Strongly Consistent Replication and Recovery of Fault-Tolerant CORBA Applications

P. Narasimhan*
Institute of Software Research International
School of Computer Science
Carnegie-Mellon University, Pittsburgh, PA 15213-3891
priya@cs.cmu.edu

L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system provides transparent fault tolerance for CORBA applications, without requiring the modification of either the application or the ORB. Eternal replicates the application objects, and ensures strong replica consistency by employing a reliable totally-ordered multicast protocol for conveying the IIOP messages of the application. To achieve strong replica consistency during recovery, Eternal retrieves and transfers the three kinds of state – application-level state, ORB/POA-level state and infrastructure-level state – from an existing replica to a new or recovering replica, and logs and replays messages.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [13] supports applications that consist of objects distributed across a system, with client objects invoking server objects that 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, transcending differences in their programming language (language transparency) and their physical locations (location transparency). CORBA's TCP/IP-based Internet Inter-ORB Protocol (IIOP) allows a client object and a server object to communicate, regardless of differences in their respective operating systems, byte orders and hardware architectures.

Enhancing CORBA with fault tolerance while maintaining CORBA's transparency and simplicity of application programming is a challenge. The Eternal system [8] addresses this challenge 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 CORBA application code nor the ORB needs to be modified to benefit from the fault tolerance that Eternal

*Work performed while the author was at the University of California, Santa Barbara.

provides. 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.

The Eternal system 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 service, even if some of its replicas fail, or the processors hosting some of its replicas fail. The Eternal system maintains *strong replica consistency*, even as objects and the processors that host them fail, even as object receive and process invocations, and even as objects are created and destroyed.

2 Strong Replica Consistency

Strong replica consistency ensures that all of the replicas of an object have the same state and exhibit the same behavior, and is important for simplifying the programming of fault tolerant applications. In the absence of strong replica consistency, during normal operation, the "replicas" might exhibit different or inconsistent behavior and, after a fault, a new replica might exhibit different or inconsistent behavior from the old replica before the fault. Handling such differences and inconsistencies imposes a substantial burden on the application programmer.

To ensure strong replica consistency, the application objects must be *deterministic* 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.

In cases where the application objects are inherently non-deterministic, *e.g.*, the application uses system-specific or processor-specific functions, Eternal provides mechanisms to identify and "sanitize" sources of non-determinism, thereby giving the application the appearance of being deterministic. To maintain strong replica consistency, Eternal addresses a number of issues, including:

- **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 the Totem reliable totally-ordered multicast protocol [7] for conveying the IIOP invocations (responses) to the replicas of a CORBA server (client), thereby facilitating replica consistency under both fault-free and recovery conditions.

- **Duplicate operations.** Replication, by its very nature, can 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 invoking client replicas. Eternal provides unique invocation (response) identifiers to ensure that such duplicate invocations (responses) from a replicated client (server) are never delivered to their target server (client) objects.

- **Multithreading.** Many commercial ORBs and CORBA applications employ multithreading, a significant source of non-deterministic behavior. Replicas of a multithreaded object might become inconsistent if the threads, and the operations that they execute, are not carefully controlled. Section 6.1 describes the mechanisms that Eternal provides to ensure strong replica consistency, in the face of multithreading in the ORB or in the application.

- **Recovery.** Replicating an object allows it to provide useful services even if one of its replicas fails. For true fault tolerance, it must be possible to recover a failed replica, and to reinstate it

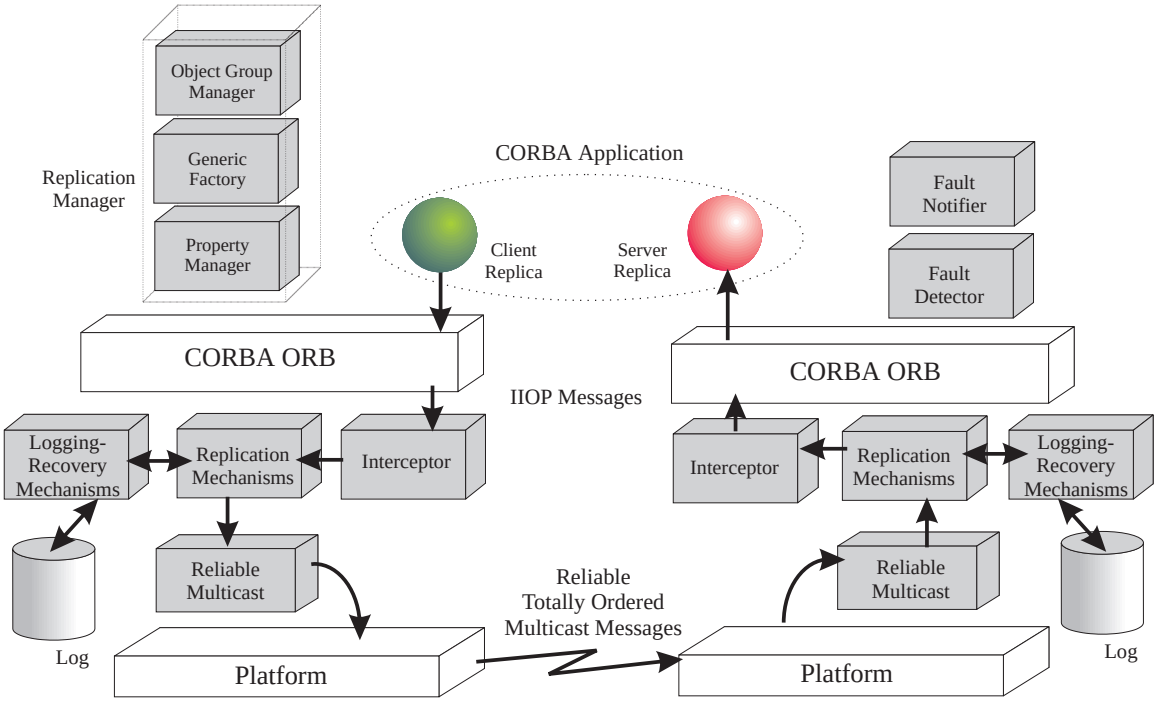


Figure 1: Eternal's OMG-compliant fault tolerance infrastructure, consisting of components above and below the ORB.

to be useful again. However, *before* a new or recovered replica issues an invocation, performs an operation, or returns a response, its state must be synchronized with that of the other operational replicas of the object. Eternal provides state transfer mechanisms that retrieve the state from an existing operational replica of the object, and transfer the state to a new or recovering replica before making it operational.

