SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:771–788 (DOI: 10.1002/spe.458)

Eternal—a component-based framework for transparent fault-tolerant CORBA

P. Narasimhan[1,*,†], L. E. Moser[2] and P. M. Melliar-Smith[2]

[1]*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.*
[2]*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, U.S.A.*

SUMMARY

Enterprises are increasingly involved in worldwide round-the-clock e-commerce and e-business, which requires them to be operational 24 hours per day, 7 days per week. With outages leading to loss of revenue, reputation and customers, fault tolerance becomes increasingly important. By mixing the fault tolerance logic into the application logic, existing fault tolerance practices render applications more complex, more prone to errors, and more difficult to maintain and build. The Eternal system is a component-based middleware framework that provides transparent fault tolerance for enterprise applications, and that ensures continuous 24 × 7 operation without requiring special skills of the application programmers. The Eternal system implements the new Fault-Tolerant CORBA standard. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: consistency; CORBA; enterprise; Eternal; replication; fault tolerance; framework; transparency

1. INTRODUCTION

E-commerce and e-business on the Internet with worldwide round-the-clock operation require enterprises to be operational 24 hours per day, 7 days per week. Outages are very expensive for these enterprises, with accompanying loss of revenue and reputation, and with disgruntled customers. Table I gives the estimates of the financial losses [1] due to downtime for different types of enterprises. With an increasing demand for 24 × 7 operation, and with downtime being unacceptable and prohibitive in cost, enterprises must have fault tolerance to survive in today's marketplace.

*Correspondence to: P. Narasimhan, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
†E-mail: priya@cs.cmu.edu

Contract/grant sponsor: Defense Advanced Research Projects Agency in conjunction with the office of Naval Research and the Air Force Research Laboratory; contract/grant number: N00174-95-K-0083, F3602-97-1-0248.

Table I. Financial impacts of outages for different types of enterprises, according to a December 1997 report from Dataquest. These figures probably underestimate the cost of downtime. Computer Economics and InfoCorp Consulting projected the cost of downtime in 1999 to be $6.6 billion due to lost productivity and revenues.

Industry sector	Business operation	Industry cost per hour of downtime ($)	Average cost per hour of downtime ($)
Financial	Brokerage operations	5 600 000–7 300 000	6 500 000
Financial	Credit card/sales authorization	2 200 000–3 100 000	2 600 000
Media	Pay-per-view television	67 000–233 000	150 000
Retail	Home shopping (TV)	87 000–140 000	113 000
Retail	Home catalog sales	60 000–120 000	90 000
Transportation	Airline reservations	67 000–112 000	90 000
Media	Teleticket sales	56 000–82 000	69 000
Transportation	Package shipping	24 000–32 000	28 000
Finance	ATM fees	12 000–17 000	15 000

Existing fault tolerance practices tend to combine the application logic with the fault tolerance logic in proprietary ways. Programmers with expertise in fault tolerance are scarce, and retraining existing programmers to build fault tolerance into the applications is not a viable option. With custom fault tolerance built into the application, a fault-tolerant application takes longer to develop and market, and becomes more difficult to understand, maintain and extend. The ability to field new applications quickly, to exploit market opportunities, and to survive in today's competitive marketplace, is becoming increasingly important.

Ideally, fault tolerance technology should be a 'black-box' framework, which provides services transparently to the application by hiding the complicated interactions within the infrastructure. However, black-box frameworks are more difficult to build, primarily because the framework has to 'work harder' to provide the level of transparency that the application programmer enjoys while using the framework.

The Common Object Request Broker Architecture (CORBA) [2] middleware supports e-commerce and e-business applications. CORBA applications consist of objects distributed across a system, with client objects invoking server objects, which return responses to the client objects after performing the requested operations. CORBA's Object Request Broker (ORB) acts as an intermediary in the communication between a client object and a server object, shielding them from any differences in their programming languages and their physical locations. CORBA's TCP/IP-based Internet Inter-ORB Protocol (IIOP) sustains this communication even if the client object and the server object use different operating systems, byte orders and hardware architectures.

The Eternal system [3] is a component-based framework that enhances CORBA by providing fault tolerance for CORBA applications, without requiring the application programmer to be concerned with the difficult issues of fault tolerance. The value of Eternal in developing fault-tolerant CORBA applications lies in the *transparency* of its approach, i.e. neither the application code nor the CORBA middleware needs to be modified to benefit from Eternal's fault tolerance. The transparency of Eternal

allows existing CORBA applications to be rendered fault-tolerant easily and quickly, by application programmers who have no special experience or training in fault tolerance.

At the same time, to enable the CORBA application programmer to be aware of the tradeoffs, limitations and design choices in deploying an application with Eternal, the building blocks of Eternal's framework, i.e. the key interactions and the key components, are captured in a way that is accessible to both CORBA application programmers and framework developers.

Our experience in designing and building the Eternal system led to our active participation in developing the new standard for Fault-Tolerant CORBA [4] that the Object Management Group (OMG), the CORBA standards body, approved in March 2000. The final specifications for the new Fault-Tolerant CORBA standard correspond closely to our fault tolerance framework—the Eternal system is the first commercial offering that implements the new standard.

2. CHALLENGES FOR CONSISTENT REPLICATION

Eternal provides fault tolerance for CORBA applications by replicating the objects of the application. The purpose of replication is to provide multiple, redundant, identical copies, or *replicas*, of an object so that the object can continue to provide useful services, even though some of its replicas fail, or as the processors hosting some of its replicas fail. The essence of replication is that the replicas of an object must be consistent in state.

For ensuring *strong replica consistency*, Eternal requires application objects to be *deterministic* (or to be rendered 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, then the two replicas will reach the same final state. The different components of Eternal interact to address the challenges in maintaining strong replica consistency.

