack
                Increment numDesc
                $I_{numDesc}$ = first enqueued descendant of $I_{stackTop}$ in $Q_{op}$
                if ($I_{numDesc}$ completes operation $I_{stackTop}$)
                    Remove $I_{stackTop}$ from the stack of operations
                else
                    Push $I_{numDesc}$ onto the stack of operations
                endif
                $i_{numDesc}$ = scheduling identifier for $I_{numDesc}$
                if (thread pool $Q_{thr}$ is empty)
                    Create new threads into $Q_{thr}$
                endif
                $T_{numDesc}$ = first available thread in $Q_{thr}$
                Dispatch operation $I_{numDesc}$ onto the thread $T_{numDesc}$
            endif
        endif
        return
    endcase
endswitch
```

Figure 6.5: Algorithm executed by the operation scheduler each time it is activated. The operation scheduler is associated with a MT-domain D, whose logical thread-of-control T_D executes the operation I_D with scheduling identifier s_D.

- A stack of the descendant remote callbacks that have been dispatched onto threads within the MT-domain and that are awaiting responses. When the thread-of-control becomes available, the stack is empty. The first operation I_D to be pushed onto the stack is the one that assumes the thread-of-control T_D. Subsequently, descendants of I_D that are remote callbacks on the MT-domain are also pushed onto the stack. The invocation on top of the stack is removed from the stack as soon as it completes, and an invocation is pushed onto the stack as soon as it is dispatched to a thread within the MT-domain.
- A thread pool Q_{thr} of pre-spawned threads to avoid the overhead of thread creation with every new dispatch of an operation to a thread. This thread pool is used purely for efficiency, rather than for correctness.

The operation scheduler at every replica of a MT-domain executes the deterministic algorithm, shown in Figure 6.5, to schedule operations within the replica that it controls. The execution of this algorithm is triggered by the occurrence of any of the following events:

- The release of the MT-domain's thread-of-control when the current operation completes, allowing the next operation to be dispatched
- The suspension of all threads within the MT-domain, in anticipation of receiving a response, allowing the delivery of a received response, or a new descendant distributed callback invocation
- The delivery of a totally ordered invocation or response message to the operation scheduler, requiring the scheduler to decide if the message should be enqueued, scheduled or dispatched.

6.4 Implementation in Eternal

As shown in Figure 6.6, the Eternal system transparently replicates the objects, and the MT-domains, of the application. For every replica of a MT-domain or an object, the Interceptor transparently captures its IIOP invocation and response messages, which were originally destined for TCP/IP, and diverts them instead to the Replication Mechanisms. The Replication Mechanisms perform the encapsulation (retrieval) of IIOP messages to (from) the messages of the underlying reliable totally ordered multicast group communication system. In addition, the Replication Mechanisms implements mechanisms for the detection and suppression of duplicate invocations and duplicate responses, and for state transfer and recovery, as described in Chapter 3 and Chapter 4.

The Eternal system provides an *operation scheduler* within the address space of each replica of a MT-domain. Although each replica has its own scheduler, all of the schedulers for the replicas of a MT-domain reach identical scheduling decisions. This deterministic behavior of the schedulers ensures replica consistency for every MT-domain within the application.

The operation scheduler must operate at a level that allows it to govern the concurrency within each object of the MT-domain, irrespective of the ORB's multithreading policies. The operation scheduler must receive all of the totally ordered operations, and decide on their

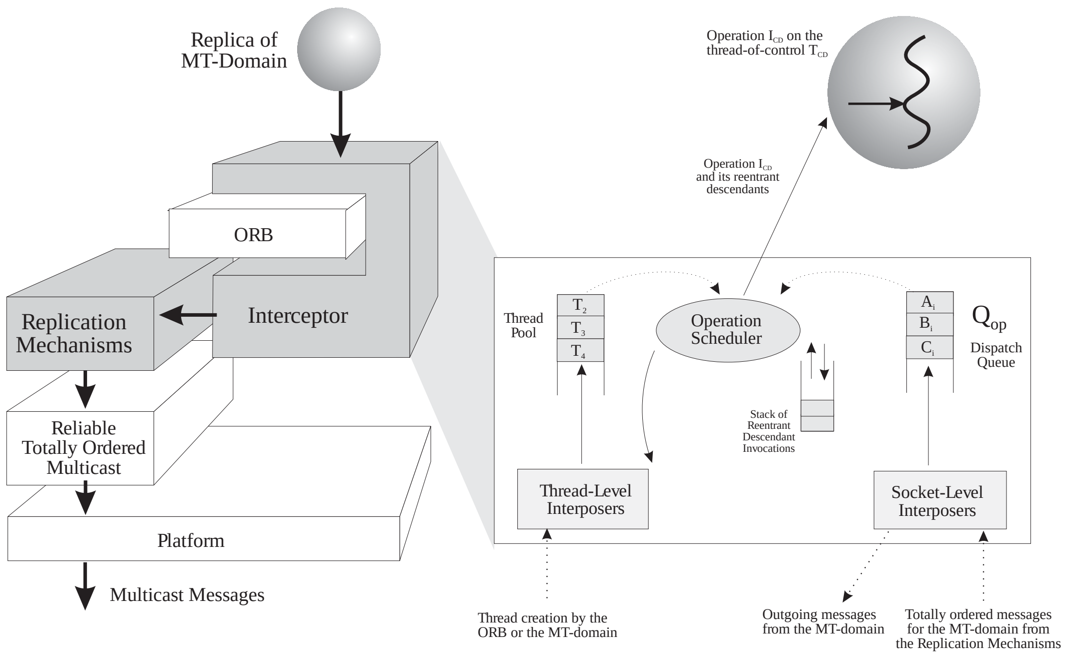

Replica of MT-Domain
ORB
Replication Mechanisms
Interceptor
Reliable Totally Ordered Multicast
Platform
Multicast Messages
Operation I_{CD} on the thread-of-control T_{CD}
Operation I_{CD} and its reentrant descendants
Thread Pool
T_2
T_3
T_4
Operation Scheduler
A_i
B_i
C_i
Q_{op}
Dispatch Queue
Stack of Reentrant Descendant Invocations
Thread-Level Interposers
Socket-Level Interposers
Thread creation by the ORB or the MT-domain
Outgoing messages from the MT-domain
Totally ordered messages for the MT-domain from the Replication Mechanisms

Figure 6.6: Implementation of the MT-domain operation scheduler of the Eternal system using the Interceptor, which is transparently co-located with the replica of the MT-domain.

delivery to the application. In the Eternal system, the operation scheduler is implemented at the level of the Interceptor, as shown in Figure 6.6.

The transparency of the Interceptor has the added advantage that the operation scheduler can perform its function without the modification of the ORB or the application. The Interceptor also enables the scheduler to overide any dispatching or threading performed by the ORB or the application, without either of them being aware of the scheduler's existence.

By exploiting the socket library interposer of the Interceptor, the operation scheduler can receive, transparently, all of the operations destined for the replica of the MT-domain.

The operation scheduler exploits the thread-level interposer of the Interceptor to provide alternative implementations of some of the thread library routines in order to enable the MT-domain's operation scheduler to determine the status of the MT-domain's thread-of-control, as well as to control the creation, dispatch and destruction of threads spawned within the MT-domain.

Thus, the operation scheduler must examine every IIOP message, that it receives through the totally ordered messages from the Replication Mechanisms, to determine if it contains a method invocation or response. The combination of the socket-level and the thread-level interposers ensures that the dispatch of threads that execute operations within a MT-domain is dictated solely by the operation scheduler, rather than by the MT-domain or by the ORB.

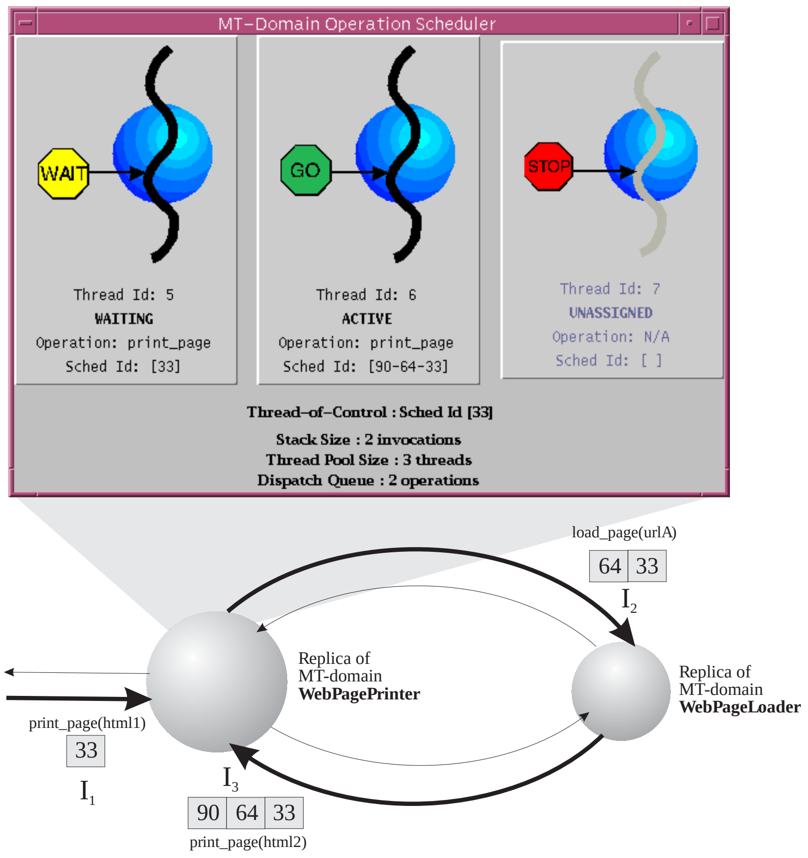


Figure 6.7: Snapshot of a MT-domain operation scheduler for a specific example.

6.4.1 Example

Figure 6.7 shows the actions of the operation scheduler for a replicated multithreaded application that is running over the Eternal system. The simple application was implemented using ORBacus (from Object-Oriented Concepts) [55], a commercial implementation of CORBA that provides for a variety of ORB concurrency models.

The application in this case consists of two MT-domains. One of these is a `WebPagePrinter`, with a method `print_page()`, which takes an HTML document as a parameter. The other MT-domain is a `WebPageLoader`, with a method `load_page()`, which takes the URL of a web page as a parameter. Given an HTML document, the `print_page()` method can understand and parse HTML sytax and print the text of the document.

An HTML document may have embedded URLs that refer to other Web pages. The

`WebPagePrinter` is intended to print all of the Web pages reachable from the HTML document supplied as a parameter. To do this, the `print_page()` method first scans the HTML document supplied as a parameter for any embedded URLs. An operation on the `WebPagePrinter` completes only when all of the pages reachable from the original HTML document, as well as the original HTML document itself, have been printed. The method `print_page()` is essentially a recursive operation because it must trace all of the embedded URLs in a given HTML document, and print each of the HTML documents at these URLs. Because the `print_page()` method takes a HTML document as a parameter, rather than a URL, it requires the assistance of the `WebPageLoader` to load a URL before it can print it.

In this example, the original HTML document `html1` contains a single embedded URL, `urlA`, which points to a HTML document (`urlA` does not itself contain any embedded URLs). The `print_page()` method is invoked (invocation I_1) on the `WebPage-Printer` and executes. The `WebPagePrinter` scans the document `html1`, and encounters `urlA`. This causes the `WebPagePrinter` to invoke (invocation I_2) the `load_page()` method of the `WebPageLoader`, passing it the embedded URL, `urlA`, as a parameter. The `load_page()` method retrieves the HTML document `html2` at `urlA`, and invokes (invocation I_3) the `print_page()` method of the `WebPagePrinter` to print `html2`. If `html2` contained an embedded URL, there would be yet another level of recursion.