3 Architecture of the Eternal System

The Eternal system consists of CORBA objects *above* the ORB and mechanisms *below* the ORB, as shown in Figure 1. The objects above the ORB support the interfaces of the Object Management Group's Fault-Tolerant CORBA standard [14], while the underlying mechanisms (components below the ORB) provide the fault tolerance that is necessary for these interfaces to operate correctly. Although these fault tolerance mechanisms can be located above the ORB, they are implemented underneath the ORB in the Eternal System, in the interests of both performance and transparency.

The Replication Manager replicates each application object, according to user-specified requirements, and distributes the replicas across the system. The Fault Detector detects the failure of replicas, objects and processors in the system. The Fault Notifier uses the information gathered by the Fault Detector to notify other objects of faults that have occurred in the system. The Replication Manager, the Fault Detector and the Fault Notifier themselves are CORBA objects; thus, they too are replicated for fault tolerance.

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 transmission via the Totem reliable totally-ordered multicast protocol [7].

The Replication Mechanisms filter duplicate operations, and control the dispatching of operations to the multiple threads of an object. The Logging-Recovery Mechanisms are responsible for the logging of messages and checkpoints, as well as for the retrieval, transfer and assignment of state. The Replication and Logging-Recovery Mechanisms, underneath the ORB, consist of non-CORBA C++ objects, and are present on every processor that is to host replicated objects.

Some applications may contain objects that need not, or cannot, be replicated, $e.g.$, graphical user interfaces that serve as pure clients. To handle such applications, the Eternal Gateway component [11], shown in Figure 2, allows an unreplicated client object to communicate with replicated server objects. The Gateway provides the translation between the client's TCP/IP-based messages and the server's reliable multicast messages both of which encapsulate IIOP data. In addition, the Gateway filters the duplicate responses from the replicated server so that the client benefits from the server's fault tolerance, without being aware of the server's replication.

For the CORBA applications that it supports, the Eternal system tolerates the following kinds of faults at the processor, process and object levels:

- Crash (fail-stop) fault: when a processor/process/object halts prematurely, but was operating correctly until it stopped,
- Omission fault: when a processor/process/object fails to respond to a request,
- Timing fault: when the response from a processor/process/object occurs outside a specified time interval, and
- Communication fault: when the message sent by a processor/process/object is lost on the communication medium.

By exploiting protocols with more stringent guarantees than the Totem System provides, Eternal can also tolerate arbitrary (also known as commission or Byzantine) faults [9], where a processor/process/object is subverted, resulting in malicious behavior.

4 Replication Management

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 handles the creation, deletion and replication of both the application objects and the infrastructure objects. The Replication Manager replicates objects, and distributes the replicas across the system. Although each replica of an object has an individual object reference, the Replication Manager 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, the Generic Factory and the Object Group Manager, as shown in Figure 2.

The Property Manager allows the user to assign values to 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 the various replication styles is described in further depth in Section 7.

- **Membership Style** – addition, or removal, of an object's replicas 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.
- **Initial Number of Replicas** – the number of replicas of an 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.

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

The Object Group Manager allows users to control directly the creation, deletion and location of individual replicas of an application object. While use of the Object Group Manager violates replication transparency (because the user is explicitly aware of the replicas of an object) and might violate strong replica consistency (unless proper care is exercised), it is useful for expert users who wish to exercise direct control over the replication of application objects.

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 and because it assures strong replica consistency, which the Eternal infrastructure maintains under both fault-free and recovery conditions.

5 Fault Detection and Notification

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

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

The Eternal infrastructure 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, alternatively, 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.