- *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. Eternal achieves this by exploiting a reliable totally-ordered multicast protocol for conveying the IIOP invocations (responses) to the replicas of a CORBA server (client).
- *Duplicate operations.* Replication, by its very nature, may lead to duplicate operations. For example, if every replica of a three-way replicated client object invokes a method of a replicated server object, every server replica will receive three copies of the same invocation, one from each of the client replicas. Eternal ensures that such duplicate invocations (responses) due to a replicated client (server) object are filtered so that the server (client) object receives only a single copy of every distinct invocation (response).
- *Recovery.* When a new replica is activated, or when a failed replica is recovered, *before* it can start to operate, it must have the same state as the other replicas of the object. Eternal retrieves

[‡]In cases where the application is inherently non-deterministic, e.g. the application uses system-specific or processor-specific functions, Eternal provides mechanisms to identify and 'sanitize' such sources of non-determinism, thereby rendering the application deterministic from the viewpoint of replication.

the state from an existing operational replica of the object and transfers the state to the new or recovering replica before making it operational.

- *Multithreading*. Replicas of a multithreaded CORBA object may become inconsistent if the threads, and the operations that they execute are not carefully controlled. For multithreaded ORBs or objects that allow the simultaneous execution of multiple operations, Eternal provides mechanisms to ensure strong replica consistency, regardless of the multithreading of the ORB or the application.

3. FAULT TOLERANCE DOMAINS

Today's business-to-business applications span enterprises across the Internet, with the application objects of one enterprise communicating with, and performing operations on, the application objects of another enterprise. 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 Figure 1. Each enterprise is likely to be, and indeed should be, responsible only for the reliability of the objects under its control, but 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 over which an enterprise's fault tolerance framework exercises control constitutes a *fault tolerance domain*; any two fault tolerance domains (i.e. any two enterprises, each with its own fault tolerance domain) can be connected through a *gateway component*. The Eternal system constitutes the fault tolerance framework within the fault tolerance domain, with external clients using Eternal's gateway components to communicate with replicated servers inside the domain. Communication with the fault tolerance domain occurs through reliable multicast, while communication outside the fault tolerance domain is based on TCP/IP.

In Figure 1, the replicated objects inside one enterprise (represented by the fault tolerance domain in Europe) can communicate with the replicated objects inside another enterprise (represented by the fault tolerance domain in Asia) through the use of gateways. Unreplicated client objects (such as the customer in the United States, running on processor P1) can also exploit a gateway in order to communicate with the replicated objects inside a fault tolerance domain. To enable communication between the two enterprises, a wide-area fault tolerance domain (comprising a number of gateways) 'bridges' the European and Asian domains.

3.1. Gateway components

Eternal's gateway components serve as the 'entry point' for unreplicated or external clients into the fault tolerance domain. Unreplicated clients 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, would violate strong 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

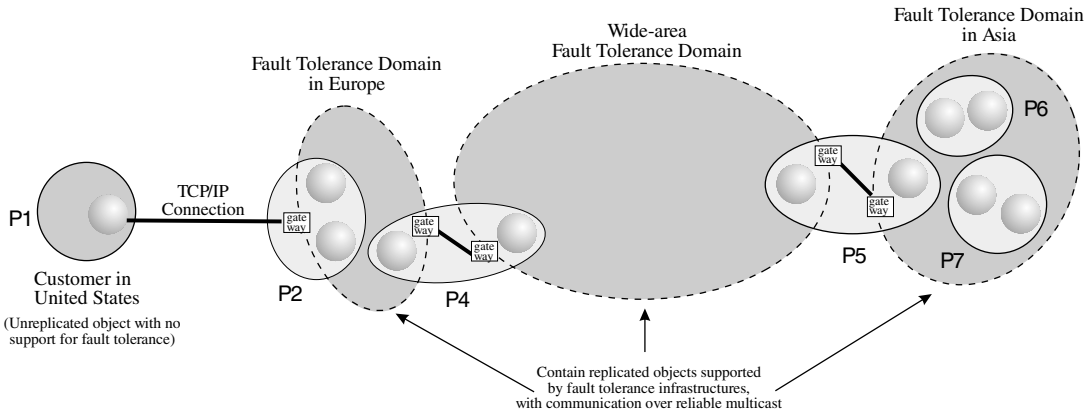

Figure 1. Gateways bridge the three fault tolerance domains and allow objects in one fault tolerance domain to communicate with those in another. The objects within each fault tolerance domain are replicated, and typically run over fault tolerance infrastructures. P*i* represents a processor hosting some application objects.

replicas, and invoke an operation on only that replica, making its state inconsistent with that of the other replicas of the server.

To prevent such surreptitious communication between an external client and an individual replica within the fault tolerance domain, Eternal provides additional mechanisms to ensure that an IOR[§] published by a replicated CORBA server within the fault tolerance domain directs external clients transparently to the gateway, rather than to any single server replica. This transparency implies that an unreplicated client is never aware of the existence of the fault tolerance domain, of the replication of the servers within the fault tolerance domain, or of Eternal's fault tolerance framework itself. The client implicitly assumes that its peer endpoint (which is, in fact, the gateway) is the real server and sends IIOP invocations (destined for the server) to the gateway, which then multicasts these messages to the target replicated server.

A single gateway component to a fault tolerance domain is insufficient to guarantee the level of reliability that customers of networked applications have come to expect. The use of redundant gateway components requires additional intelligence on the part of the client-side ORB to exploit the multiple gateway components. In the absence of the required support in current ORBs, Eternal provides a thin transparent client-side interception layer that mimics the simple fail-over support that an enhanced client-side ORB could provide to allow unreplicated CORBA clients to reconnect to an identical redundant gateway component transparently, should the first gateway component fail.

[§] An Interoperable Object Reference (IOR) of a CORBA server object contains the addressing information—host and port—of the server object. Clients use IORs to connect to the server.

3.2. Firewall support

Eternal's gateway component can also serve as a CORBA firewall as a secondary purpose. In firewall mode, a gateway component must be discriminating of messages that it forwards into, or out of, the fault tolerance domain. Client-side firewalls must provide outbound protection, i.e. they must allow only authorized clients to 'talk' to the outside world, while server-side firewalls must provide inbound protection, i.e. they must allow only authorized clients to 'talk' to the servers within the fault tolerance domain.