Figure 6.7 shows a replica of the `WebPagePrinter` and a replica of the `WebPageLoader`. A snapshot of the operation scheduler for the `WebPagePrinter` is also shown as it is running. The operation scheduler for the replica of the `WebPagePrinter` assigns I_1 to thread id 5 within the replica with the scheduling identifier 33. The thread-of-control is assumed by operation I_1. The invocation I_2 is subsequently issued by the `WebPagePrinter` as a result of I_1, and must complete in order for the thread-of-control to be released. Thread id 5 suspends itself, awaiting a response to invocation I_2.

The invocation I_2 is dispatched within the replica of the `WebPageLoader`, with the scheduling identifier 64-33. Invocation I_3, issued by the `WebPageLoader` is a remote callback invocation on the MT-domain `WebPagePrinter`, and must execute in order for I_2, and thus I_1, to complete. The operation scheduler recognizes that I_3 is a descendant through the presence of the scheduling identifier (33) of invocation I_1 in the scheduling identifier (90-64-33) of invocation I_3.

The operation scheduler dispatches I_3 onto thread id 6. Meanwhile, other operations (unrelated to the thread-of-control) might arrive at the `WebPagePrinter`. The operation scheduler enqueues such operations until the thread-of-control is released. In this case, the dispatch queue contains two such operations that are waiting to be serviced once I_1 completes. Although the operation scheduler has an unassigned thread (with thread id 7) that is capable of executing one of the operations in the dispatch queue, the scheduler at every one of the replicas of the `WebPagePrinter` will not dispatch this operation until the thread-of-control is released.

Thus, the operation scheduler enforces a single logical thread-of-control, held in this case by operation I_1. The thread-of-control may be realized through multiple threads (thread ids 5 and 6), although at most one of these threads is actively executing at any point in time, while the others are suspended, awaiting a response. This deterministic scheduling and dispatching of operations provided by the Eternal system maintains replica consistency for nondeterministic multithreaded applications that are replicated.

6.5 Handling ORB Concurrency Models

Several different concurrency models [55, 61] may be used by multithreaded ORBs to dispatch requests to client and server objects within the application. Each of these models is suited to particular applications; not every ORB implements all of the possible strategies.

The operation scheduler is transparent to the multithreading policies of the ORB; thus, irrespective of the threading model of the ORB, the mechanisms of the operation scheduler can ensure the consistency of the replicas of a concurrency domain. The scheduling algorithm, as well as the enforcement of the thread-of-control, are also independent of the specific concurrency model (thread-per-request, thread-per-object, etc) adopted by the ORB that hosts the MT-domain.

6.5.1 Thread-per-Request Model

In this model, the ORB attempts to spawn a new thread for each new request on a MT-domain, even if this request is issued by the same client. While this enables remote callback operations to proceed on MT-domains, the potential for replica inconsistency arises if two threads executing the same, or different, requests within the same object update shared data. The Interceptor ensures that all of the ORB's attempts to spawn new threads are handled through a thread library interposer.

The interposed thread creation routines ensure that the ORB-spawned threads are collected into the dispatcher's thread pool Q_{thr} from which threads are extracted for execution by the CD's operation dispatcher. One of the disadvantages of the thread-per-request model is the cost of thread creation with each request. By enforcing a limit on the number of allowable threads within the dispatcher's thread pool Q_{thr}, and by using an interposed thread creation routine that respects this limit, the scheduler can reduce the overhead of unnecessary thread creation.

6.5.2 Thread-per-Connection Model

In this model, the ORB spawns a thread to handle multiple requests between a CORBA client object and server object. The potential for replica inconsistency arises if more than one client contacts a single server object. This results in multiple threads executing within the same object, with the potential for updates on shared data. From the point of view of replication, the thread-per-request and the thread-per-connection models have the same potential for replica inconsistency. The mechanisms of the scheduler that ensure replica consistency for the thread-per-request model are therefore equally applicable to both. The only difference is that the thread-per-connection model results in fewer calls to the interposed thread creation routine than the thread-per-request model.

6.5.3 Thread-per-Object Model

In this model, the ORB spawns a single thread to handle all requests for a single CORBA object. This can ensure replica consistency provided that the CORBA application consists of executing processes that contain only a single object. Typical applications have multiple

CORBA objects wihin a single process, by design, so that these objects can share data, for instance. In this case, the concurrency domain model imposed on the replicas of multi-object processes ensures replica consistency.

6.5.4 Thread Pool Model

This model is similar to the thread-per-request model, but with the ORB having spawned the threads ahead of time, to eliminate the overhead of thread creation at the time that the request is received. This model has the same pitfalls with regard to replica consistency as the thread-per-request model, and the mechanisms of the operation scheduler handle this model in the same manner. However, in this model, the operation dispatcher employs a thread pool to reduce the thread creation overhead while executing the scheduling algorithm.

Chapter 7

Gateways

Applications are increasingly spanning enterprises across the Internet, with the application objects within one enterprise communicating with, and performing operations, on the application objects of another enterprise. In this case, the reliability of the application as a whole depends on the reliability of the objects in each of the communicating enterprises, which are separated possibly by a considerable distance, as shown in Fig 7.1. Each enterprise is likely to be, and indeed should be, responsible only for the reliability of the objects under its control, but each enterprise must nevertheless allow the objects of a different enterprise to communicate with its own objects without compromising the consistency of the replicated objects of either enterprise. The domain of control of the fault tolerance infrastructure of each enterprise constitutes a *fault tolerance domain*; different fault tolerance domains can be connected through a *gateway*.

The concepts of fault tolerance domains and gateways are not restricted to communication between enterprises. Internet-based applications such as stock trading involve customers using Web browsers (typically unreplicated thin clients) to communicate with the servers (typically replicated for fault tolerance) of a stock trading company. In this case, the unreplicated Web browser should not have to be aware of the replication of the stock trading servers, but can nevertheless benefit from the fault tolerance of the servers. The unreplicated clients (the Web browsers) can be made to communicate with the replicated servers (the stock trading servers) through a gateway that hides the replication of the servers. The replicated server objects are managed by the fault tolerance infrastructure of the stock trading company, and the gateway serves as the "entry point" into the fault tolerance domain. The gateway is a crucial element because it must "understand" the reliability mechanisms inside the fault tolerance domain, as well as the unreliable semantics of the external client, and must bridge between these different semantics and mechanisms, without compromising the reliability of the objects within the fault tolerance domain.

A different motivation for a fault tolerance domain is that an application can have a very large number of objects that require replication, and it might not be a scalable or feasible solution to have a single fault tolerance infrastructure manage the replication of all

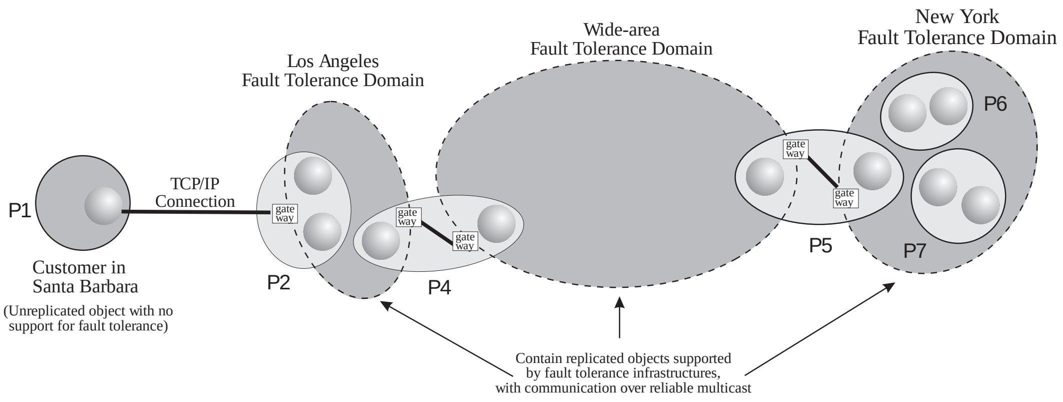

Los Angeles
Fault Tolerance Domain
Wide-area
Fault Tolerance Domain
New York
Fault Tolerance Domain
P1
TCP/IP
Connection
gate
way
P2
P4
P5
P6
P7
Customer in
Santa Barbara
(Unreplicated object with no
support for fault tolerance)
Contain replicated objects supported
by fault tolerance infrastructures,
with communication over reliable multicast

Figure 7.1: Gateways bridge fault tolerance domains, and allow objects in one fault tolerance domain to communicate with those in another. Here, P_i represents a processor hosting some application objects.

of these objects. Instead, it would be preferable to decompose the application into smaller collections of objects, with each collection of objects being managed by a distinct fault tolerance infrastructure, and therefore constituting a fault tolerance domain. Regardless of the motivation for a fault tolerance domain, the gateway mechanism is identical and essential.

The Eternal system constitutes the fault tolerance infrastructure (within the fault tolerance domain). While the fault tolerance infrastructure ensures strong replica consistency within the fault tolerance domain, it is the responsibility of the gateway to ensure that unreplicated clients wishing to contact replicated objects within the fault tolerance domain (through the gateway) do not compromise the replica consistency of those replicated objects.

7.1 Fault Tolerance Domains

Eternal must allow the CORBA applications that it supports to communicate with unreplicated objects that are necessarily outside the fault tolerance domain, *i.e.*, Eternal's domain of control. Some of these unreplicated objects (*e.g.*, a Web browser on a personal computer that provides no fault tolerance) might not be supported by, or have access to, Eternal's fault tolerance infrastructure, and might run over standard IIOP-enabled ORBs.

Eternal ensures that these unreplicated objects outside the fault tolerance domain can nevertheless communicate with the replicated objects that are under Eternal's control inside the fault tolerance domain. Furthermore, Eternal makes this communication possible without the unreplicated object ever being aware of the existence of a fault tolerance domain, of the replication of the objects within the fault tolerance domain, or of Eternal itself. Thus, Eternal extends the replication transparency that it provides to the application objects within the fault tolerance domain equally to unreplicated objects outside Eternal's control.

The gateways that the Eternal system provides serve as the “entry point” for unreplicated clients into the fault tolerance domain, and allow unreplicated external objects to invoke replicated Eternal-managed objects.

Within a fault tolerance domain:

- All objects are replicated, with the replication managed by Eternal’s fault tolerance infrastructure.
- Communication between replicated objects occurs through a reliable totally ordered multicast protocol, thereby facilitating replica consistency, as described in Section 3.1.
- Replicated clients do not use the TCP/IP {host, port} information within the Interoperable Object Reference (IOR) of any of the server replicas to contact the replicated server. Instead, the Eternal Interceptor transparently diverts the socket establishment routines at every client replica to form a connection to the local Eternal Replication Mechanisms, which then multicast the notification of the connection establishment to the Replication Mechanisms hosting the server replicas.

Outside a fault tolerance domain:

- Objects are unreplicated, and are unaware of the internal mechanisms of, and the replication within, the fault tolerance domain.
- Communication occurs through CORBA’s TCP/IP-based Internet Inter-ORB Protocol (IIOP).
- Clients use the TCP/IP {host, port} information within the Interoperable Object Reference (IOR) of the target server to establish a connection with the server.

Unreplicated objects outside the fault tolerance domain must never be allowed to access the replicated objects within the fault tolerance domain directly. Such direct communication, if permitted, can violate replica consistency. The reason is that the unreplicated client can communicate only through TCP/IP, thereby implying that it would contact only *one* of the server replicas, and invoke an operation on that replica alone.