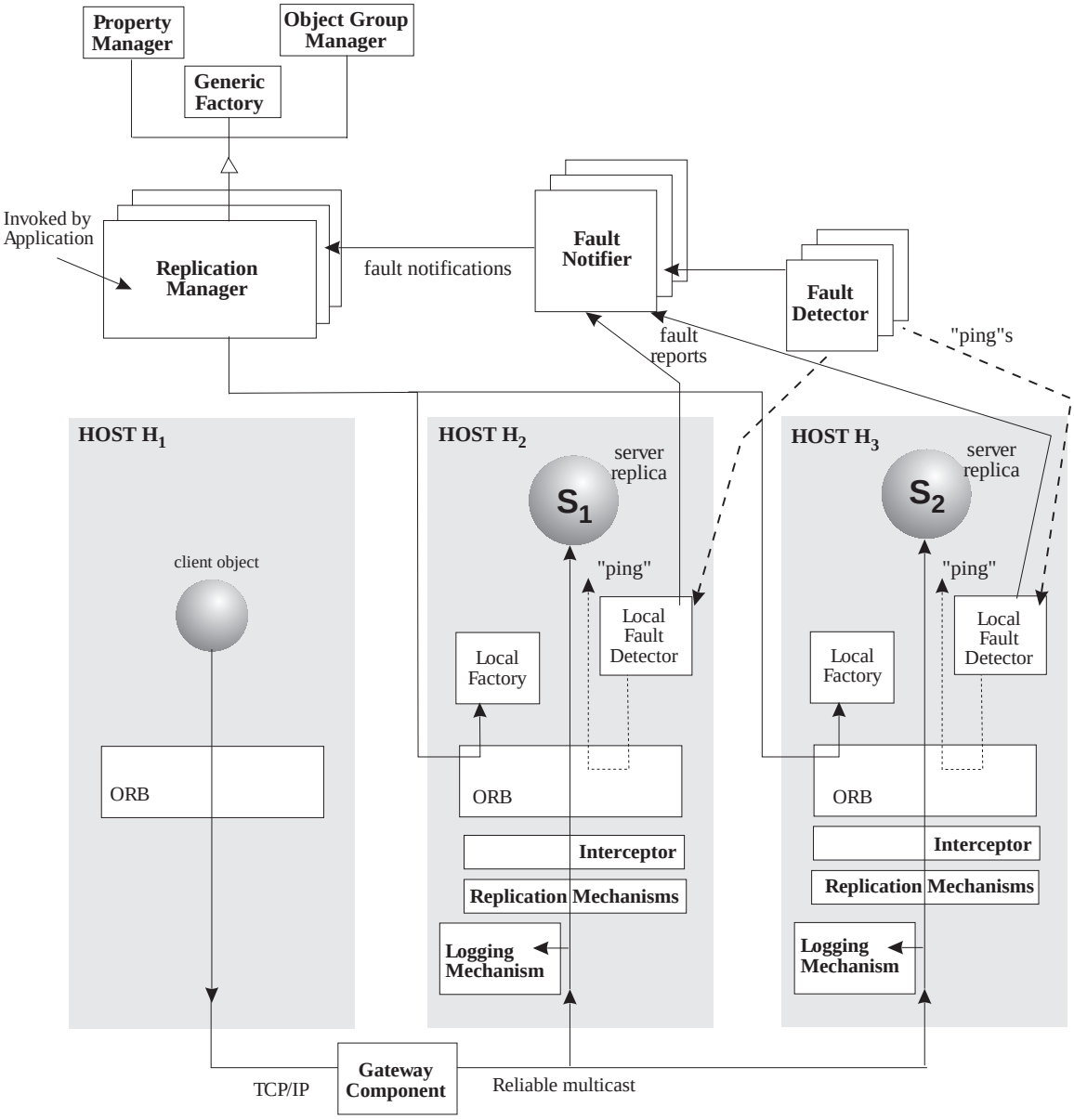


Figure 2: Interaction between Eternal's components below the ORB and above the ORB, for an unreplicated client replica that communicates with two replicas of an actively replicated server.

- **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 2, the Fault Detectors can be structured hierarchically, with the global replicated Fault Detector triggering the operation of local fault detectors on each processor. Any faults detected by the local fault detectors are reported to the global replicated Fault Notifier. The Replication Manager, being a subscriber of the Fault Notifier, receives reports of faults that occur in the system, and can initiate appropriate actions to enable the system to recover from faults.

6 Interception

The Eternal Interceptor is a non-ORB-level, non-application-level component that transparently "attaches" itself to every executing CORBA object, without the knowledge of the object or the ORB, and that can modify the object's behavior as desired. The advantage of Eternal's Interceptor, located below the ORB, is not only its transparency to the ORB and to the application, but also its implementation in an ORB-independent and application-independent manner. In fact, because every implementation of CORBA must support the TCP/IP-based IIOP interface, by focussing on intercepting the ORB's TCP/IP interface, the Eternal Interceptor can attach itself readily to any ORB, without requiring any additional ORB-specific or application-specific code. This ORB independence and application independence has allowed Eternal to be used with many different ORBs without modification of, or even access to, the code of the ORB or the application.

While the interception approach allows the easy insertion of fault tolerance mechanisms into a CORBA application, our current interception mechanisms is dependent on operating system-specific hooks. Thus, while the implementation of the interceptor does not need to be ported across different ORBs, it must be ported to run on different operating systems. Fortunately, similar (and often, several different kinds of) interception hooks exist on almost every operating system, primarily because operating systems themselves use these hooks for the run-time monitoring and profiling of the processes that they support.

With the Unix operating system, there exist at least two possible approaches to implementing interceptors. The first of these approaches, the $/proc$-based implementation, provides for interception at the level of system calls. The second approach, the library interpositioning implementation (also possible on Windows NT), provides for interception at the level of library routines. While the techniques differ, the intent and the use of interceptors in both cases are identical, and require no modification of the intercepted CORBA objects, the ORB or the operating system.

The specific system calls to redefine in a $/proc$-based implementation, or the specific library routines to redefine in a library-interpositioning implementation, depend on the extent of the information that the interceptor must extract 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.

Eternal's Interceptor currently employs the library interpositioning approach, because of its lower overheads and ease of deployment with various ORBs. The system calls, or library routines, of interest to Eternal are those associated with the communication of CORBA's IIOP messages. Because the ORB conveys the IIOP messages over TCP/IP, Eternal's Interceptor captures, and redefines, the routines related to TCP/IP communication. The redefinition of these routines allows the Interceptor to divert the captured IIOP messages to the Eternal Replication Mechanisms.

The Eternal Interceptor can be extended to perform other useful functions, including the profiling of IIOP messages and the debugging of the attached CORBA applications. Of particular importance, from the viewpoint of consistent replication, is the fact that the Interceptor can be exploited to sanitize the non-determinism exhibited by multithreaded ORBs and/or multithreaded CORBA applications.