Because IIOP is a TCP/IP-based protocol, Eternal's server-side firewall can act as a TCP/IP proxy and can use packet filters at the transport level. For each incoming message, these TCP/IP filters determine the client's identity (host and port) and consult the domain's Access Control List (ACL) to determine if the message can be safely multicast to the replicated server inside the domain. A client-side firewall can be implemented by using the well-known SOCKS proxy mechanism. Here, the Eternal's thin client-side interception layer is SOCKS-enabled and communicates with a client-side SOCKS-enabled proxy server. The client-side library authenticates the client to the proxy server; on successful authentication, the proxy server establishes a connection and subsequently forwards messages to the real target of the client's requests. Unfortunately, the server-side TCP/IP proxy and the client-side SOCKS mechanism both operate and authenticate CORBA clients at the TCP/IP level. However, they have no way of verifying that the TCP/IP packets contain valid IIOP messages, making it possible for a client to 'feed' malicious information within IIOP messages after successful authentication by the firewall.

Thus, server-side firewalls must be equipped with IIOP filters that examine every IIOP message *header* to discover the name of the operation to be invoked, the identity of the server object and the identity of the client. However, the IIOP message *body* is encoded in CORBA's Common Data Representation (CDR) format. CDR translates IDL data types into a byte-independent octet stream, whose decoding requires knowledge of the interface definitions of the encoded objects, information that is certainly not available to an IIOP message filter.

The problem of decoding the IIOP message body can be overcome using a server-side proxy object, above the ORB, in conjunction with the TCP-level firewall mechanisms. The proxy object must receive and verify all incoming requests and multicast only 'safe' requests to the target replicated servers within the fault tolerance domain. This requires the CORBA proxy to support CORBA's Dynamic Skeleton Interface (DSI), to allow it to receive any request, regardless of the proxy's interface. The CORBA proxy also requires the use of CORBA's Dynamic Invocation Interface (DII) to forward a 'safe' request to a server, without any knowledge of the server's interface.

Furthermore, the proxy object (as a real CORBA object) has access to all CORBA services, including CORBA's Security Service, and can thus handle CORBA's entire security functionality. Apart from its support for firewalls, CORBA provides for additional security through the Security Service, the SecIOP specification and the integration of SSL into the ORB. The CORBA Security Service is implemented through ORB-level interceptors that can authenticate and audit certain events. CORBA's SecIOP can be used with IIOP for secure interoperability between different vendor's implementations of the Security Service. The integration of SSL into the ORB involves inserting SSL between the IIOP and the TCP/IP layers, to provide for authentication, data privacy and data integrity. The ORB–SSL integration, used in conjunction with the firewall mechanisms, is the most suitable choice for Internet e-business and e-commerce systems.

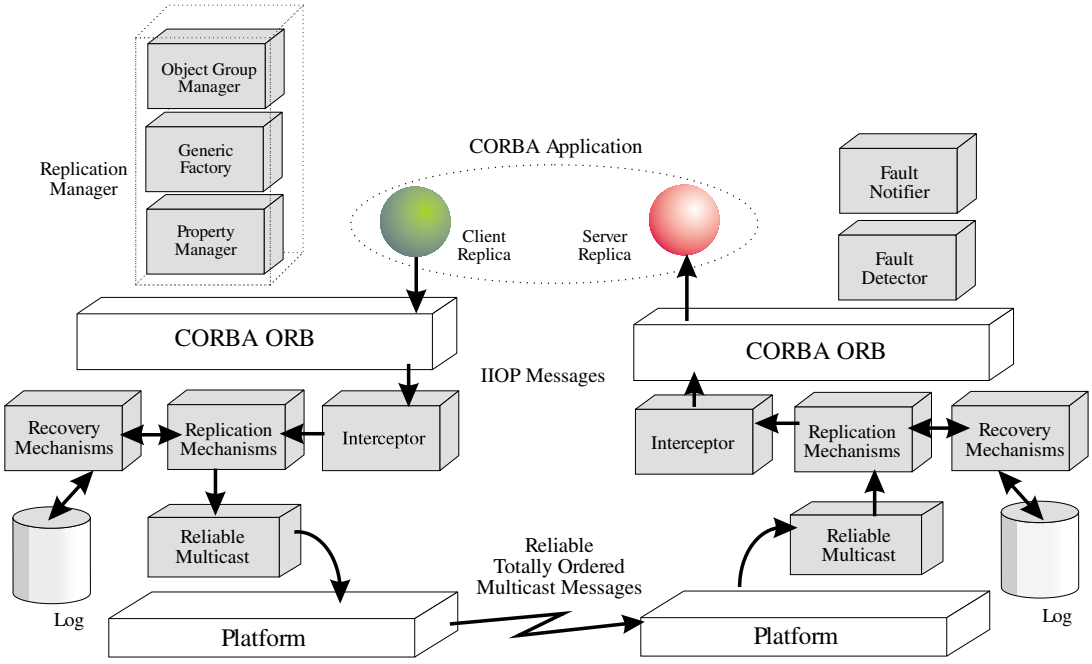

Figure 2. Eternal's OMG-compliant fault tolerance framework, consisting of the shaded components that are above and below the ORB.

4. THE ETERNAL SYSTEM

Eternal's OMG-compliant fault tolerance framework, used within the fault tolerance domain, consists of components both *above* the ORB and *below* the ORB, as shown in Figure 2. Because Eternal's components above the ORB are composed of CORBA objects, they can also be replicated (just as the application's CORBA objects are) by Eternal, with an adequate number of replicas distributed across the processors in the system. On the other hand, Eternal's components underneath the ORB, referred to as Eternal's Mechanisms for the sake of clarity, are composed of non-CORBA C++ objects and must be present on every processor within the Eternal-controlled fault tolerance domain.