If the server is actively replicated, and only the single invoked server replica performs the operation, it can have a different state from the other replicas of the same server object, resulting in inconsistent replication. If the server is passively replicated, and the single primary replica is invoked, the primary replica can itself invoke other nested operations as a result of the original invocation. If the primary fails before it receives the results of the nested invocations, a new primary server replica will be elected. However, because the new primary (formerly a backup replica) did not receive the original invocation, it will not be to handle the returned responses from the nested invocations, and will also not be able to return a response to the original invocation. Thus, to ensure replica consistency, the replicas of an object must be contacted only through a reliable totally ordered multicast, and not individually through TCP/IP.

To ensure this, additional mechanisms are provided by Eternal so that an IOR published by a replicated object within the fault tolerance domain “point” the external clients in the

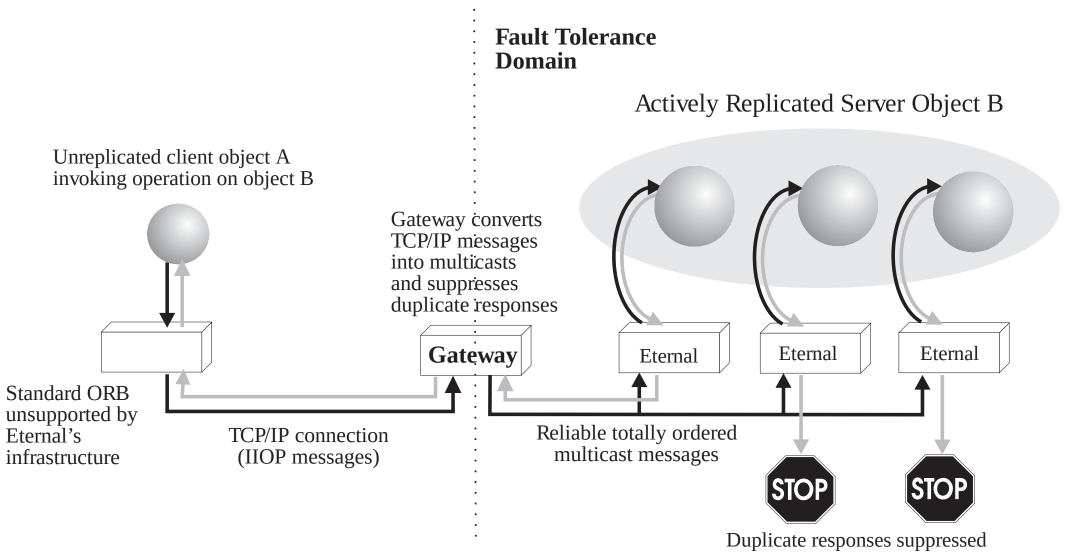

Fault Tolerance Domain
Actively Replicated Server Object B
Unreplicated client object A invoking operation on object B
Gateway converts TCP/IP messages into multicasts and suppresses duplicate responses
Gateway
Eternal
Eternal
Eternal
Standard ORB unsupported by Eternal's infrastructure
TCP/IP connection (IIOP messages)
Reliable totally ordered multicast messages
STOP
STOP
Duplicate responses suppressed

Figure 7.2: Eternal's gateways allow unreplicated clients to communicate with replicated servers.

direction of the IIOP-enabled gateway, rather than the target replicated object itself. However, the external client that uses this IOR is unaware of this. When using the information in the IOR for connection establishment, the client implicitly assumes that the endpoint is the real server and, thus, sends IIOP invocations (destined for the server) to the gateway.

Note that the gateway is not a CORBA object, but constitutes part of the mechanisms provided by the fault tolerance infrastructure of Eternal. However, by receiving the unreplicated client's IIOP invocations without returning exceptions, and by forwarding the replicated server's IIOP responses to the unreplicated clients, the gateway appears to the client to be a remote CORBA server object.

To perform the invocation (response) forwarding into (out of) the fault tolerance domain, the gateway must be able to interpret the IIOP messages sent over TCP/IP connections from outside the fault tolerance domain, as well as the reliable totally ordered multicast protocol messages within the fault tolerance domain, and must provide the necessary translation between them. This functionality of the gateway is shown in Figure 7.2.

Another aspect of the gateway is that it must "hide" the replication of the servers from the external client. This involves detecting duplicate responses returned by the replicas of the server, and filtering out only a single distinct response to the external client. In addition, the gateway must itself be reliable so that it does not constitute a single point of failure.

7.2 Connection Establishment

In a typical CORBA application, a server object publishes its location {server_host, server_port} through an Interoperable Object Reference (IOR). A client object wishing to contact the server extracts the TCP/IP addressing information from the IOR, and establishes a connection to the host and port specified in the IOR. Once the connection is established, the client and the server can communicate over the connection using IIOP messages.

A client implicitly "believes" that a server's IOR contains the {server_host, server_port} information, unless informed otherwise through a LOCATION_FORWARD (where the endpoint contacted by the client provides a forwarding address to the actual server location) by the remote endpoint, on receipt of the first IIOP message from the client.

When a gateway is used, every unreplicated external client must continue to "believe" that the remote endpoint to which it connects (using the information in the server IOR) is the server, when, in fact, the remote endpoint is the gateway. This can be done by ensuring that the addressing information in the IOR is the {gateway_host, gateway_port}, and also by ensuring that the gateway always returns the expected IIOP responses to the client's IIOP invocations so that the client never suspects otherwise.

Eternal accomplishes the replacement of the {server_host, server_port} in the IOR of each server replica with the {gateway_host, gateway_port} through the use of its Interceptor. There are several choices for the library symbols that the Interceptor could interpose to perform the {host, port} replacement in the IOR.

The first and most obvious choice would have been to intercept the *write()* call when the ORB attempted to write the IOR to a file or to a Naming Service. At this point, we could have parsed the original IOR and performed the necessary {host, port} replacement. This is not a good strategy because all other invocations of the *write()* call would also have been caught, though not modified, by the Interceptor. Because *write()* is the popular choice for transmitting messages over the network, writing to files, writing to the screen, etc., interposing on the *write()* call leads to an unnecessarily large number of undesired interceptions, and consequently, a higher overhead in the path of message transmission. Of course, the number of undesirable interceptions can be reduced through the conditional interception of the *write()* call, with the default definition of the *write()* call being used subsequent to the IOR replacement.

A better choice is to interpose at the point that the server-side ORB queries the operating system for the host and the port information, *prior* to publishing the IOR. By interposing on the *getsockname()* call and/or the *sysinfo()* call (with the SI_HOSTNAME command) to return the gateway_host and the gateway_port instead of the server_host and the server_port, respectively, the IOR that the server-side ORB publishes automatically contains the {gateway_host, gateway_port}. This eliminates the effort of having to parse the IOR string to do the replacement, and also results in far fewer undesirable interceptions. The gateway_host and the gateway_port are dedicated choices that are supplied to the interceptor at system configuration time.

When an unreplicated client uses this IOR, the client-side ORB, implicitly assuming that the host and port in the IOR refer to the server object, connects the client to the gateway. The gateway now becomes the recipient of every IIOP message sent by the unreplicated client, which continues to "believe" that the gateway is indeed the target server object. By

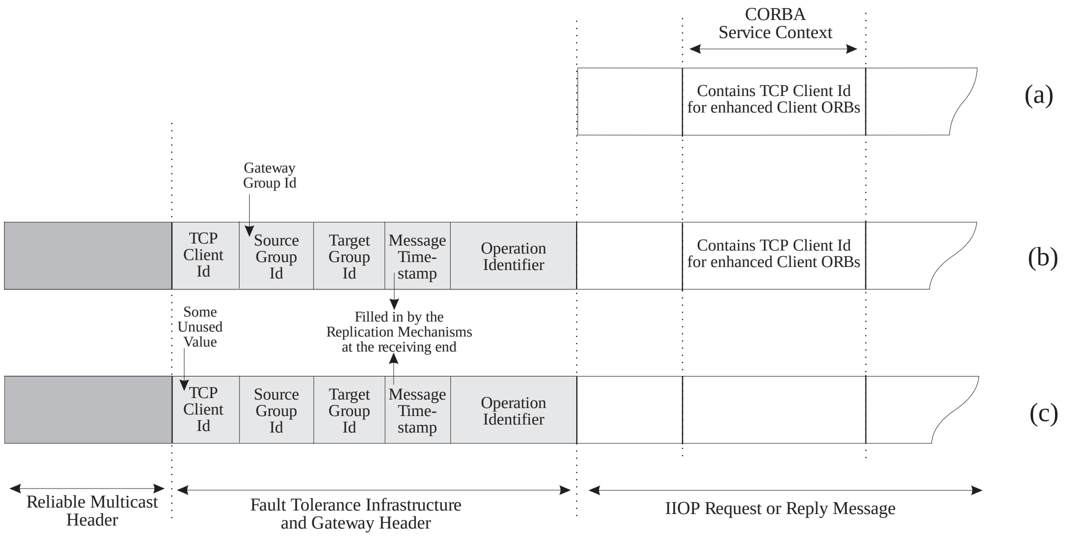

CORBA
Service Context
Contains TCP Client Id for enhanced Client ORBs
(a)
Gateway Group Id
TCP Client Id
Source Group Id
Target Group Id
Message Time-stamp
Operation Identifier
Contains TCP Client Id for enhanced Client ORBs
(b)
Some Unused Value
Filled in by the Replication Mechanisms at the receiving end
TCP Client Id
Source Group Id
Target Group Id
Message Time-stamp
Operation Identifier
(c)
Reliable Multicast Header
Fault Tolerance Infrastructure and Gateway Header
IIOP Request or Reply Message

Figure 7.3: Messages sent (a) between an unreplicated client and the gateway, (b) from the gateway to a replicated object within a fault tolerance domain, and (c) between replicated objects within a fault tolerance domain.

extracting the server's object key (which the client-side ORB inserts into IIOP invocations to identify the target server), the gateway identifies the target server, multicasts the client invocation to the server object group. The gateway inserts sufficient information into the multicast messages to enable it to associate the server's response with the client's invocation.

The gateway process must be continuously listening on the dedicated {gateway_host, gateway_port} for connections from unreplicated clients. For each new client that contacts the gateway, the gateway spawns a new TCP/IP socket to communicate solely with that client, and uses the original socket to listen for further clients. The additional spawned sockets are destroyed when the connection between the unreplicated client and the gateway terminates.

Note that the replacement of the {server_host, server_port} in the IOR does not affect connection establishment or communication within the fault tolerance domain. Replicated clients wishing to communicate with replicated servers within the fault tolerance domain never use this TCP/IP-specific addressing information, but use instead the server's object group identifier to contact the replicated server through the fault tolerance infrastructure.

7.3 Encapsulation of IIOP into Multicast Messages

A gateway must encapsulate the IIOP invocations from the external unreplicated clients into multicast messages for transmission to the target replicated server object within the fault tolerance domain. Similarly, the corresponding IIOP responses, encapsulated within the multicast messages returned by the replicated server object, must be extracted by the gateway and returned to the unreplicated clients.

```
for (every received IIOP message)
{
  Obtain TCP client identifier
  Map socket identifier to client identifier
  Generate and record operation identifier
  Generate message header containing:
    – TCP client identifier
    – Gateway group identifier (sender)
    – Server group identifier (receiver)
    – Operation identifier
  Encapsulate header and IIOP message
    into multicast message
  Send multicast message into the
    fault tolerance domain
}
```

Messages from unreplicated clients

```
for (every received multicast message)
{
  Extract operation identifier
  Examine if message is a duplicate
  if (non-duplicate message)
  {
    Extract TCP client identifier
    Find corresponding socket identifier
    Extract IIOP message
    Send IIOP message over the socket
      identifier to the TCP client
  }
  else
    Discard duplicate message
}
```

Messages from replicated objects

Figure 7.4: Actions of the gateway for incoming messages from (a) external unreplicated clients outside a fault tolerance domain, and (b) replicated objects within a fault tolerance domain.