6.1 Consistency under Multithreading

Many commercial ORBs are multithreaded, and multithreading can yield substantial performance advantages. Unfortunately, the specification of multithreading in the CORBA standard does not place any guarantee on the order of operations dispatched by a multithreaded ORB. In particular, the

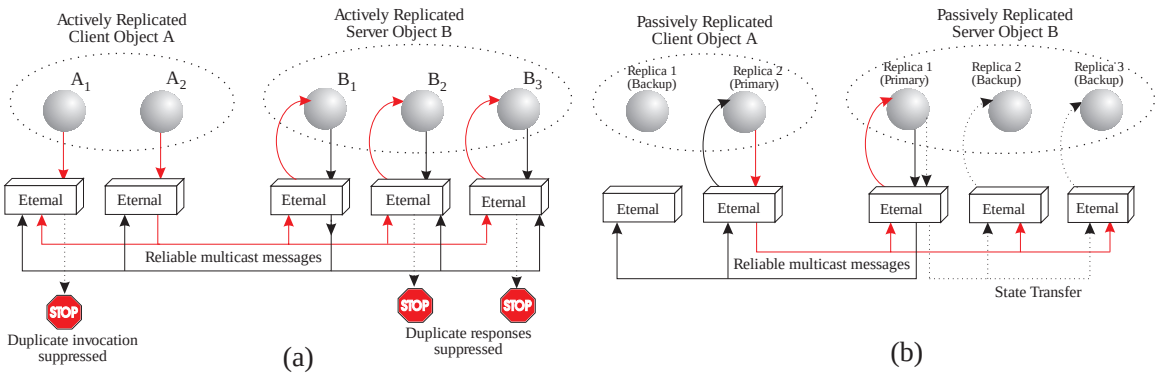


Figure 3: Two of the Replication Styles supported by the Eternal infrastructure: (a) active replication, and (b) warm passive replication.

specification of the Portable Object Adapter (POA) provides no guarantees on the POA's dispatching of requests to threads. The ORB/POA may dispatch several requests for the same object within multiple threads at the same time.

In addition to ORB-level threads, the CORBA application may itself be multithreaded, with the thread scheduling being determined by the application programmer. The application programmer must ensure correct sequencing of operations and must prevent thread hazards. Careful application programming can ensure thread-safe operations within a single replica of an object; however, it does not guarantee that threads and operations are dispatched in the same order across all of the replicas of the object.

To preserve replica consistency for multithreaded objects, the Eternal infrastructure 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 operation scheduler [10] that is controlled jointly by the Interceptor and the Replication Mechanisms.

The scheduler 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 can transparently override any thread or operation scheduling performed by either the nondeterministic multithreaded ORB, or by the replica itself.

Based on the incoming totally ordered sequence of messages that arrive at the Interceptor through the Replication Mechanisms, the scheduler at each replica decides on the immediate delivery, or the delayed delivery, of the messages to that replica. At each replica, the scheduler's decisions are identical and, thus, operations and threads are dispatched identically at each replica, ensuring determinstic operation across all replicas of the object.

7 Replication Styles

7.1 Active Replication

For *active* replication, shown in Figure 3(a), each replica processes (invokes) every operation. The failure of a single active replica is masked by the presence of the other active replicas that also perform the operation and generate the desired result.

For an actively replicated server (client) object, the Replication Mechanisms deliver every IIOP invocation (response) intended for the replicated server (client) to every server (client) replica

through the Interceptor. Thus, Eternal ensures that all of the replicas of an actively replicated object have the same state at the end of each operation.

Active replication may lead to duplicate operations because every replica of an actively replicated object sends the invocation or response. Duplicate operations must not be allowed to be executed on their target objects because they will corrupt the state of the target objects. Eternal provides mechanisms for detecting, and suppressing, duplicate operations so that an operation is never executed more than once on the target object.

Before allowing a new or recovering active replica to become operational, Eternal initializes its state using the state of an existing active replica of the object. However, because such a recovering active replica may continue to receive invocations and responses during this state transfer, Eternal enqueues the invocations and responses that arrive during the state transfer, and applies them to the recovered replica subsequent to the state transfer. Note that this enqueueing of messages occurs only at the recovering active replica; Eternal allows the remaining (operational) active replicas to proceed normally with the incoming sequence of invocations and responses.

7.2 Passive Replication

For *passive* replication, only one of the replicas, designated the *primary*, processes (invokes) the operations. With *warm passive* replication, shown in Figure 3(b), the remaining passive replicas, known as *backups*, are preloaded into memory and are synchronized periodically with the primary replica, so that one of them can take over should the primary replica fail. Eternal thus ensures that all of the replicas of a warm passively replicated object have the same state at the end of each periodic state transfer. With *cold passive* replication, however, the backup replicas are not even loaded into memory as long as the primary replica continues to operate. To allow for recovery, the state of the primary replica is periodically captured and stored into a log. If the existing primary replica fails, a backup replica is loaded into memory, has its state initialized from this log, and then becomes the new primary.

For a passively replicated server (client) object, the Replication Mechanisms deliver every IIOP invocation (response) only to the primary replica of the passively replicated server (client) object. The request (reply) messages are, however, logged at every backup replica (see Section 8.2) so that those messages are available to initialize the state of that replica, should it ever become the primary.

If the primary replica fails, one of the backup replicas is chosen to be the new primary and, as such, must be restored to the state that the old primary possessed immediately before it failed. Because the old primary replica is no longer available once it has failed, the state of the primary (while it is operational) must be continuously or periodically captured and stored so that it is available for recovery if the primary replica fails. While a backup replica is being recovered to take over as the new primary, it may continue to receive invocations and responses. Eternal enqueues the invocations and responses that arrive during the recovery period, and applies them to the new primary replica subsequent to recovery.