Using Eternal's fault tolerance framework, both client and server objects of the CORBA application can be replicated, with the replicas distributed across the system. Eternal supports different replication styles—active, cold passive and warm passive replication—for every application object. To facilitate replica consistency, the Eternal framework conveys the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying Totem system [5].

The Replication Manager component replicates each application object, according to user-specified requirements, and distributes the replicas across the system. The Fault Detector component detects

the failure of replicas, objects and processors in the system. The Fault Notifier component uses the information gathered by the Fault Detector to notify other interested components of faults that have occurred in the system.

The 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 Replication Mechanisms for multicasting via Totem. The Replication Mechanisms, together with the 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 Replication Manager, the Fault Detector and the Fault Notifier are themselves implemented as collections of CORBA objects and, thus, can benefit from Eternal's fault tolerance.

4.1. Replication manager

To manage the replication of an object, Eternal employs the notion of an object group, where the members of the group correspond to the replicas of an object. In Eternal, both client and server objects can be replicated and, thus, constitute object groups.

The Replication Manager is a crucial component of the Eternal's fault tolerance framework and handles the creation, deletion and replication of both the application objects and the framework objects within the fault tolerance domain. The Replication Manager component replicates objects and distributes the replicas across the system. Although each replica of an object has an individual object reference, the Replication Manager component fabricates an object group reference for the replicated object that clients use to contact the replicated object. The Replication Manager's functionality is achieved through the Property Manager, Generic Factory and Object Group Manager components.

4.1.1. Property Manager

The Property Manager component allows a user to assign values to a number of fault tolerance properties for every application object that is to be replicated. Eternal provides the user with the flexibility to configure the replication of every application object by assigning the values of various fault tolerance properties, including:

- *Replication Style*—stateless, actively replicated, cold passively replicated or warm passively replicated. Eternal's support for active replication and warm passive replication are shown in Figure 3.
- *Membership Style*—addition of replicas to, or removal of replicas from, the object group is application controlled or infrastructure controlled.¶
- *Consistency Style*– replica consistency (including recovery, checkpointing, logging, etc.) is application controlled or infrastructure controlled.
- *Factories*—objects that create and delete the replicas of the object.

¶'Infrastructure controlled' is equivalent to 'Eternal controlled'.

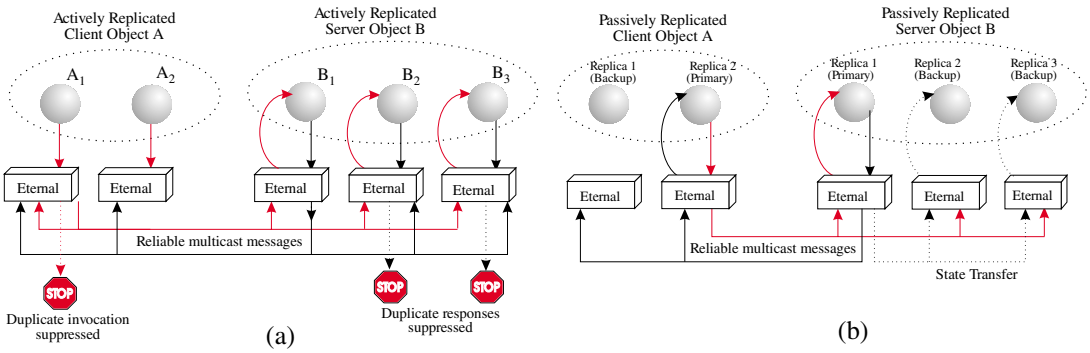

Figure 3. Two of the replication styles supported by the Eternal system for every client or server object include the (a) active replication style, and the (b) warm passive replication style.

- *Initial Number of Replicas*—the number of replicas of the object to be created initially.
- *Minimum Number of Replicas*—the number of replicas of the object that must exist for the object to be sufficiently protected against faults.
- *Checkpoint Interval*—the frequency at which the state of an object is to be retrieved and logged for the purposes of recovery.

Infrastructure-controlled Membership Style, in conjunction with infrastructure-controlled Consistency Style, is favored for the development of fault-tolerant CORBA applications, because it provides the maximal ease of use and transparency to the application, with the assurance of strong replica consistency, which the Eternal framework maintains under both fault-free and recovery conditions.

4.1.2. Generic Factory

The Generic Factory component allows users to create replicated objects in the same way that they would normally create unreplicated objects. The Generic Factory interface is inherited by the Replication Manager component to allow the application to invoke the Replication Manager directly to create and delete replicated objects. When asked to create a replicated object through its Generic Factory interface, the Replication Manager component, in turn, delegates the operation to the factories on the processors where the individual replicas of the object are to be created.

4.1.3. Object Group Manager

The Object Group Manager component allows users to control directly the creation, deletion and location of individual replicas of an application object. While this violates replication transparency (because the user is explicitly aware of the replicas of an object), and must be used with care so that

replica consistency is maintained, it is useful for expert users who wish to exercise direct control over the replication of application objects.

4.2. Fault Detector and Fault Notifier

The Fault Detector component is capable of detecting host, process and object faults. Each application object inherits a `Monitorable` interface to allow the Fault Detector component to determine the object's status. The Fault Detector component communicates the occurrence of faults to the Fault Notifier.

On receiving reports of faults from the Fault Detector component, the Fault Notifier component filters them to eliminate any inappropriate or duplicate reports. The Fault Notifier component then distributes fault event notifications to all of the objects that have subscribed to receive such notifications. The Replication Manager component is one such subscriber.

Eternal allows the user to influence fault detection for an object through the following fault tolerance properties:

- *Fault Monitoring Style*—the object is monitored by periodic 'pinging' (pull monitoring) of the object, or by periodic 'i-am-alive' messages (push monitoring) sent by the object.
- *Fault Monitoring Granularity*—the replicated object is monitored on the basis of an individual replica, a location, or a location-and-type.
- *Fault Monitoring Interval*—the frequency at which an object is to be 'pinged' to detect if it is alive or has failed.