When an IIOP-encapsulating message is multicast by the gateway into the fault tolerance domain, the message contains the gateway_group_id as the sender group, and the server_group_id (determined by the gateway from the server's object key embedded in the client's IIOP invocation) as the destination group. The message is received in total order by the Replication Mechanisms hosting each of the server replicas. The replicated server performs the operation, and the fault tolerance infrastructure multicasts the results to the gateway. The replicated server assumes that the gateway that sent the IIOP invocation is a CORBA client object. Eternal's transparency through interception effectively ensures that neither the unreplicated client, nor any of the server replicas, is ever aware of communicating through the fault tolerance infrastructure using reliable multicast. The gateway (and, of course, the fault tolerance infrastructure itself) is the only party in the chain of communication that is aware of the reliable multicast and the fault tolerance infrastructure.

When the replicated server returns the response to the gateway, the IIOP response from each server replica is encapsulated by the Replication Mechanisms hosting that replica into a multicast message. The message contains the server_group_id as the sender group, and the gateway_group_id as the destination group. This information is insufficient for the gateway to route the IIOP response to the client replica that invoked the operation because multiple unreplicated TCP/IP-based clients might have invoked the same replicated server through the gateway. The gateway has no way of discriminating between these clients.

Thus, every multicast message must contain additional information, inserted by the gateway to identify each TCP/IP client that contacts the gateway. The resulting multicast messages have the structure shown in Figure 7.3. For multicast messages exchanged between replicated objects within the fault tolerance domain, the TCP/IP client identification is set to some unused value. The gateway (as well as the fault tolerance infrastructure) uses the destination group identifier, the source group identifier and the TCP/IP client identifier *collectively* to route every message to its intended destination.

Ideally, the client identification information ought to be supplied by the client-side ORB, as discussed in Section 7.4.2. Because this is not the case with current ORBs, the gateway can maintain a simple counter, one for each destination server group. For each incoming TCP/IP client, the gateway first determines the server_group_id from the first IIOP message received from the client. The gateway then uses the value of the counter corresponding to that server group as the TCP/IP client identifier. The counter is then incremented, to serve as the identifier for the next TCP/IP client for the same replicated server. The disadvantage of the gateway-assigned client identifiers, over identifiers supplied by the client-side ORB, is discussed in Section 7.4.1.

Figure 7.4 shows the sequence of steps that the gateway executes for incoming IIOP messages from outside the fault tolerance domain, and incoming multicast messages from within the fault tolerance domain.

To ensure replica consistency, duplicate detection and suppression mechanisms are used by Eternal's fault tolerance infrastructure throughout the fault tolerance domain; the gateways also employ these mechanisms for filtering duplicate responses from the replicated server objects within the fault tolerance domain. The gateway returns only a distinct copy of each response to the invoking external client. The duplicate copies of each response, if not suppressed, would be delivered to the client object, and could cause the client object's state to be corrupted.

To detect duplicate copies of each response, both the fault tolerance infrastructure and the gateway prepend the operation identifiers of Section 3.4 to each message that is multicast within the fault tolerance domain, as shown in Figure 7.3.

7.4 ORB-Related Issues

7.4.1 Using Existing ORBs

Existing ORBs do not have the capability to traverse a list of profiles, and select the next profile if the first one fails on connection. The disadvantage of this is that redundant gateways are not possible. Clients might experience disconnection if the processor hosting the gateway fails, and does not recover. The processor hosting the gateway is a single point of failure. If the client ORB has the capability to understand only the first IIOP profile (the standard TAG_INTERNET_IOP profile), and if the gateway to which it connects using the first profile fails, the client has no alternative but to abandon the request. Furthermore, the client does not know the status of any invocations that it has already sent, for which it is still awaiting responses.

An alternative to using multiple gateways is to have a cold passively replicated gateway. In this case, the gateway's state should be checkpointed often enough to allow it to be recovered. However, clients will still be disconnected from the gateway if it fails, and must have mechanisms to allow them to reconnect to the gateway, when it recovers.

In the case of redundant gateways, the new gateway to which the client connects (on failure of the first gateway) has no way of "knowing" that this is the same client. The simple counter mechanism, described in Section 7.3, is insufficient in this case to identify the client. This means that, even if the new gateway receives the response for an outstanding invocation sent by the client through the first gateway, the new gateway does not know which of its connected clients should receive this response. Secondly, if the client were now to re-issue all of the pending invocations to the new gateway, the new gateway could, in turn, re-issue these invocations to the replicated objects within the fault tolerance domain, thereby corrupting their state.

Thus, due to lack of client-side identification provided by the ORB, the gateway cannot prevent duplication of client requests if

- The unreplicated client fails, recovers and resends its request (this is outside the fault tolerance domain's and the gateway's control, and cannot be handled without extending some of the fault tolerance mechanisms to the unreplicated client)
- The gateway process fails, and then recovers, and the client reconnects to the gateway
- Redundant gateways are used, and the original gateway fails, and the client switches to the next operational gateway

7.4.2 Enhancements to Existing ORBs

If only a single gateway is provided for a fault tolerance domain, it is insufficient to guarantee the level of reliability that customers of Internet-based applications have come to expect. For instance, if a customer uses an unreplicated Web browser to connect to a replicated stock trading server through a gateway, the failure of the gateway could leave the customer wondering about the status of any outstanding invocations issued on the stock trading server. Because the gateway constitutes a single point of failure, the benefits of the server replication are lost to the customer.

The use of redundant gateways requires additional intelligence on the part of the client-side ORB to exploit the multiple gateways. Unfortunately, the required mechanisms are not part of the current CORBA standard. In the absence of the required support in current ORBs, we have implemented a thin client-side interception layer that mimics the support that an enhanced client-side ORB would provide to allow unreplicated CORBA clients to benefit from fault tolerance. Ultimately, though, as discussed in Section 7.4.2, we envisage that the functionality of this interception layer should be incorporated into the client-side ORB itself.

According to the current CORBA standard, a profile contains addressing information within an IOR. An object's IOR can contain multiple profiles, with each profile designating an alternative address for contacting the object. To allow the addressing information for the multiple gateways to be made available to unreplicated clients, the Eternal Interceptor "stitches" together the addressing information for each gateway into a single multi-profile IOR.

On the client side, the thin interception layer is endowed with the capability of traversing the profiles within the multi-profile IOR, should this be required. The interception layer connects the client object to the first gateway listed in the multi-profile IOR, and inserts a unique TCP/IP client identifier into the service context[1] of each IIOP message sent out by the client. The advantage of using the service context field is that it can be safely ignored (as is the case here) by a server ORB that does not understand it. It is intended purely for the consumption of the gateway.

For each IIOP request message that a gateway receives from a client, the gateway first multicasts the message to the group of gateways. This is done so that every gateway in the group has a record of the invocation in case the first connected gateway fails. The gateway group then multicasts the message into the fault tolerance domain, and the gateway group (and not the connected gateway alone) receives the response.

If the first gateway fails to respond, the client-side interception layer transparently skips to the next profile in the multi-profile IOR, and connects the client to the next operational gateway, and reissues any pending invocations. If the client object sent an invocation for which a response was expected from the first gateway, the client-side interception layer obtains it from the next operational gateway. This is possible because the client-side interception layer supplies the same unique client identifier for each of its requests, along with a unique request identifier, which would make it possible for the new gateway to detect reinvocations due to reconnection of the client-side interception layer to a different gateway. The reason for the reinvocations is two-fold: firstly, it allows the client-side interception layer to communicate the client's unique identifier to the gateway, and secondly, the client-side interception layer has no way of knowing if the first invocation ever reached the original failed gateway. Each gateway also contains the intelligence to inform all of the other gateways in the event that the client fails. In this case, the gateways can delete any state that they have stored on behalf of the client.

The duplicate detection and suppression mechanisms described in Section 3.4, along with the unique client identifier, and CORBA's existing request identifier mechanisms, enable the gateway to preserve the replica consistency within the fault tolerance domain, as well as to protect the unreplicated client outside the fault tolerance domain from having its state corrupted. Furthermore, the redundant gateways scheme enables the unreplicated client to benefit from the fault tolerance of the server.

[1]The service context is a part of the IIOP request and reply messages, where the user may insert information. If a receiving ORB cannot interpret this information, it will ignore it.

Chapter 8

Implementation and Performance

The current implementation of the Eternal system is capable of providing fault tolerance to unmodified CORBA applications using the following unmodified commercial ORBs:

Solaris 2.x on UltraSPARC workstations

- VisiBroker [22] from Inprise Corporation
- Orbix [23] from Iona Technologies
- CORBAplus [14] from Expersoft (now Vertel)
- ORBacus [55] from Object-Oriented Concepts Inc.
- TAO [62] from Washington University, St. Louis
- omniORB2 [32] from AT & T Laboratories, U.K.
- ILU [26] from Xerox PARC

RedHatLinux 6.0 on Intel PCs

- VisiBroker [22] from Inprise Corporation
- ORBacus [55] from Object-Oriented Concepts Inc.
- omniORB2 [32] from AT & T Laboratories, U.K.

The current implementation of Eternal exploits library interpositioning, which is less dependent on operating system specific mechanisms and has lower overheads than our initial implementation, which was based on intercepting the */proc* interface of the Solaris operating system. Either approach (library interpositioning or using */proc*) allows the mechanisms of Eternal to be used with diverse commercial ORBs, with no modification of either the ORB or the application. The only stipulation is that the vendor's implementation of CORBA must support IIOP, as mandated by the CORBA standard.

The mechanisms that the Eternal system employs to ensure replica consistency are implemented beneath the ORB and, thus, are made transparent to the application and to the ORB through the use of interception.

8.1 Challenges

Although all of the commercial CORBA implementations conform to the CORBA standard, each ORB employs certain vendor-specific mechanisms which detract, in some measure, from its interoperability with other ORBs. This is due to the different interpretations of the CORBA standard on the part of the ORB vendors, who are responsible for translating the CORBA standard into an ORB implementation.

Moreover, not all of the commercial ORBs can "talk" to each other successfully under all circumstances. For instance, some of the ORBs tend to "pack" multiple GIOP messages into a single entity for efficient transmission, while other ORBs expect to receive GIOP messages that are not necessarily in this packed form. Because Eternal deals with the IIOP interface of each ORB, it is possible to transcend the intrinsic differences between ORBs by having the Eternal Replication Mechanisms perform some additional, and appropriate, conversion between the IIOP formats of different ORBs.

Furthermore, ORBs that are optimized for specific purposes tend to use non-standard or different system calls as part of the IIOP interface. For certain ORBs, Eternal intercepts these additional system calls, and maps them transparently to Totem. Nevertheless, for every ORB that uses system calls that Eternal does not handle as yet, the Interceptor and the Replication Mechanisms need to be modified to extend their capabilities to these new system calls.

One of our aims is to be able to operate Eternal in an environment of networked heterogeneous ORBs. However, we faced a number of challenges in building the Interceptor to ensure that the approach worked successfully with various commercial implementations of CORBA. The issues that had to be resolved dealt with vendor-specific features of the different ORBs. Because the Interceptor interfaces to the ORB at one end and to the operating system at the other, it must be equipped to handle the use of possibly non-standard mechanisms at either interface.