7.3 Comparison of Replication Styles

It might appear that substantially different mechanisms are required for the different replication styles. For example, duplicate detection might appear to be necessary only for active replication. However, if the primary replica of an object partially processes an operation and then fails, the backup replica that becomes the new primary might need to reprocess the operation. As a consequence of that reprocessing, the new primary will invoke operations that might already been invoked

by the old primary. Processing those requests twice would be an error. Thus, duplicate detection is required for passive replication during fault recovery. Similarly, state synchronization is required, for active replication, when a failed active replica is repaired, and is re-introduced into the system. Indeed, the mechanisms required to support active and passive replication are substantially equivalent, although they are used under different circumstances.

The Eternal System allows the user to select the replication style that would be most appropriate for the objects of his/her CORBA application. Some of the criteria for choosing between active and passive replication for a CORBA object include the following:

- **Cost of checkpointing.** 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.

- **Computational resources.** 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.

- **Bandwidth.** 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.

- **Speed of recovery.** 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 re-election 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 $vs.$ passive replication is also dictated by other 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 [4] have been considered, with the aim of addressing the reduction of multicast overhead in active replication styles, as well as of achieving the best of the active and passive replication styles.

8 Recovery

This section describes the logging, checkpointing and recovery mechanisms of the Eternal System that ensure strong replica consistency for both active and passive replication styles, when a new or recovered replica is launched.

8.1 Consistent State

Every replicated CORBA object can be regarded as having three kinds of state: *application-level state*, *ORB/POA-level state*, and *infrastructure-level state*. Any fault-tolerant CORBA infrastructure that aims to provide strong replica consistency must maintain consistent application-level, ORB/POA-level and infrastructure-level state across all of the replicas of every replicated CORBA object. Maintaining consistent state is a challenging problem, particularly during recovery.

8.1.1 Application-Level State

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

To enable application-specific state to be captured, in accordance with the Fault-Tolerant CORBA standard, every replicated CORBA object must inherit the OMG-IDL `Checkpointable` interface, shown in Figure 4.

This inherited IDL interface has two methods, $get\_state()$ and $set\_state()$. The $get\_state()$ method, when invoked on a CORBA object, returns the current application-level state of the object. The $set\_state()$ method with specific state as its parameter, when invoked on a CORBA object, overwrites the object's current application-level state with the value of this parameter.

Because it is not possible to anticipate, or to standardize on, the format of the application-level state of every conceivable application object, the application-level state is defined to be of the CORBA type `sequence<octet>`. A variable of type `sequence<octet>` can "hold" any primitive, structured or user-defined CORBA type.

8.1.2 ORB/POA-Level State

Ideally, an ORB should be viewable as a "black-box" that is stateless. In reality, because the ORB and the Portable Object Adapter (POA) handle all connection-level and transport-level information on behalf of a CORBA object that they support, the ORB and the POA necessarily maintain some information about the object. The existence of such ORB/POA-level state implies that there really are no "stateless" objects; a replicated CORBA object with no application-level state will nevertheless have associated ORB/POA-level state. That ORB/POA-level state is modified as the ORB creates objects, establishes connections and processes incoming messages.

The ORB/POA-level state for a CORBA object consists of the values of various data structures (last-seen request identifier, threading policy, *etc.*) stored by the ORB, at runtime, on behalf of

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

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

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

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

Figure 4: The `Checkpointable` IDL interface that must be inherited by every CORBA application object to enable the checkpointing and transfer of application-level state.

the object. Unfortunately, these "pieces" of ORB/POA-level state are not visible at the application level. The internal ORB/POA-level state is not standardized and, thus, not identical across ORBs from different vendors. Thus, a strongly consistent replicated object must have all of its replicas running over ORBs from the same ORB vendor.

8.1.3 Infrastructure-Level State

Infrastructure-level state* is completely independent of, and invisible to, the replicated object as well as to the ORB and the POA, and involves information that the Eternal infrastructure needs for maintaining consistent replication. The infrastructure-level state contains information that is essential for duplicate detection and for garbage collection of the log.

For every operational replica that it hosts, Eternal's Recovery Mechanisms (running on the same processor as the replica) store information locally regarding:

- The invocations that the replica has issued, and for which the replica is awaiting responses
- The invocations and responses that have been enqueued (while the replica is not quiescent) for delivery to the replica when it becomes quiescent
- The replication style of the replica, including whether it is an active, warm passive primary, warm passive backup, cold passive primary or cold passive backup replica
- The Eternal-generated operation identifiers that enable the Logging-Recovery Mechanisms to filter duplicate invocations and responses intended for the replica.

*Infrastructure-level state is not unique to the Eternal system. In fact, any system that provides fault tolerance must maintain *some* state on behalf of the replicated objects that it hosts.