As shown in Figure 4, the Fault Detection framework can be structured in a hierarchical way, with the global replicated Fault Detector component triggering the operation of local fault detector components located on each processor. Any faults detected by the local fault detectors are reported to the global replicated Fault Notifier component. The Replication Manager component, being a subscriber of the Fault Notifier component, receives reports of any faults that occur in the system and can initiate appropriate actions to enable the system to recover from the faults.

4.3. Interceptor

Eternal's Interceptor is a non-ORB-level, non-application-level component that transparently 'attaches' itself to every CORBA object at runtime, without the object's or the ORB's knowledge, and that can modify the object's behavior as desired. The Interceptor ensures that the application's IIOP messages (containing the client's invocations and the server's responses), originally destined for TCP/IP, are diverted instead to the Replication Mechanisms. The advantage of the Interceptor, located underneath the ORB, is not only its transparency to the ORB and to the application, but also its implementation in an ORB-independent manner.

Eternal's Interceptor currently employs the library interpositioning hooks found on Unix and Windows NT. Library interpositioning involves the transparent runtime replacement of the socket-level library routines used by the CORBA application for connection establishment over TCP/IP. Eternal's Interceptor captures, and redefines, those routines so that the application's TCP/IP connections are transparently converted into connections to the Replication Mechanisms. Once the connections are

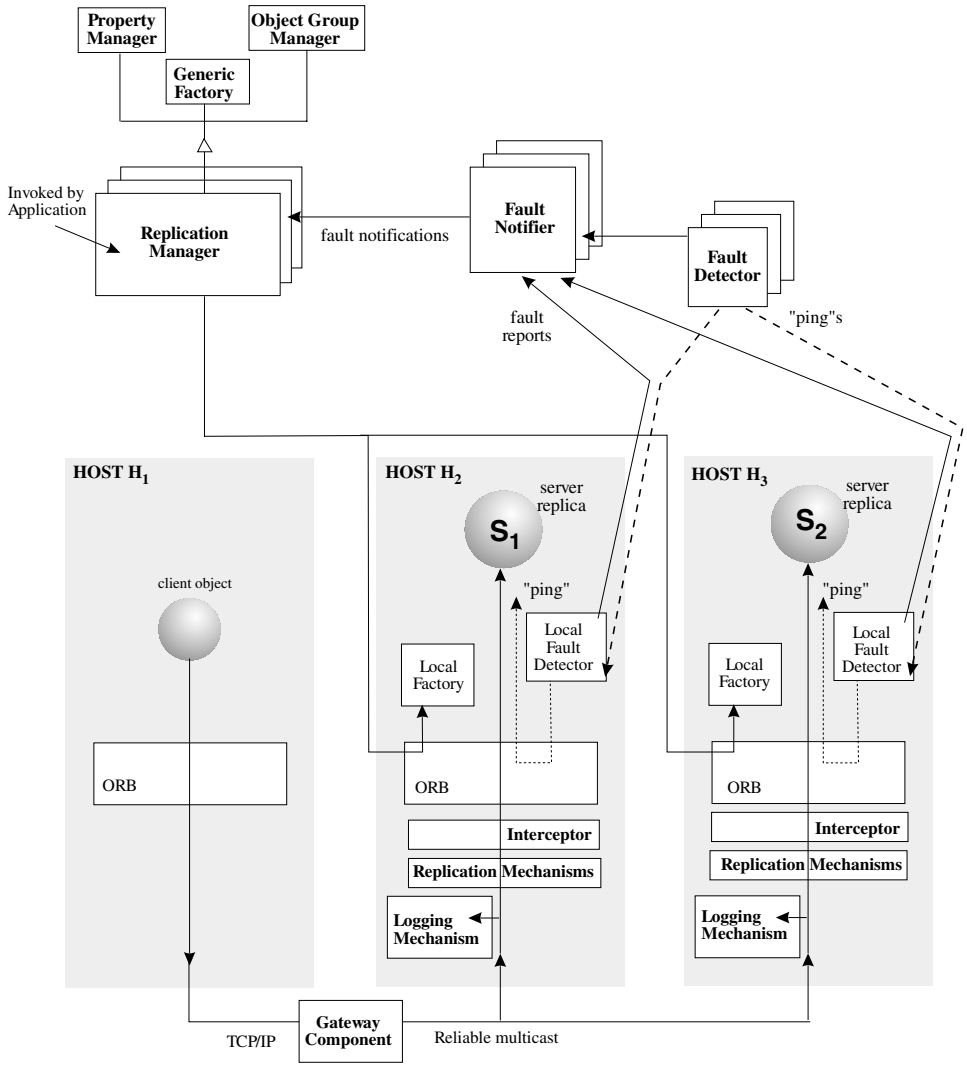

Figure 4. Interaction between Eternal's components that are below the ORB (namely, the Interceptor, the Replication Mechanisms, the Logging/Recovery Mechanisms and the Gateway) and those that are above the ORB (namely, the Replication Manager, the Fault Notifier and the Fault Detector).

established, the Interceptor adds no overhead in the path of message communication, because the application automatically (and unknowingly) uses the connection to the Replication Mechanisms to send IIOP messages.

4.4. Replication Mechanisms

The interaction between Eternal's components above the ORB and Eternal's Mechanisms, i.e. Eternal's components below the ORB, is shown in Figure 4. To facilitate replica consistency, Eternal's Replication Mechanisms (underneath the ORB) convey the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying multicast group communication protocol [5].

Eternal's Replication Mechanisms perform different operations for the different replication styles, as shown in Figure 4. For an actively replicated server (client) object, each replica responds to (invokes) every operation. Thus, the Replication Mechanisms deliver every IIOP invocation (response) intended for a replicated server (client) to every server (client) replica through the Interceptor. For active replication, the failure of a single active replica is masked due to the presence of the other active replicas that are also performing the operation.