8.1.1 Transcending ORB-Specific Mechanisms

Different implementations of CORBA over the same operating system use different system calls to communicate messages over the TCP/IP-based Internet Inter-ORB Protocol (IIOP). In particular, the low-level implementation of input and output operations of CORBA clients or servers can be optimized by the choice of different I/O-related system calls of the Unix System V STREAMS interface. The `read` and `write` system calls, typically used for I/O operations, are not necessarily the most efficient because the buffer arguments of these system calls are limited in size. On the other hand, STREAMS-based system calls such as `getmsg` and `putmsg` allow for multiple messages to be conveyed to the kernel in a single buffer argument. They also allow for control information to be passed along with the data. This is particularly important for connection-related system calls, because the embedded control information typically represents a transport primitive that identifies the establishment or the release of a connection, or other relevant connection-specific information.

In addition, ORBs such as the COOL ORB [25], tend to convey a considerable amount of information through `ioctl` system calls, which are highly efficient for I/O. The difficulty with interpreting the `ioctl` system calls, as required of the Interceptor, is that the system call is generic, but has many different formats, each depending on the context of the system call, and the `ioctl` command used. While some of the `ioctl` commands (and their associated data structures) are well documented, a large proportion of `ioctl`-related information is embedded in the source code of the operating system, to which the system developer does not necessarily have, or desire, access. This is particularly true of the `ioctl`s related to the STREAMS module `sockmod`, used for TCP/IP communication. One of our biggest challenges, and accomplishments, to date has been the exhaustive interpretation of the formats of the often poorly documented messages used by the kernel, to the extent that the Interceptor "understands" all protocol-related communication, irrespective of the ORB being used, and of the specific operating system interface that the ORB vendor chooses to employ.

8.1.2 Proprietary ORB Protocols

Another concern with the various commercial ORBs is their use of proprietary communication protocols instead of the Internet Inter-ORB Protocol (IIOP) mandated by the CORBA standard. Vendors employ customized protocols for communication between CORBA objects to increase efficiency or to reduce the use of bandwidth for connection management.

For instance, Orbix 2.2 from Iona Technologies typically uses the proprietary Orbix protocol (instead of IIOP) for all communication between objects and a vendor-specific daemon, `orbixd`. The daemon is required to be running continuously on every machine that hosts Orbix-based CORBA servers. Its function is to assign ports to servers and to enable clients to discover servers at runtime. Although Iona has enabled IIOP as the default protocol for inter-object communication in their latest release (Orbix 2.3), communication with `orbixd` is not necessarily via a standard protocol.

Furthermore, for reasons of security, certain ORBs tend to send IIOP messages in an encrypted form. While these security mechanisms are desirable and required by the application, they effectively prevent the Interceptor from understanding all of the IIOP messages. We are currently developing mechanisms that will make it possible for the Interceptor to function with no knowledge of the content of IIOP messages, to provide fault tolerance while retaining the degree of security enforced by the application.

8.1.3 Connection Management in ORBs

A further source of problems is that commercial ORBs, such as Orbix, embed the vendor-specific daemon's, rather than the server's, host name and port number in the IOR published by the ORB for the server. Thus, clients contact the daemon first, with no knowledge of the server's location. The daemon then facilitates the connection establishment between the client and the server. The client retains its connection to the daemon on the server's machine in addition to its connection with the server itself.

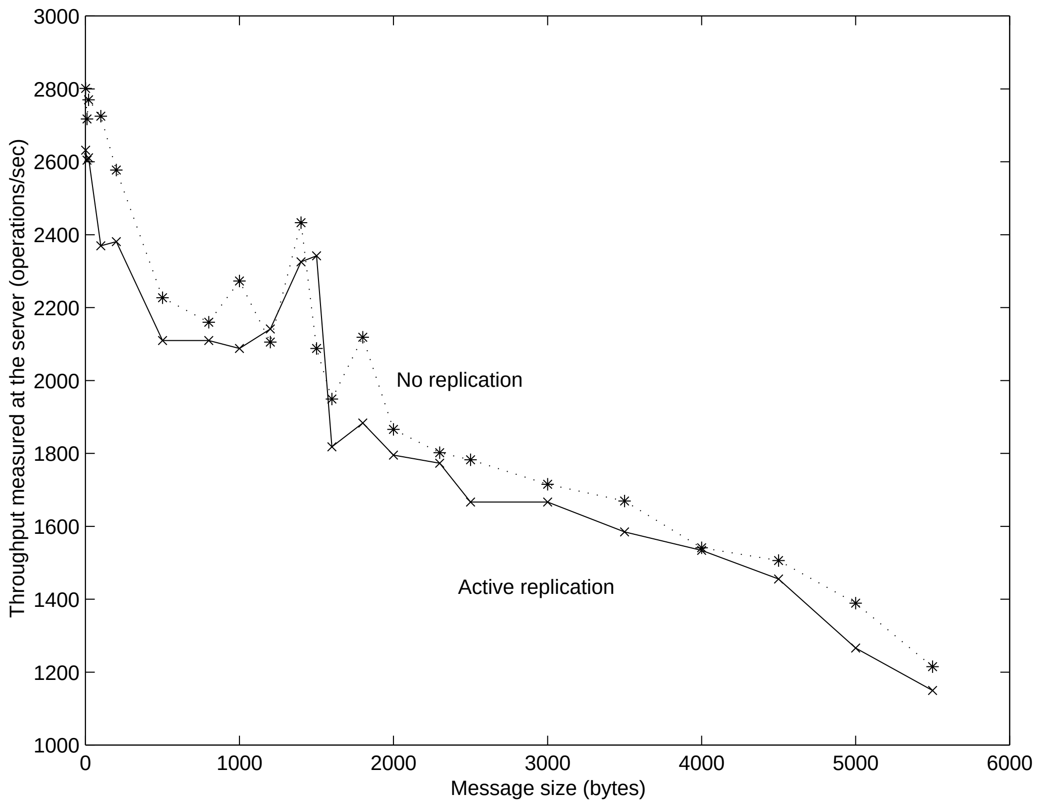


Figure 8.1: Throughput for varying message sizes measured for a test application running over VisiBroker for C++ on Solaris 2.x.

An increasing number of commercial ORBs are now equipped with daemons that enable communication between CORBA objects. While the intention of the vendor in implementing a daemon is to reduce the burden of connection management on the client and the server, the daemon has the disadvantage of being a single point of failure.

Also, for each client or server that opens a communicating TCP/IP connection, it is essential that the TCP/IP connection remain open even if the connection endpoint dies. For the interception approach, this is necessary because we map all communication over a single TCP/IP connection (with a target object) onto a multicast connection to each of the target object's replicas. Thus, the existence of the TCP/IP connection is required as long as at least one replica of the target object exists. However, the presence of the daemon hinders this because the daemon manages all client-server communication. Specifically, when a server replica dies, the daemon notifies the client replica at the endpoint of the death of the server replica, through the connection between the daemon and the client. This forces the client to tear down its TCP/IP connection, despite the existence of alternative server replicas. While this is the correct behavior for normal unreplicated applications, it is undesirable from the point of view of fault tolerance, as provided by Eternal.

In order for the Interceptor, rather than a vendor-specific daemon, to remain in control of connection management and server activation, it was necessary to develop mechanisms that supported interception of the daemon itself. Fortunately, in the case of Orbix, the daemon is also endowed with an IDL interface, `IT_daemon`, which supports methods to register and activate servers, to assign values to connection-specific parameters and to manage client-server communication. Because the daemon possesses an IDL interface, its behavior (while using its IDL interface) is similar to that of other CORBA objects, and it can be forced to use IIOP for communicating with server and client objects. Although such an ORB daemon can be treated as a CORBA object, it need not be replicated because it manages information and objects that are local to a machine.

The interception of the daemon is trickier than that of other CORBA objects. The reason is that the daemon encapsulates mostly vendor-specific information and may choose to communicate this in protocols that are not a part of the CORBA standard. To transcend the differences in connection management between different ORBs, and thereby to ensure true interoperability, we are exploiting the interception approach for developing inter-ORB adapter mechanisms. The intent of these mechanisms is to prevent application programmers from worrying about the interoperability issues that currently arise between ORBs from different vendors. By virtue of their functionality, the adapters will be the only part of Eternal that handle ORB-specific details. We also anticipate that their use will decrease as ORB vendors adhere more closely to common interfaces and to the CORBA standard.

8.2 Performance

The performance of Eternal has been measured for test applications running over different commercial ORBs, and using different CORBA invocation semantics (synchronous, deferred synchronous, asynchronous one-way), for different message sizes and for different types of method parameters.

For Solaris 2.x on 167Mhz SPARC workstations, when application objects are replicated, and Eternal's Mechanisms are used to maintain replica consistency, test applications typically incur about a 10% increase in remote invocation/response time, as compared with their unreplicated unreliable counterparts. For RedHatLinux 6.0 on 400Mhz Intel Pentium processors, this overhead reduces to 2% to 3%.

8.2.1 Throughput Measurements

The overheads due to Eternal are determined by measuring the throughput of a simple test application based on the VisiBroker 3.3 ORB for C++ running over the Solaris 2.5.1 operating system in a network of SPARC workstations connected by a 100Mbps Ethernet. The test application involves a client object acting as a packet driver, sending a continuous stream of invocations to the server object. Each invocation carries a fixed-length message to the server object, asking the server object to echo the message. The throughput is measured as the size of the message is varied from 1byte to 5500bytes.

In the unreplicated case, there is a single client object and a single server object, each running on a separate SPARC workstation. In the replicated case, there are three active

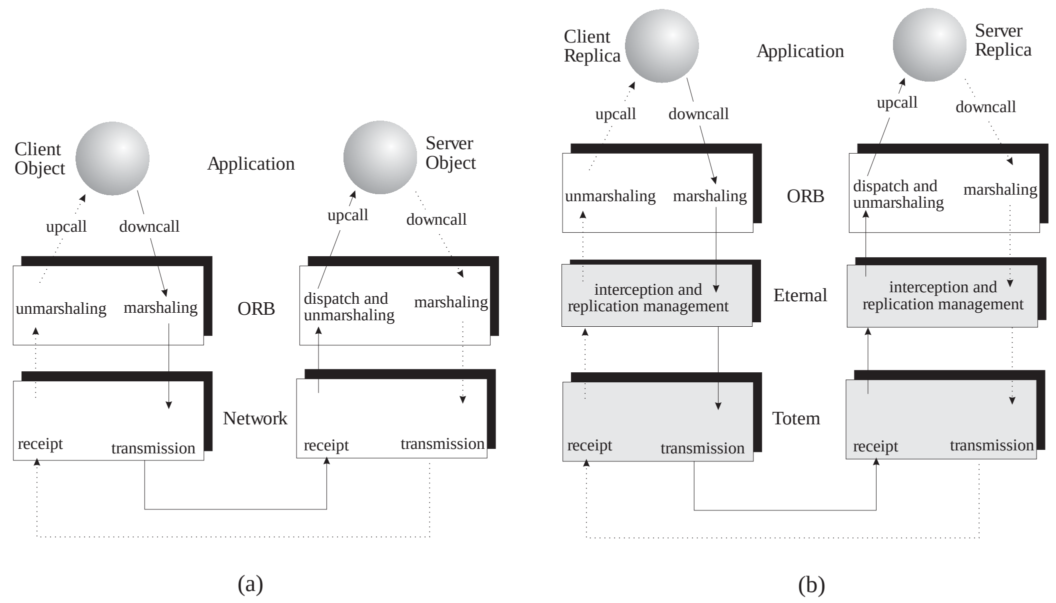


Figure 8.2: The round-trip time for an invocation is a measure of (a) the time taken by the ORB's marshaling/unmarshaling mechanisms and the communication infrastructure when the objects are unreplicated, and (b) the additional time due to Eternal's Mechanisms and the Totem protocols when the objects are replicated and managed by Eternal.