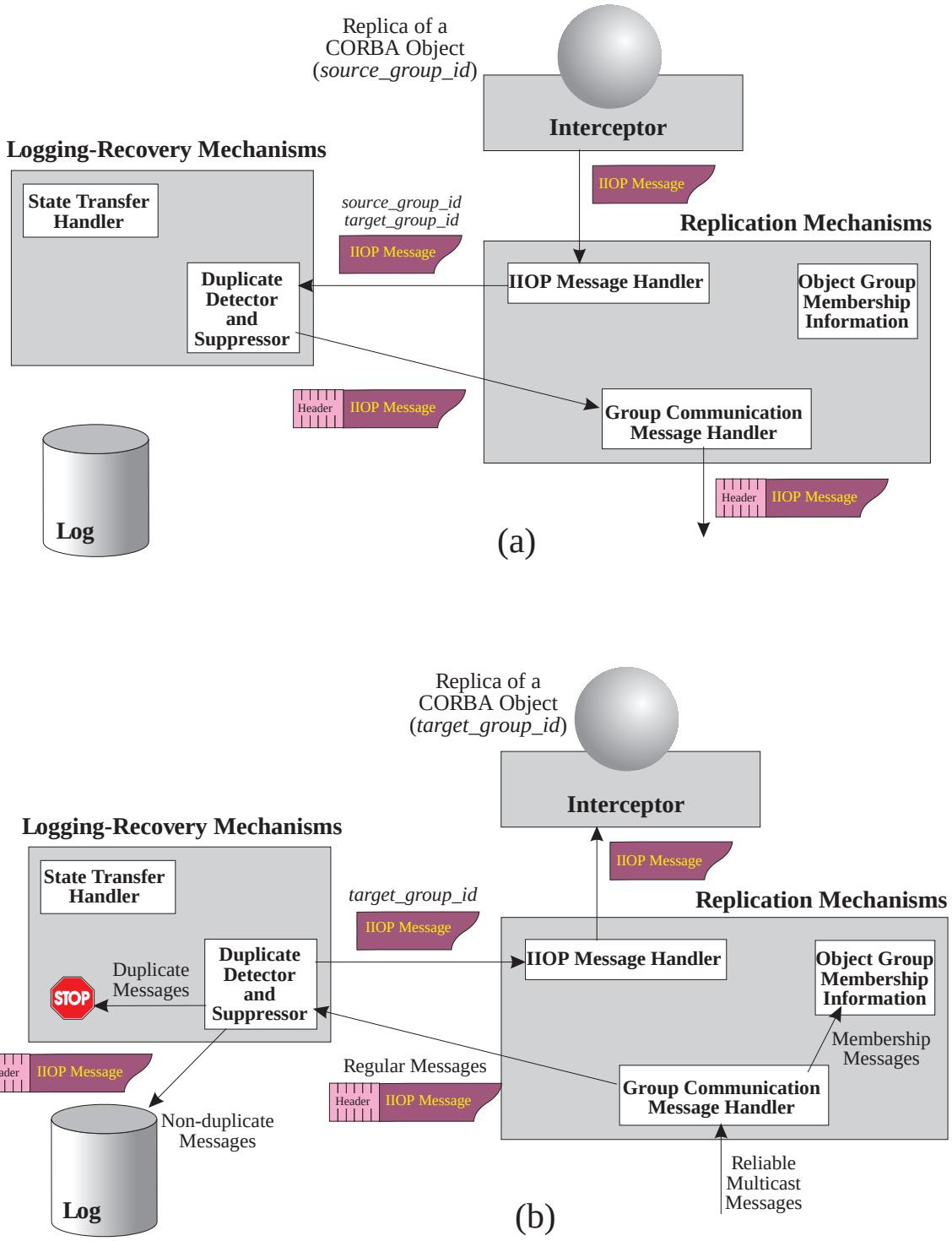


Figure 5: Interaction between Eternal's Replication Mechanisms and Logging-Recovery Mechanisms for (a) outgoing messages and (b) incoming messages.

8.2 Logging-Recovery Mechanisms

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. State transfer to a new or recovering replica includes the transfer of application-level state to the new replica, ORB-level state to the ORB hosting the new replica, and infrastructure-level state to the Logging-Recovery Mechanisms that manage the new replica.

The Logging-Recovery Mechanisms invoke the `get_state()` method of the `Checkpointable` interface to retrieve, or checkpoint, the application object's state. For active replication, there is no need to log any checkpoints or messages until a replica is being recovered. At that point, Eternal's mechanisms for synchronizing state transfer handle the retrieval of checkpoints and the logging of messages, just as for passive replication. For warm passive replication, the Logging-Recovery Mechanisms checkpoint the primary replica's state periodically, and transfer the checkpointed state to the backup replicas. For cold passive replication, the Logging-Recovery Mechanisms store the checkpointed state of the primary replica into a log for restoring the state of a new primary, if the existing primary fails. The frequency of checkpointing is determined on a per-group basis, by the user at deployment time, when the other fault tolerance properties (number of replicas, location of replicas, etc) are also assigned their values.

Because the state retrieval from an existing replica occurs at a different point in the message sequence from the assignment of the retrieved state to the new replica, the Logging-Recovery Mechanisms at the state retrieval and assignment locations must synchronize the state retrieval and assignment messages. Furthermore, the Logging-Recovery Mechanisms must enqueue all new invocations and responses that arrive for a replica while its state is being assigned.

The Logging-Recovery Mechanisms on a processor are responsible for storing the invocations, responses and state checkpoints of the replicas hosted on that processor. Typically, replicas of several different objects (*i.e.*, members of several different object groups) exist on a processor and, thus, the Logging-Recovery Mechanisms maintain a single physical log per processor, with the log being indexed by the object group identifier. Each entry in the log is a log record that contains a received IIOP message along with a special Eternal-specific header associated with the IIOP message for duplicate detection, garbage collection of the log, etc. The records are stored in the log as they arrive at the Logging-Recovery Mechanisms from the Replication Mechanisms, which, in turn, receives them from the totally-ordered message sequence that the underlying Totem multicast protocol provides.

Because the Logging-Recovery Mechanisms have access to the log, they are in a position to match up responses with their corresponding invocations and to detect and suppress duplicate invocations, responses and state transfer messages. To enable incoming response messages to be matched with their corresponding invocations, the Logging-Recovery Mechanisms insert an invocation (response) identifier into the Eternal-specific header of each outgoing IIOP invocation (response) message, as shown in Figure 5(a). For an outgoing response, the Logging-Recovery Mechanisms "remember" and reuse a portion of the invocation identifier associated with the invocation that resulted in this response. The portion of the invocation identifier that is reused in its counterpart response identifier is the *operation identifier*, which represents the operation consisting of the invocation-response pair.