For a passively replicated server (client) object, only one of the replicas, designated the *primary*, responds to (invokes) every operation. The remaining replicas of the object are referred to as the *backup* replicas. The Replication Mechanisms deliver every IIOP invocation (response) only to the primary replica of a passively replicated server (client) object. In the case of warm passive replication, the backup replicas are synchronized periodically with the primary replica. In the case of cold passive replication, the backup replicas are not loaded, but Eternal periodically retrieves, and stores in a log, the state of the primary replica. In the event that the primary replica fails, one of the backup replicas takes over as the new primary replica.

4.5. Recovery Mechanisms

Every replicated CORBA object can be regarded as having three kinds of state: *application state* (known to, and programmed into the object by, the application programmer), *ORB state* (maintained by the ORB for the object) and *infrastructure state* (invisible to the application programmer and maintained for the object by Eternal). Application state is typically represented by the values of the data structures of the replicated object. ORB state is vendor-dependent and consists of the values of the data structures (last-seen request identifier, threading policy, etc.). Infrastructure state is independent of, and invisible to, the replicated object as well as the ORB, and involves information that Eternal maintains for consistent replication.

Eternal's Recovery Mechanisms (underneath the ORB) ensure that all of the replicas of an object are consistent in application, ORB and infrastructure state. The Recovery Mechanisms handle the restoration of a new primary replica's state, as well as the periodic retrieval of an operational primary replica's state. The transfer of state to a new or recovering replica includes the transfer of application state to the new replica, ORB state to the ORB hosting the new replica and infrastructure state to the Recovery Mechanisms that manage the new or recovered replica. To enable application state to be captured and logged for the purposes of recovery, every replicated CORBA object must inherit

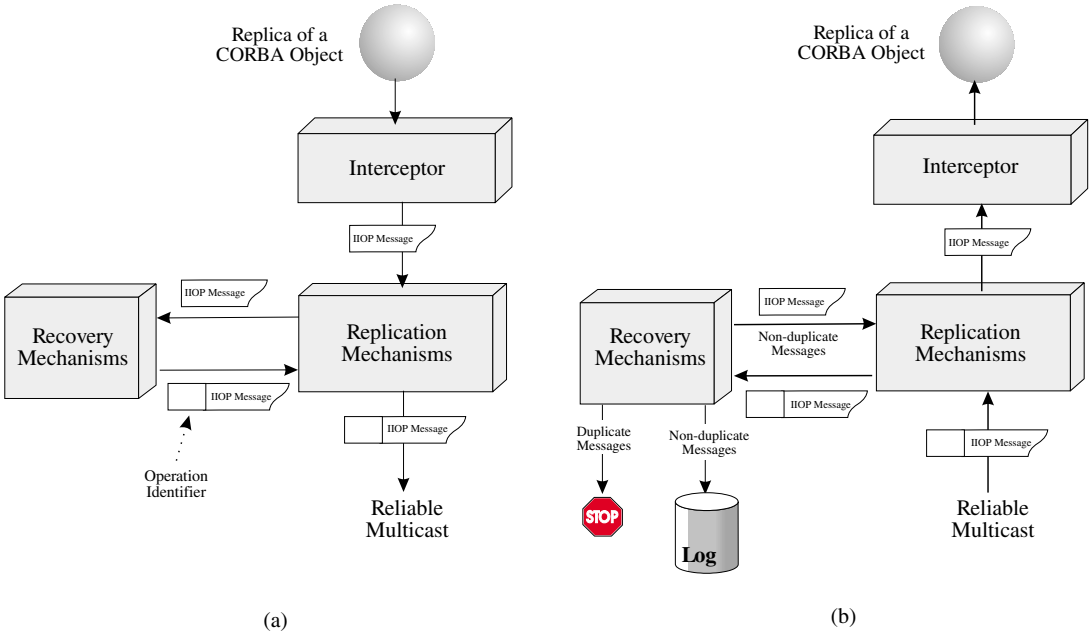

Figure 5. Interaction between Eternal's Interceptor, Replication Mechanisms and Recovery Mechanisms for (a) outgoing messages and (b) incoming messages.

a `Checkpointable` interface that contains methods for retrieving (`get_state()`) and assigning (`set_state()`) an object's state.

Because state retrieval from an existing active (primary passive) replica occurs at a different point in the message sequence from the assignment of the retrieved state to the new active (backup passive) replicas, the Recovery Mechanisms at the state retrieval and assignment locations synchronize the retrieval and state assignment messages. Furthermore, the Recovery Mechanisms log all new messages that arrive for a replica while its state is being assigned for delivery after the state assignment is complete.

To enable incoming response messages to be matched with their corresponding invocations, and to ensure that the target application objects receive only one copy of every distinct invocation or response intended for them, the Recovery Mechanisms insert an operation identifier into the Eternal-specific header for each outgoing IIOP message.

By deriving operation identifiers from the unique totally-ordered sequence numbers assigned by the underlying multicast group communication protocol to each message that it delivers, the Recovery Mechanisms on *different* processors assign the same operation identifier for a specific operation. Thus, if a three-way replicated client invokes an operation, the three duplicates (one from each replica) of the same invocation carry the same operation identifier in the Eternal-specific header; distinct invocations are assigned distinct operation identifiers. At the Recovery Mechanisms hosting the target

server replicas, the first of the three invocations to arrive is delivered to the server; the examination of the operation identifiers of the subsequently received duplicates leads to their suppression. The interaction between the Interceptor, Replication Mechanisms and Recovery Mechanisms is shown in Figure 5.

4.6. Scheduler Component

To preserve replica consistency for multithreaded objects, the Eternal system enforces deterministic behavior across all of the replicas of a multithreaded object by controlling the dispatching of threads and operations identically within every replica through a deterministic Scheduler Component. The Scheduler is not shown in Figure 2; it augments the Replication Mechanisms with support for strong replica consistency in the face of multithreaded applications.