replicas of the client object and three active replicas of the server object, with each replica located on a separate SPARC workstation. The replica consistency is handled by Eternal's Mechanisms. The results for the unreplicated and replicated cases are shown in the graph of Figure 8.1.

As the graph shows, the overheads due to Eternal's interception, replication management and multicasting are reasonable, being in the range of 10% or less for most message sizes. As the message size increases, the throughput in the replicated case reduces gradually from 2630 messages/s for a 1-byte message to 1320 message/s for a 5500-byte message.

8.2.2 CORBA Benchmarks

The Distributed Systems Research Group of Charles University in Prague, Czech Republic, has developed a suite of benchmarks [9] with several criteria for evaluating a commercial ORB. The results of their benchmarks, as well as the code for their performance tests, are available for Orbix, VisiBroker, omniORB2 and ORBacus.

This suite of benchmarks includes performance tests for measuring the dependence of the round-trip invocation time on the type and the encapsulation of the parameters of the method being invoked. The benchmark suite involves a single client object and a single server object, with the client invoking every method of the server.

The server object implements the IDL interface `through` shown in Figure 8.3. This interface comprises methods that take arguments of different IDL types – primitive IDL data

```
typedef float float_arr[256];
typedef double double_arr[128];
typedef long long_arr[256];
typedef short short_arr[512];
typedef unsigned long ulong_arr[256];
typedef unsigned short ushort_arr[512];
typedef char char_arr[1024];
typedef octet octet_arr[1024];
typedef any any_arr_1024[1024];
typedef any any_arr_512[512];
typedef any any_arr_256[256];
typedef sequence<float,256> float_seq;
typedef sequence<double,128> double_seq;
typedef sequence<long,256> long_seq;
typedef sequence<short,512> short_seq;
typedef sequence<unsigned long,256> ulong_seq;
typedef sequence<unsigned short,512> ushort_seq;
typedef sequence<char,1024> char_seq;
typedef sequence<octet,1024> octet_seq;
typedef sequence<any,1024> any_seq_1024;
typedef sequence<any,512> any_seq_512;
typedef sequence<any,256> any_seq_256;
typedef string<1024> str;
interface through    {
        long infloat(in float x);
        long indouble(in double x);
        long inlong(in long x);
        long inulong(in unsigned long x);
        long inshort(in short x);
        long inushort(in unsigned short x);
        long inchar(in char x);
        long inoctet(in octet x);
        long inany(in any x);
        long floatArray(in float_arr x);
        long doubleArray(in double_arr x);
        long longArray(in long_arr x);
        long shortArray(in short_arr x);
        long ulongArray(in ulong_arr x);
        long ushortArray(in ushort_arr x);
        long charArray(in char_arr x);
        long octetArray(in octet_arr x);
        long anyArray1024(in any_arr_1024 x);
        long anyArray512(in any_arr_512 x);
        long anyArray256(in any_arr_256 x);
        long floatSeq(in float_seq x);
        long doubleSeq(in double_seq x);
        long longSeq(in long_seq x);
        long shortSeq(in short_seq x);
        long ulongSeq(in ulong_seq x);
        long ushortSeq(in ushort_seq x);
        long charSeq(in char_seq x);
        long octetSeq(in octet_seq x);
        long anySeq1024(in any_seq_1024 x);
        long anySeq512(in any_seq_512 x);
        long anySeq256(in any_seq_256 x);
        long StringString(in str x);
};
```

Figure 8.3: The IDL interface `through` used in the benchmarks.

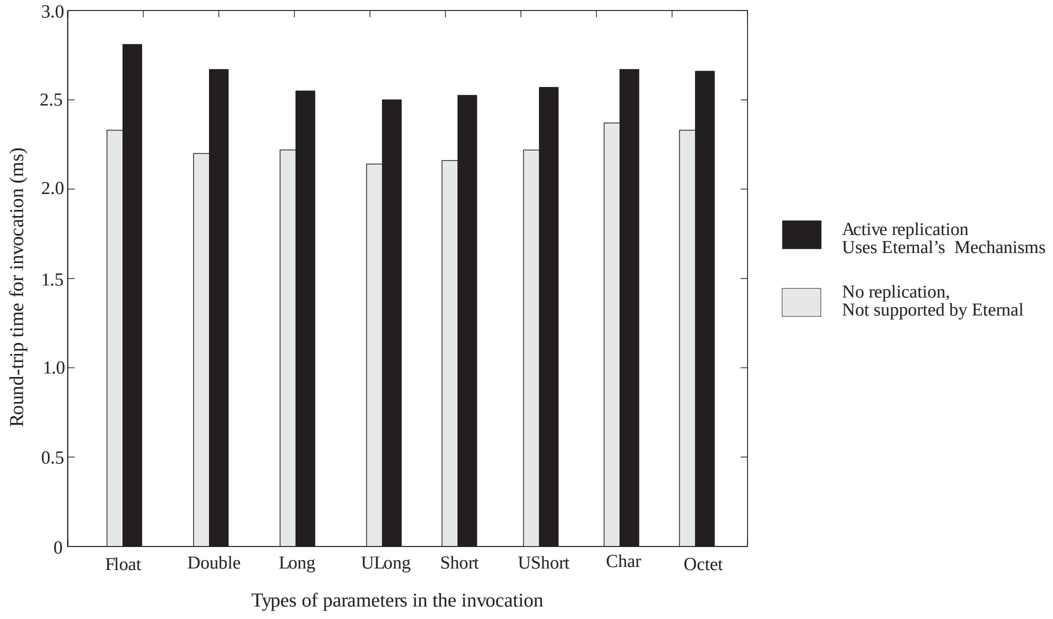


Figure 8.4: Round-trip times for the benchmark application for invocations that involve primitive IDL types as individual entities.

types (`float, double, long, short, unsigned long, unsigned short, char, octet`) as separate entities, primitive data types contained in CORBA `sequence`s, primitive data types in the form of CORBA `any`s, and primitive data types contained in CORBA `array`s.

For those methods of this interface that involve primitive types aggregated into a user-defined IDL type (through a CORBA `sequence` or a CORBA `array`), the amount of data being transferred with each invocation of the server object is normalized for the purposes of comparison. As shown in the figure, every `array` or `sequence` type in the `through` interface contains 1024 bytes (1kB) of data, regardless of the primitive IDL type that it contains.

These benchmarks primarily measure the round-trip time in a synchronous invocation of one of the methods of the server object's `through` interface. A synchronous invocation implies that the CORBA client object blocks after issuing the invocation, and cannot send another invocation until it receives the response from the server. When the benchmark application is unreplicated, the round-trip time reveals the speed of the ORB implementation (along with the platform and the communication infrastructure that supports the ORB) and the overhead in passing parameters of specific types. When the objects of the benchmark application are actively replicated, with Eternal's Mechanisms maintaining replica consistency using the Totem protocols, this round-trip time includes the additional overhead of using Eternal's infrastructure, as shown in Figure 8.2. The overhead due to Eternal encompasses the overhead due to interception, replication management and multicasting. These overheads are useful to the CORBA application programmer in determining the choice of data types to be used in method parameters, based on the efficiency of their transmission; in addition, the overheads are useful to Eternal in estimating the cost of state transfer.

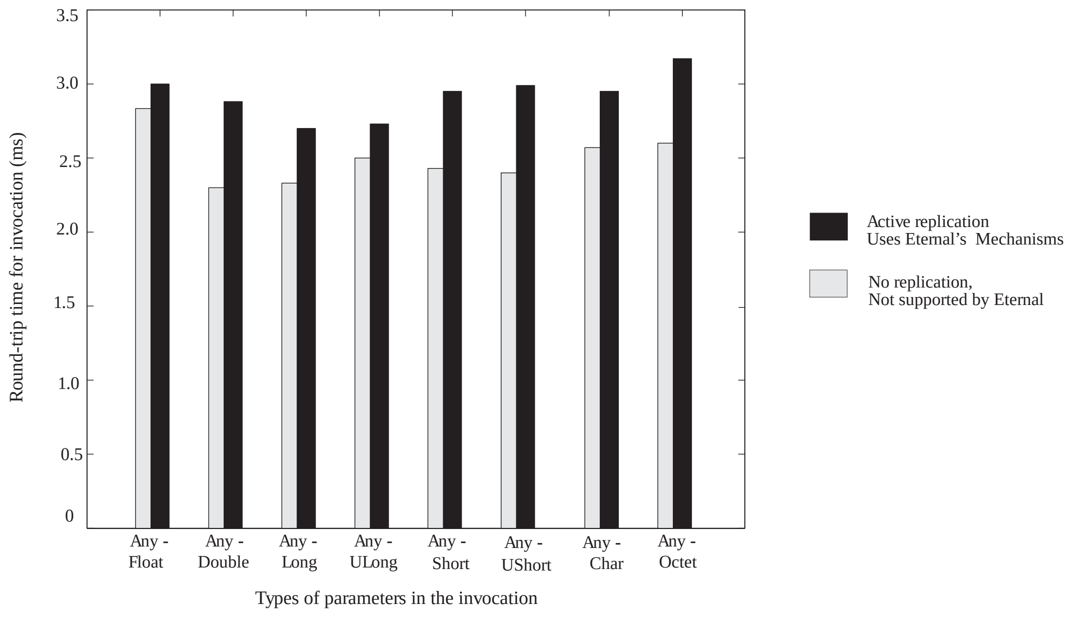


Figure 8.5: Round-trip times for the benchmark application for invocations that involve CORBA `any`s that encapsulating primitive IDL types.

The benchmark application is designed to work for several different ORBs; we have performed the tests for VisiBroker for C++, version 3.3, on a network of six 167Mhz SPARC workstations running Solaris 2.5.1.

8.2.2.1 Primitive IDL Types and CORBA `any`

The results from the benchmark experiments are shown in Figure 8.4 and Figure 8.5. The overheads due to the Eternal system are more or less the same for all of the primitive data types. Eternal's overhead is as low as 13% in the case of a `char` parameter in the client's invocation, and as high as 20% in the case of a `double` parameter in the client's invocation. The absolute round-trip time is greatest in the case of marshaling a CORBA `float` with Eternal's Mechanisms.

In the case of the CORBA `any` parameters, the overhead varies significantly, depending on the primtive IDL type that is being encapsulated in the CORBA `any` parameter. Eternal's overhead is as low as 6% for a CORBA `any` that represents a `float` value, and is as high as 21% for a CORBA `any` that represents an `unsigned short` value. The absolute round-trip time is greatest in the case of marshaling an `octet` into a CORBA `any`.