By exploiting the totally-ordered message sequence numbers that Totem assigns to each message that it delivers, the Logging-Recovery Mechanisms on *different* processors ensure that they always assign the same unique operation identifier for each distinct operation. Also, by performing the duplicate detection and suppression on every incoming message that they receive, as shown in

Figure 5(b), the Logging-Recovery Mechanisms ensure that the target application objects receive only one copy of every distinct invocation or response intended for them. Furthermore, the Logging-Recovery Mechanisms perform the duplicate detection *before* recording messages in the log; thus, the log is a sequence of non-duplicate log records.

During recovery, the Logging-Recovery Mechanisms hosting an existing replica "piggyback" the infrastructure-level state for the replica onto the application-level state and the ORB/POA-level state that they transfer to the Logging-Recovery Mechanisms hosting the new replica. The Logging-Recovery Mechanisms that receive the three kinds of state assign the application-level state, the ORB/POA-level state and the infrastructure-level state *before* allowing the new replica to become fully operational, and to receive or process any normal incoming invocations or responses. The retrieval, as well as the assignment, of the three different kinds of state appears as a single atomic action so that the state transfer of the three kinds of state occurs at a single logical point in time.

8.3 The Need for Quiescence

By no means does the checkpointing frequency guarantee that the replicated object will perform the state retrieval (via a $get\_state()$ operation) immediately. The replicated object may be in the middle of another operation, or may be blocked waiting for a response, *etc*. To decide on the appropriate time to deliver the $get\_state()$ invocation, the Eternal system must determine the moment that the object is quiescent, *i.e.*, when it is "safe," from the viewpoint of strong replica consistency, to deliver a new invocation to the object. Determining whether an object is quiescent is a non-trivial matter – it involves examining the status of current invocations on the object, the threads that are currently executing within the process containing the object, and data that the object may share with other in-process collocated objects. The use of `oneway` operations (CORBA invocations that do not return responses) introduces additional complications for quiescence.

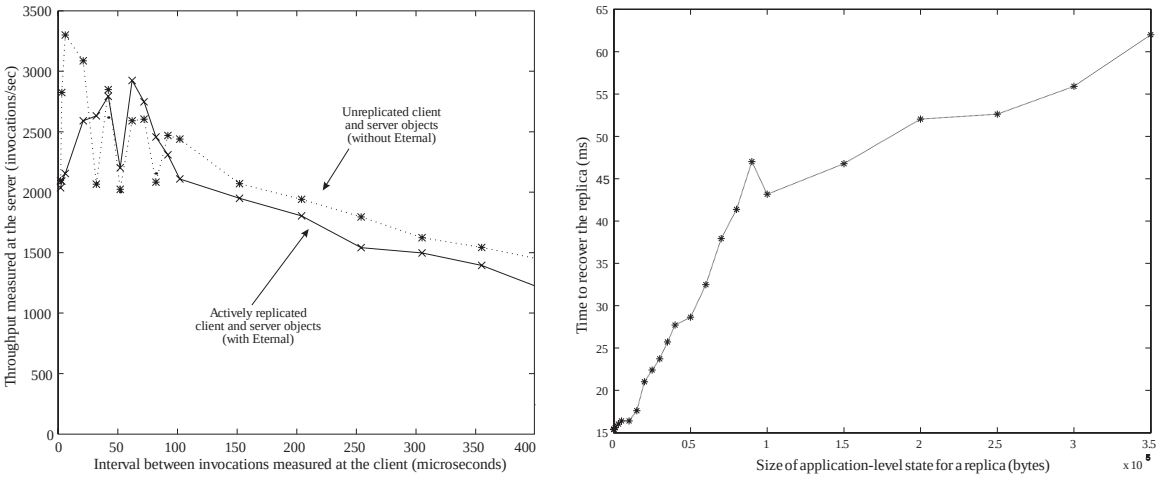
9 Implementation and Performance

The Eternal system provides support for the replication and recovery of unmodified CORBA objects running over unmodified commercial ORBs, including Inprise's VisiBroker, Iona Technologies' Orbix, Vertel's e*ORB, Object-Oriented Concepts' ORBacus, Washington University, St. Louis' TAO, AT & T Laboratories' omniORB2, Expersoft's CORBAplus and Xerox PARC's ILU.

Using Eternal, for Solaris 2.x on 167Mhz SPARC workstations connected by a 100Mbps Ethernet, when application objects are actively replicated, test applications typically incur only a 10-15% increase in round-trip invocation/response time, compared with their unreplicated unreliable counterparts. For RedHatLinux 6.0 on 400Mhz Intel Pentium processors connected by a 100Mbps Ethernet, this overhead reduces to 3%.

To measure the performance of Eternal, we used a simple test application developed with the VisiBroker 3.2 C++ 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 graph in Figure 6(a) shows the throughput obtained for the three-way active replication of both the client and the server objects using Eternal with 1 Kbyte messages, as compared with the throughput obtained for the unreplicated client and server objects without Eternal.

With a very short busy-waiting delay between reception of a reply by the client and issue of the next request, instability in the throughput is visible. This is caused by the scheduling algorithms of



(a) (b)

Figure 6: (a) Throughputs obtained for an unreplicated application (without Eternal) and its three-way actively replicated counterpart (with Eternal), and (b) Variation of the recovery time for a server replica with the size of the replica's application-level state.

Solaris. If the client loses control during the busy-waiting delay, additional context switching delay is incurred and a lower throughput is measured. For delays greater than 100 microseconds, stable throughput measurements are obtained, with a reduction in throughput of about 10%.

We also measured the performance of the Eternal system during the recovery of a failed replica of an object. The graph in Figure 6(b) shows the time to recover a server replica in a test application developed with Inprise's VisiBroker 4.0 C++ ORB. 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.