The Scheduler Component dictates the creation, activation, deactivation and destruction of threads, within every replica of a multithreaded object, as required for the execution of the current operation 'holding' the logical thread-of-control. Exploiting the thread library interpositioning mechanisms of Eternal's Interceptor, the Scheduler Component can transparently override any thread or operation scheduling performed either by the non deterministic multithreaded ORB within the replica or by the replica itself.

Based on the incoming totally-ordered message sequence, the Scheduler Component at each replica decides on the immediate delivery, or the delayed delivery, of the messages to that replica. At each replica, the Scheduler Component's decisions are identical and, thus, operations and threads are dispatched identically at each replica, ensuring deterministic operation across all of the replicas of an object.

5. IMPLEMENTATION AND PERFORMANCE

The current implementation of Eternal's fault tolerance framework provides transparent fault tolerance to unmodified CORBA applications running over the following unmodified commercial implementations of CORBA over standard operating systems (Solaris 2.x, Red Hat Linux 6.0 and HP-UX 10.20):

- VisiBroker from Inprise Corporation;
- Orbix from Iona Technologies;
- e*ORB and CORBAplus from Vertel (previously Expersoft);
- ORBacus from Object-Oriented Concepts, Inc.;
- TAO from Washington University, St. Louis;
- omniORB2 from AT&T Laboratories, U.K.; and
- ILU from Xerox PARC.

The efficient Totem multicast group communication protocol allows Eternal to provide fault tolerance with minimal overhead. Using Eternal, for Solaris 2.x on 167 MHz SPARC workstations connected by a 100 Mbps Ethernet, when application objects are actively replicated, test applications

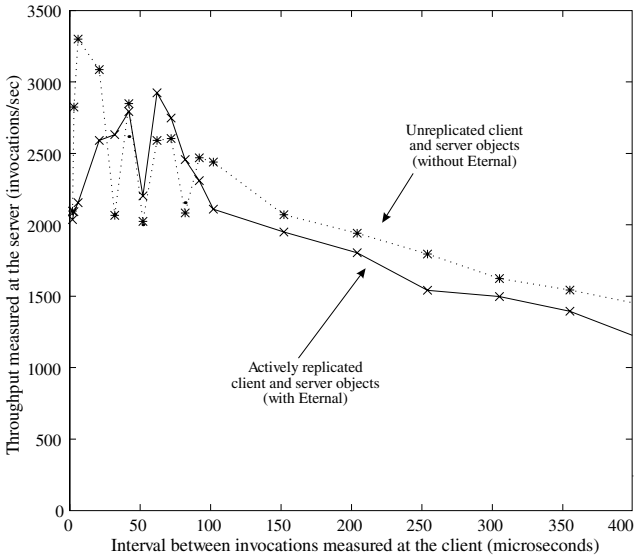

Figure 6. Variation of the throughput measured at the server with the interval between invocations at the client side. These results are shown both for an unreplicated application (without Eternal) as well as its three-way actively replicated counterpart (with Eternal).

typically incur only a 10% increase in round-trip invocation time, compared with their unreplicated unreliable counterparts. For RedHatLinux 6.0 on 400 MHz Intel Pentium processors connected by a 100 Mbps Ethernet, this overhead reduces to 3%.

To measure the performance of Eternal, 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. Figure 6 shows the throughput obtained for the three-way active replication of both the client and the server objects using Eternal, as compared with the throughput obtained for the unreplicated client and server objects without Eternal.

The performance of the Eternal system during the recovery of a new or failed replica of an object is shown in Figure 7. The time to recover a server replica in a test application developed with Inprise's VisiBroker 4.0 C++ ORB is shown. The measurements were taken over a network of dual-processor 167 MHz UltraSPARC workstations, running Solaris 2.7, and connected by a 100 Mbps Ethernet. The time to recover such a failed replica was measured as the time interval between the relaunch of the failed replica and the replica's reinstatement to normal operation. The recovery times obtained with this test application for varying sizes (from 10 bytes to 350 000 bytes) of the application-level state that is transferred across the network to recover a failed server replica are shown.

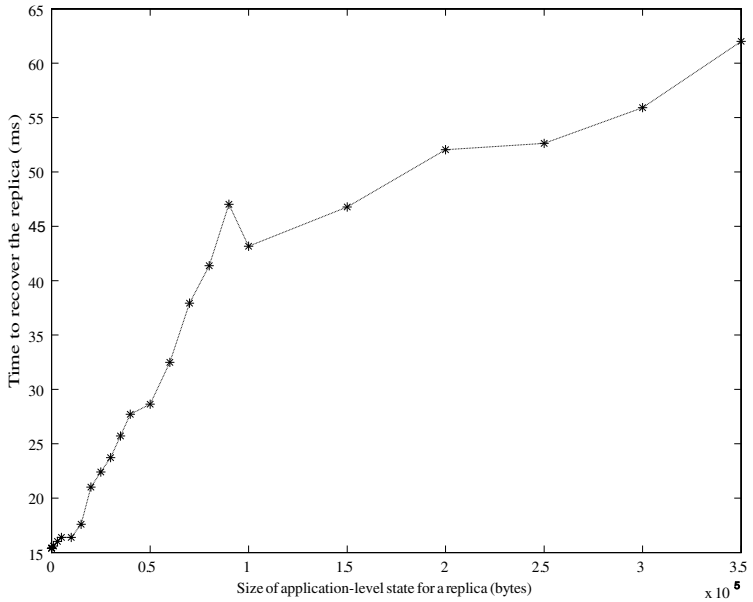

Figure 7. Variation of the recovery time for a server replica with the size of the replica's application-level state. The recovery time includes the time to recover all the three kinds of state; however, the ORB-level and the infrastructure-level states are fixed in size.

6. RELATED WORK

Other systems have been developed that address issues related to consistent object replication and fault tolerance in the context of CORBA. The Object Group Service (OGS) [6] provides replication for CORBA applications through a set of CORBA services. Replica consistency is ensured through group communication based on a consensus algorithm implemented through CORBA service objects. OGS provides interfaces for detecting the liveness of objects and mechanisms for duplicate detection and suppression and for the transfer of application-level state.