8.2.2.2 Arrays

The results from the benchmark experiments are shown in Figure 8.6. The absolute round-trip time for `arrays` is understandably larger than for primitive data types or `anys`. The overheads due to Eternal vary significantly, depending on the primtive IDL type that is

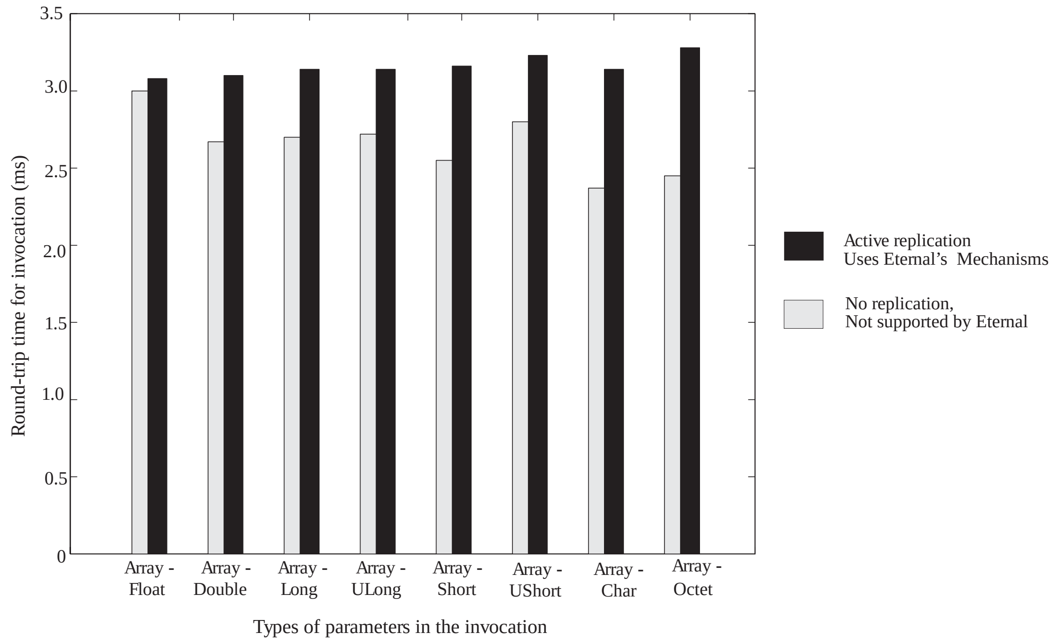


Figure 8.6: Round-trip times for the benchmark application for invocations that involve CORBA `array`s encapsulating primitive IDL types.

being aggregated into the CORBA `array`. Eternal's overhead is as low as 6% for a CORBA `array` that contains 256 `float`s, and is as high as 21% for a CORBA `array` that contains 1024 `octet`s of data. The round-trip time is greatest in the case of the `array` of `octet`s.

8.2.2.3 Sequences

The benchmark's results are shown in Figure 8.7. The absolute round-trip time does not vary much for the different types of data that are encapsulated into `sequence`s. The overheads due to Eternal also do not vary much, being more or less 20% for the different types. The absolute round-trip time, as well as Eternal's overhead, are very similar for a `string` of length 1024 bytes and for a `sequence` of 1024 `octet`s.

8.2.3 Different Levels of Fault Tolerance

The Eternal system can provide different levels of fault tolerance, depending on the needs of the application, and the types of faults that the application must be protected against.

Systems that are designed for a high level of fault tolerance incur a high associated overhead, primarily due to signature generation and verification, which are computationally expensive operations that depend on modular exponentiation.

For a high level of fault tolerance, the Eternal system utilizes the SecureRing protocols and CryptoLib [31], a library of routines for public and private key systems. Signatures are computed by RSA decrypting a message digest using the private key, while verification

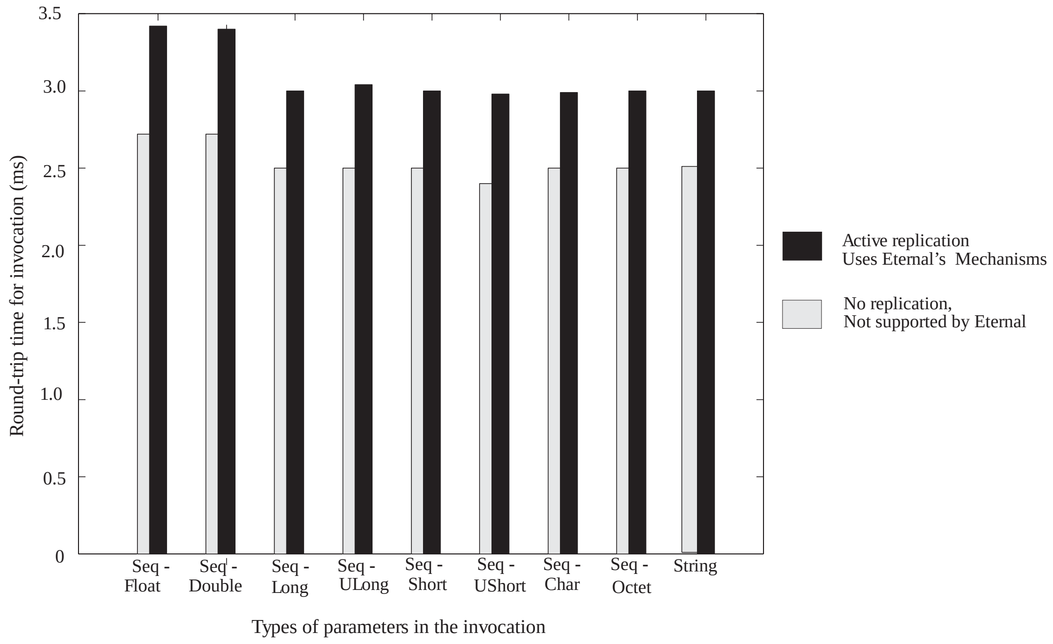


Figure 8.7: Round-trip times for the benchmark application for invocations that involve CORBA `sequence`s encapsulating primitive IDL types.

is performed by RSA encrypting the signature using the public key. Because the message digest is a fixed size (16 bytes), the time required for signing is independent of the size of the original message. However, signature generation time is highly related to key modulus size; thus, a tradeoff exists between performance and the degree of security attained.

Both Totem and the SecureRing protocols have been designed to amortize the cost of computing a signature over the number j of messages $m_1, \ldots, m_j$ sent per token visit. This parameter j can be tuned to achieve optimal performance for different applications

To measure the performance of Eternal for the different levels of fault tolerance, we used a simple test application developed with the VisiBroker 3.2 ORB. The measurements were taken over a network of six dual-processor 167 MHz UltraSPARC workstations, running the Solaris 2.5.1 operating system and connected by a 100 Mbps Ethernet.

The client object of the test application acts as a packet driver, sending a constant stream of one-way invocations at a specified rate to the server object. Each invocation is contained in a fixed-length (64 bytes) IIOP message. The rate at which the server object is invoked is varied at the client object; the throughput is measured at the server object.

The graph in Figure 8.8 shows the throughputs obtained with this test application for the following cases, which are listed in the order of increasing level of fault tolerance:

- **Case 1:** Unreplicated client and server objects without the Eternal system. The throughput is determined by the ORB mechanisms alone.

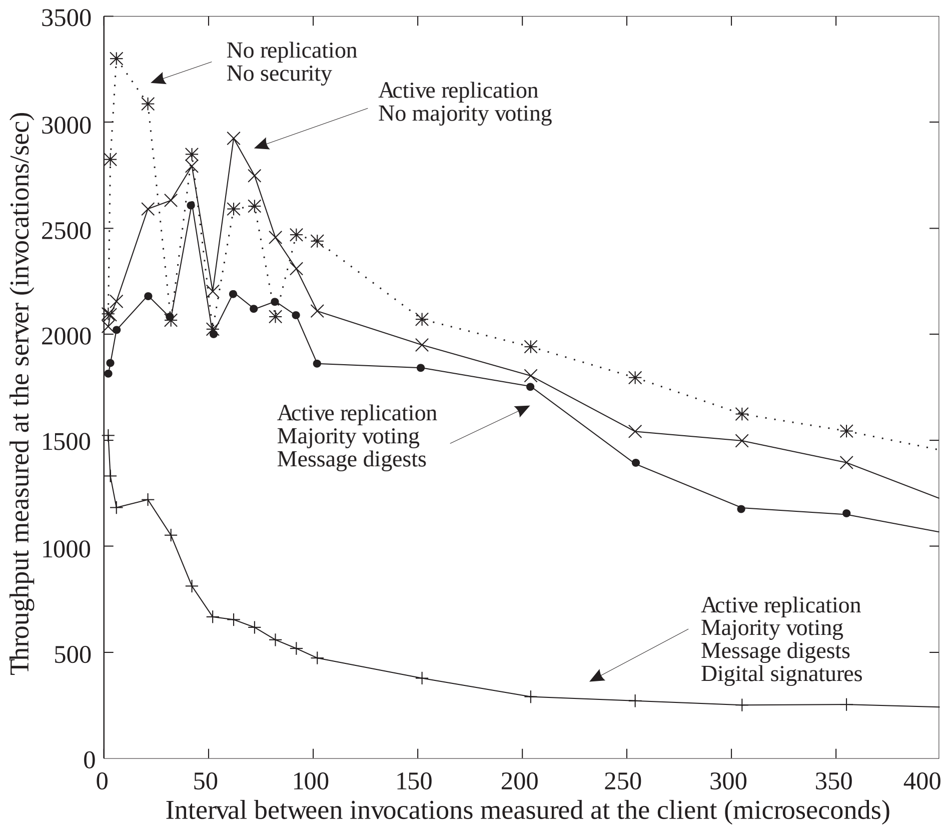


Figure 8.8: Performance of the Eternal system.

- **Case 2:** Three-way active replication of both client and server objects without majority voting. Reliable totally ordered multicasts without either the message digests or the signatures are used. The throughput is dictated by the cost of interception, active replication and multicasting, in addition to the costs of case 1. The Totem system is employed in this case.

- **Case 3:** Three-way active replication of both client and server objects with majority voting. Secure reliable totally ordered multicasts with message digests are used. The throughput is dictated by the cost of message digests, in addition to the costs of case 2. The SecureRing protocols are employed in this case.

- **Case 4:** Three-way active replication of both client and server objects with majority voting. Secure reliable totally ordered multicasts with message digests and digitally signed tokens are used. The throughput is dictated by the cost of signatures, in addition to the costs of case 3. The SecureRing protocols are employed in this case.

In the performance measurements for cases 2, 3 and 4, up to six multicast messages are sent with each token visit, where each multicast message encapsulates possibly multiple IIOP messages. While the cost of computing a single signature is spread over six messages, the use of signatures is nevertheless computationally expensive, as can be seen from the overheads of the Eternal system in case 4. However, the results indicate that the overheads of the Eternal system without signatures (cases 2 and 3) are low. In particular, the overheads are in the range of 7-15% for remote invocations for the triplicated clients and the triplicated servers of case 2. In all of the cases, the overheads are quite reasonable given the level of fault tolerance that the Eternal system provides.

The graph also indicates some transient behavior, attributable to the ORB, for cases 1, 2 and 3 when the time between consecutive invocations at the client is less than 100 μs. For such high message generation rates at the client, the ORB batches multiple one-way invocations before transmission. While some performance benefit is gained from this activity of the ORB, the unpredictability of the ORB's batching, evident from the transient behavior in the graph, can lead to undesirable fluctuations in the throughput of the application. This behavior of the ORB is not as significant in case 4, where the computation of the signatures dominates the CPU usage on each processor, effectively reducing the fraction of CPU time allocated to other processing, such as the ORB's batching of IIOP messages.

Chapter 9

Conclusion

Fault tolerance for CORBA could be provided entirely through CORBA service objects, located above the ORB, with application-level interfaces written in IDL. While it is necessary to expose some interfaces of the framework, particularly those for management, to the application for ease of use and customization, it is less desirable to expose the more difficult aspects of fault tolerance, such as replica consistency and fault recovery, through application-level interfaces. Moreover, implementation of fault tolerance above a CORBA ORB is not necessarily the most efficient approach due to the overhead of the ORB in the communication paths.

On the other hand, fault tolerance for CORBA could be provided by embedding the fault tolerance mechanisms within the ORB. Unfortunately, this involves modifying the ORB to provide the necessary support. The extent of the modification to the ORB depends on the ORB, as well as on the level of fault tolerance provided, with the likelihood that the resulting modified ORB is non-compliant with the CORBA standard. However, because the mechanisms form an intrinsic part of the ORB, the new functionality can be made available in a way that is transparent to the application.