During the experiments, one or the other of the server replicas was killed and then re-launched. The time to recover such a failed replica was measured as the time interval between the re-launch of the failed replica and the replica's reinstatement to normal operation. The graph shows 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. The ORB/POA-level state and the infrastructure-level state are independent of, and therefore do not vary with, the size of the application-level state.

Regardless of the size of the application-level state, the entire application-level state is encapsulated in a single IIOP message by the ORB. However, at the transport layer of the reliable multicast protocol, the Ethernet medium fragments any message that is larger than the maximum Ethernet frame size (1518 bytes). This implies that large (> 1518 bytes) IIOP messages will be transmitted over the Ethernet by Eternal using multiple multicast messages. Thus, for any replicated object, the number of multicast messages required for state transfer, and therefore the time required for recovery, increases with the size of the object's application-level state, as seen in the graph.

Thus, in addition to the resource usage of an object, the size of the object's application-level state, and the requirements on the object's recovery time, also influence the choice of the object's replication style – active replication (more resource-intensive, fewer state transfers, faster recovery) *vs.* passive replication (less resource-intensive, more frequent state transfers, slower recovery).

10 Related Work

The Delta-4 system [16] aimed to provide fault tolerance in a distributed Unix environment, through the use of an atomic multicast protocol to tolerate crash faults at the process level. Delta-4 included support for active replication and passive replication, as well as hybrid semi-active replication. Backward error recovery is achieved by integrating checkpointing with inter-process communication.

The Arjuna system [15] uses object replication together with an atomic transaction strategy to provide fault tolerance. The types of replication supported include active replication, coordinator-cohort passive replication and single-copy passive replication. Strategies similar to checkpointing are used for disseminating state updates in passive replication.

The FRIENDS [2] system aims to provide mechanisms for building fault-tolerant applications in a flexible way through the use of libraries of meta-objects. Separate meta-objects can be provided for fault tolerance, security and group communication. FRIENDS is composed of a number of subsystems, including a fault tolerance subsystem that provides support for object replication and detection of faults. A number of interfaces similar to our `Checkpointable` interface are provided for capturing the state of an object to stable storage, and for transmitting the state of the primary replica to the backup replicas in the case of passive replication.

Other systems have been developed that address issues related to object replication and fault tolerance in the context of CORBA. The Object Group Service (OGS) [3] 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.

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 [6] 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 to ensure consistent remerging of the sub-groups once communication is reestablished.

The Maestro toolkit [17] 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 contacting replicated objects.

The AQuA architecture [1] 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 transfer 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) [12] 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 [18] of applications; however, duplicate detection and suppression are not addressed.

The Interoperable Replication Logic (IRL) [5] 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 ORBs from different vendors.

The features of the Eternal System that distinguish it from all of the other approaches to providing fault tolerance for CORBA are (i) the interception approach that allows an unmodified CORBA application running over an unmodified ORB to be enhanced with fault tolerance transparently, (ii) the emphasis on strong replica consistency through mechanisms for logging and recovery of ORB/POA-level state and infrastructure-level state, in addition to application-level state, and (iii) the support for mechanisms to overcome the non-determinism inherent in multithreaded CORBA applications and in the current implementations of the CORBA standard.

11 Conclusion

The Eternal system provides strongly consistent replication and recovery of unmodified CORBA client and server objects running over unmodified CORBA-compliant off-the-shelf ORBs.

The Replication Manager replicates each application object, according to user-specified requirements, and distributes the replicas across the system. The Fault Detector detects the failure of replicas, objects and processors in the system. The Fault Notifier uses the information gathered by the Fault Detector to notify other objects 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 transmission via the Totem reliable totally-ordered multicast protocol. The Totem protocol delivers the messages in the same order to all of the replicas of CORBA objects, which guarantees that the replicas perform the same operations in the same order.

The Replication Mechanisms filter duplicate operations, which ensures that the replicas do not perform the operations multiple times. They also control the dispatching of operations to the multiple threads of an object, which renders the behavior of an object deterministic. The Logging-Recovery Mechanisms are responsible for the logging of messages and checkpoints, as well as for retrieval, transfer and assignment of state. For every replicated CORBA object, they maintain the consistency of application-level, ORB/POA-level and infrastructure-level state, and ensure that the three kinds of state are synchronized across all of the replicas of a CORBA object.

Our experience with developing the Eternal System has shown us that, in order to provide strong replica consistency, a fault-tolerant CORBA system must deal with the intricacies of the ORB. This involves the challenging tasks of ascertaining the various "pieces" that comprise ORB/POA-level state, and also of overcoming the non-deterministic multithreading inherent in the ORB. Thus, while ORBs may be safely regarded as black-boxes for almost all practical purposes, their internal mechanisms need to be understood, and often compensated for, in the interests of providing strong fault tolerance. By transparently handling all of the difficult issues associated with replication, non-determinism, recovery, state transfer and asynchrony, Eternal makes it possible for application programmers, who are not necessarily experts in reliability, to provide strong fault tolerance easily and more quickly for their existing CORBA applications.

Acknowledgments

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, and by the Multidisciplinary University Research Initiative, Air Force Office of Scientific Research, under Contract F49620-00-1-0330.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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, Oct. 1992.

[5] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Antwerp, Belgium, September 2000.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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, Oct. 1999.

[11] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Gateways for accessing fault tolerance domains. In *Proceedings of Middleware 2000, Lecture Notes in Computer Science 1795*, pages 88–103, New York, NY, April 2000.

[12] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, Antwerp, Belgium, September 2000.

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

[14] Object Management Group. Fault tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.

[15] 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.

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

[17] 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.

[18] 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.