Aspect-Oriented Programming (AOP) allows for a separation of concerns at various levels, and can be applied to configure fault-tolerant CORBA systems [7]. Using AOP, fault tolerance, timing and consensus may be regarded as aspects of a reliability framework. The programmer specifies these aspects at design time; at configuration time, the aspect information and any other design-time information are converted into runtime configuration information which is stored in a reflector component. The reflector component can then be deployed at runtime to enforce specific fault tolerance quality-of-service policies, e.g. majority voting, on the CORBA application.

Newtop is a group communication toolkit that is exploited to provide fault tolerance to CORBA using the service approach. While the fundamental ideas are similar to OGS, the Newtop-based object group service [8] has some key differences. Of particular interest is the way this 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. No mechanisms are provided, however, to ensure consistent remerging of the sub-groups once communication is re-established.

The Maestro toolkit [9] includes an IIOP-conformant ORB with an open architecture that supports multiple execution styles and request processing policies. The replicated updates execution style can be used to add reliability and high availability properties to client/server CORBA applications in settings where it is not feasible to make modifications at the client side, as is the case for unreplicated clients wishing to contact replicated objects.

The AQuA architecture [10] is a dependability framework that provides object replication and fault tolerance for CORBA applications. AQuA exploits the group communication facilities and the ordering guarantees of the underlying Ensemble and Maestro toolkits to ensure the consistency of the replicated CORBA objects. AQuA supports both active and passive replication, with state transfers to synchronize the states of the backup replicas with the state of the primary replica in the case of passive replication.

The Distributed Object-Oriented Reliable Service (DOORS) [11] provides fault tolerance through a service approach, with CORBA objects that detect, and recover from, replica and processor faults. The system provides support for resource management based on the needs of the CORBA application. DOORS employs libraries for the transparent checkpointing [12] of applications; however, duplicate detection and suppression are not addressed.

The Interoperable Replication Logic (IRL) [13] also provides fault tolerance for CORBA applications through a service approach. One of the aims of IRL is to uphold CORBA's interoperability by supporting a fault-tolerant CORBA application that is composed of objects running over implementations of CORBA from different ORB vendors[‖].

7. CONCLUSION

The Eternal system provides transparent OMG-compliant component-based fault tolerance for CORBA enterprise applications, requiring no modifications to either the application or the CORBA middleware. Eternal's components and mechanisms interact to provide strong replica consistency for every replicated object of the CORBA application. Recognizing that real-world applications are necessarily multi-tiered (i.e. containing objects that play the roles of both the client and the server), Eternal supports the replication of both client and server objects.

The types of faults tolerated by Eternal system are communication faults, including message loss and network partitioning, as well as crash faults, including processor, process and object faults. Eternal can also tolerate arbitrary faults by employing active replication with majority voting for each application

[‖] In the interests of strong replica consistency, the Fault-Tolerant CORBA standard requires that all of the replicas of a replicated object be hosted by the same ORB (i.e. an implementation of CORBA from the same ORB vendor). Without this restriction, given the current state of commercial ORBs, each replica's ORB would itself constitute a source of non-determinism. However, the Fault-Tolerant CORBA standard and the Eternal system uphold CORBA's interoperability by allowing for the interaction of *different* replicated objects across *different* ORBs, as long as *each* replicated object has all of its replicas running over the *same* ORB.

object, along with group communication protocols that are equipped to detect and handle malicious behavior within the system.

Eternal's transparency reduces the time to develop and market a new fault-tolerant application, requires no retraining of application programmers and enables existing applications to be made fault tolerant. With shorter time-to-market and higher reliability being critical to e-commerce and e-business enterprise applications, these applications can be provided the fault tolerance that they need more quickly, more affordably and more reliably using Eternal.

ACKNOWLEDGEMENTS

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.

REFERENCES

1. IBM Global Services. Improving systems availability. *White Paper*, 1998.
2. Object Management Group. The Common Object Request Broker: Architecture and specification, 2.3 edition. *OMG Technical Committee Document formal/98-12-01*, June 1999.
3. Moser LE, Melliar-Smith PM, Narasimhan P. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems* 1998; **4**(2):81–92.
4. Object Management Group. Fault tolerant CORBA (final adopted specification). *OMG Technical Committee Document ptc/2000-0404*, March 2000.
5. Moser LE, Melliar-Smith PM, Agarwal DA, Budhia RK, Lingley-Papadopoulos CA. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM* 1996; **39**(4):54–63.
6. Felber P, Guerraoui R, Schiper A. The implementation of a CORBA object group service. *Theory and Practice of Object Systems* 1998; **4**(2):93–105.
7. Polze A, Schwarz J, Malek M. Automatic generation of fault-tolerant CORBA services. *Proceedings 34th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, August 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 205–213.
8. Morgan G, Shrivastava S, Ezhilchelvan P, Little M. Design and implementation of a CORBA fault-tolerant object group service. *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999. Kluwer Academic Publishers, 1999; 361–374.
9. Vaysburd A, Birman K. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems* 1998; **4**(2):73–80.
10. Cukier M, Ren J, Sabnis C, Sanders WH, Bakken DE, Berman ME, Karr DA, Schantz R. AQuA: An adaptive architecture that provides dependable distributed objects. *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, West Lafayette, IN, October 1998; 245–253.
11. Natarajan B, Gokhale A, Yajnik S, Schmidt DC. DOORS: Towards high-performance fault-tolerant CORBA. *Proceedings International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, September 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 39–48.
12. Wang YM, Huang Y, Vo KP, Chung PY, Kintala CMR. Checkpointing and its applications. *Proceedings 25th IEEE International Symposium on Fault-Tolerant Computing*, Pasadena, CA, June 1995. IEEE Computer Society: Los Alamitos, CA, 1995; 22–31.
13. Marchetti C, Mecella M, Virgillito A, Baldoni R. An interoperable replication logic for CORBA systems. *Proceedings International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, September 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 7–16.