The novel interception approach that we have developed allows the transparent insertion of fault tolerance mechanisms underneath the ORB. Interception achieves the best of the integration and the service approaches, while providing other benefits as well. The Eternal system exploits the interception approach to provide transparent fault tolerance to CORBA, without requiring the modification of either the ORB or the application. Eternal comprises a framework that combines Mechanisms inserted underneath the ORB for transparency and efficiency, and Services implemented above the ORB for application-level control and ease of use.

Eternal's Services above the ORB include the Replication Manager that replicates each application object, according to user-specified fault tolerance properties (including the choice of replication style) and distributes the replicas across the system. Eternal's Mechanisms underneath the ORB include the Interceptor, the Replication Mechanisms and the Logging-Recovery Mechanisms.

The Interceptor of the Eternal system transparently captures the IIOP messages exchanged between the application objects, and diverts the intercepted IIOP messages to the Replication Mechanisms. The Replication Mechanisms, together with the Logging-Recovery Mechanisms, maintain strong consistency of the replicas, and detect and recover from faults.

Eternal maintains strong replica consistency of the application objects, as replicas and processors fail and recover, and as replicas perform operations that update their states. Eternal provides additional transparent mechanisms to enforce deterministic behavior, and to guarantee strong replica consistency, even in the face of multithreading in the ORB or in the application.

In the Eternal system, both the client and server objects of the CORBA application can be replicated, and support for nested operations is provided. Different replication styles – active, cold passive and warm passive replication – are provided, with the user selecting the replication style appropriate to the application object at system configuration time.

To facilitate strong replica consistency, the Eternal system conveys the IIOP messages of the CORBA application using the reliable totally ordered multicast messages of the underlying Totem system. Eternal can also tolerate arbitrary faults by exploiting protocols such as those of the SecureRing system, with more stringent guarantees than are provided by Totem. To tolerate value faults in the application, Eternal uses active replication with majority voting applied to both invocations and responses for every application object.

The technology of the Eternal system formed the basis of our response to the Object Management Group's Request for Proposals on fault-tolerant CORBA. With our close involvement in the ongoing OMG standardization process, the technology of the Eternal system is likely to form the basis of the forthcoming standard for Fault Tolerant CORBA.

9.1 Outstanding Challenges

9.1.1 ORB State

A number of challenges still exist in providing fault tolerance for CORBA. Most of these challenges surround the issue of strong replica consistency, and the factors that influence it. One of these factors, the ORB state, is adequately addressed by Eternal for request identifiers and socket connections. However, the part of the ORB state that stores information about the threads in the application, and the specific multithreading policy employed by the ORB, requires more synergy between Eternal and the ORB.

Unfortunately, multithreading is so closely tied to the ORB that any thread-level hooks into the ORB will be necessarily vendor-specific. In the absence of such hooks, it is relatively difficult to provide a way of extracting the thread-specific part of the ORB state from underneath, or from above, the ORB.

9.1.2 Partitioning and Remerging

When a network partitioning fault occurs, every object group (replicated object) might also partition into disjoint subgroups, each subgroup containing a subset of the original set of replicas, located in a different component of the partitioned system. Thus, replicas in the disjoint subgroups cannot communicate with each other. Different operations may be performed by the replicas in the different subgroups, leading to inconsistencies that must be resolved when communication is reestablished and the subgroups remerge.

In Eternal, for each replicated object, at most one primary subgroup is identified when the network partitions. Each of the other components is then a secondary subgroup for that object. At the point of remerging, while the Logging-Recovery Mechanisms transfer the state of the replicas in the primary subgroup to those in the secondary subgroup, fulfillment methods permit operations performed in the secondary subgroup also to be performed in the larger merged subgroup. The fulfillment methods may need to handle special application-specific conditions, and to resolve inconsistencies, by no means an easy task.

9.1.3 Live Upgrades

The Eternal system also exploits object replication to achieve more than fault tolerance. The ability to mask the failure of an object or a processor can also be used to mask the deliberate removal of an object or processor and its replacement by an upgraded object or processor. For the upgrade of a processor, the replacement can be a different type of processor. It is also possible, in several steps, to replace an object by another object with a different interface or implementation, without stopping the system and without requiring great system programming skill from the application developer. Over time, both hardware and software components of the system can be replaced and upgraded without interrupting the service provided by the system. Thus, our objective is a system that can run forever, a system that is Eternal.

Bibliography

[1] D. A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, August 1994.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–33, August 1998.

[3] A. D. Alexandrov. *User-level operating system extensions based on system call interposition*. PhD thesis, Department of Computer Science, University of California, Santa Barbara, June 1999.

[4] R. Balzer and N. Goldman. Mediating connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, pages 73–77, Austin, TX, May 1999.

[5] A. Baratloo, P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Hot replication of Java RMI server objects. In *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 65–78, Santa Fe, NM, April 1998.

[6] S. Bestaoui. One solution for the nondeterminism problem in the SCEPTRE 2 fault tolerance technique. In *Proceedings of the Euromicro 7th Workshop on Real-Time Systems*, pages 352–358, Odense, Denmark, June 1995.

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

[8] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the IEEE 28th International Conference on Fault-Tolerant Computing*, pages 128–137, Munich, Germany, June 1998.

[9] Charles University and MLC Systeme GmbH. CORBA Comparison Project. Technical report, http://nenya.ms.mff.cuni.cz, August 1999.

[10] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[11] T. Curry. Profiling and tracing dynamic library usage via interposition. In *Proceedings of the Summer 1994 USENIX Conference*, pages 267–78, Boston, MA, June 1994.

[12] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the 24th IEEE Fault-Tolerant Computing Symposium*, pages 298–307, Austin, TX, June 1994.

[13] Eternal Systems and Sun Microsystems. Fault tolerance for CORBA, initial joint submission. OMG Technical Committee Document orbos/98-04-08, October 1998.

[14] Expersoft Corporation. *CORBAplus for C++ Documentation - v2.2.1*, June 1998.

[15] J. C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[16] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *Proceedings of the Winter 1991 USENIX Conference*, pages 243–52, Dallas, TX, January 1991.

[17] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[18] P. Felber, A. Schiper, and R. Guerraoui. Designing a CORBA group communication service. In *Proceedings of the IEEE 15th Symposium on Reliable Distributed Systems*, pages 150–159, Niagara on the Lake, Canada, October 1996.

[19] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[20] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley Longman, Inc., January 1999.

[21] H. Higaki and T. Soneoka. Fault-tolerant object by group-to-group communiations in distributed systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, pages 62–71, Saitama, Japan, October 1992.

[22] Inprise Corporation. *VisiBroker for C++ Programmer's Guide*, 1998.

[23] Iona Technologies PLC. *Orbix Programmer's Guide*, October 1997.

[24] Isis Distributed Systems Inc. and Iona Technologies Limited. *Orbix+Isis Programmer's Guide*, 1995.

[25] C. Jacquemot, F. Herrmann, P. S. Jensen, and P. Gautron. COOL: The Chorus CORBA compliant framework. In *Proceedings of the IEEE Proceedings of COMPCON '94*, pages 132–141, San Franciso, CA, February 1994.

[26] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi. *ILU 2.0alpha14 Reference Manual.* Xerox Corporation, January 1999.

[27] K. P. Kihlstrom. *Survivable Distributed Systems: Design and Implementation.* PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, August 1999.

[28] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the IEEE 31st Annual Hawaii International Conference on System Sciences*, volume 3, pages 317–326, January 1998.

[29] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads.* Prentice Hall, Mountain View, 1996.

[30] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for Object Request Broker middleware. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks*, Salem, MA, August 1999.

[31] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*, pages 1–17, October 1993.

[32] S. Lai-Lo and D. Riddoch. *omniORB2 Version 2.7.1 User's Guide.* AT & T Laboratories, Cambridge, U. K., February 1999.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[34] J. R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[35] C. A. Lingley-Papadopoulos. The Totem process group membership and interface. Master's thesis, University of California, Santa Barbara, August 1994.

[36] S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, 1995.

[37] S. Maffeis and D. C. Schmidt. Constructing reliable distributed systems with CORBA. *IEEE Communications Magazine*, 35(2):56–60, February 1997.

[38] P. M. Melliar-Smith and L. E. Moser. Simplifying the development of fault-tolerant distributed applications. In *Proceedings of the Workshop on Parallel/Distributed Platforms in Industrial Products, 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[39] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999.

[40] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[41] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.

[42] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[43] A. Mostefaoui and M. Raynal. Efficient message logging for uncoordinated checkpointing protocols. In *Proceedings of the 2nd European Dependable Computing Conference*, pages 353–364, Taormina, Italy, October 1996.

[44] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

[45] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 81–90, Portland, OR, June 1997.

[46] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. The interception approach to reliable distributed CORBA objects. In *Panel on Reliable Distributed Objects, Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 245–248, Portland, OR, June 1997.

[47] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, 1997.

[48] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the IEEE 18th Symposium on Reliable Distributed Systems*, pages 263–273, Lausanne, Switzerland, October 1999.

[49] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replication and recovery mechanisms for strong replica consistency in reliable distributed systems. In *Proceedings of the 5th ISSAT International Conference on Reliability and Quality in Design*, pages 26–31, Las Vegas, NV, August 1999.

[50] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, pages 62–68, July 1999.

[51] Object Management Group. The Common Object Services specification. OMG Technical Committee Document formal/98-07-05, July 1998.

[52] Object Management Group. Fault tolerant CORBA using entity redundancy: Request for proposals. OMG Technical Committee Document orbos/98-04-01, April 1998.

[53] Object Management Group. Portable interceptors: Request for proposals. OMG Technical Committee Document orbos/98-09-11, September 1998.

[54] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.3 edition. OMG Technical Committee Document formal/98-12-01, June 1999.

[55] Object-Oriented Concepts, Inc. *ORBacus for C++ and Java*, 1998.

[56] G. Parrington, S. Shrivastava, S. Wheater, and M. Little. The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, Summer 1995.

[57] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.

[58] R. L. Rivest. The MD4 message digest algorithm. In *Advances in Cryptology - Proceedings of CRYPTO '90*, pages 303–11, Santa Barbara, CA, August 1990.

[59] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978.

[60] B. S. Sabnis. Proteus: A software infrastructure providing dependability for CORBA applications. Master's thesis, University of Illinois at Urbana-Champaign, 1998.

[61] D. C. Schmidt. Evaluating architectures for multithreaded Object Request Brokers. *Communications of the ACM*, 41(10):54–60, October 1998.

[62] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998.

[63] J. Schonwalder, S. Garg, Y. Huang, A. P. A. van Moorsel, and S. Yajnik. A management interface for distributed fault tolerance CORBA services. In *Proceedings of the IEEE Third International Workshop on Systems Management*, pages 98–107, Newport, RI, April 1998.

[64] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the IEEE 26th International Symposium on Fault-Tolerant Computing*, pages 250–259, Sendai, Japan, June 1996.

[65] W. R. Stevens. *UNIX Network Programming.* Prentice Hall Software Series, 1990.

[66] Sun Microsystems Inc. *SunOS 5.x Linker and Libraries Guide*, November 1995. In Solaris Software Developer Kit.

[67] Sun Microsystems Inc. and International Business Machines Corporation. *RMI-IIOP Programmer's Guide*, FCS release edition, 1999.

[68] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software - Practice and Experience*, 28(9):963–79, July 1998.

[69] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[70] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.

[71] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

[72] Y. M. Wang and W. J. Lee. COMERA: COM extensible remoting architecture. In *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 79–88, Santa Fe, NM, April 1998.