# State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects *

P. Narasimhan, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
priya@alpha.ece.ucsb.edu, 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 reliable totally-ordered multicast messages for conveying the IIOP messages of the application. To maintain replica consistency even as replicas fail and are recovered, Eternal ensures the retrieval, assignment and transfer of the three kinds of state – application-level, ORB/POA-level and infrastructure-level state – that are associated with each replicated object. Eternal's mechanisms for recovery include the synchronization of the state retrieval and the state assignment messages, as well as the logging and replay of messages and checkpoints.*

## 1 Introduction

Standards, such as the Object Management Group's Common Object Request Broker Architecture (CORBA) [13], aim to simplify application development by freeing the application programmer from low-level system details. CORBA applications consist of client objects and server objects, with client objects invoking server objects that return responses to the client objects after performing the desired operations. CORBA's Object Request Broker (ORB) acts as an intermediary between a client object and a server object, allowing them to interact, transcending differences in their programming languages and their physical locations. The Portable Object Adapter (POA), a server-side entity that deals with the actual implementations of a CORBA server object, allows application programmers to build implementations that are portable across different vendors' ORBs. CORBA's General Internet Inter-ORB Protocol (GIOP) and its TCP/IP-based mapping, the Internet Inter-ORB Protocol (IIOP), allow client and server objects to communicate regardless of differences in their operating systems, byte orders, hardware architectures, etc.

Enhancing CORBA with fault tolerance while maintaining CORBA's transparency and simplicity of application programming is a challenge. The Eternal system [10] 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 nor the ORB needs to be modified to benefit from the fault tolerance that the Eternal system provides.

## 2 The Eternal System

The Eternal system provides fault tolerance for applications running over commercial off-the-shelf implementations of CORBA. The mechanisms implemented in the Eternal system work together efficiently to provide *strong replica consistency* with low overheads, and without requiring the modification of either the application or the ORB.

In the Eternal system, the client and server objects of the CORBA application are replicated, and the replicas are distributed across the system. Different replication styles – active, cold passive and warm passive replication – of both client and server objects are supported. To facilitate replica consistency, the Eternal system conveys the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying Totem system [9].

The structure of the Eternal system is shown in Figure 1. The Eternal Replication Manager replicates each application object, according to user-specified fault tolerance properties (such as the replication style, the checkpointing interval, the fault monitoring interval, the initial number of replicas, the minimum number of replicas, *etc.*) and distributes the replicas across the system.

The Eternal Interceptor[1] captures the IIOP messages (containing the client's requests and the server's replies), which are intended for TCP/IP, and diverts them instead to the Eternal Replication Mechanisms for multicasting via Totem. The Eternal Replication Mechanisms, together with the Eternal Recovery Mechanisms, maintain strong consistency of the replicas, detect and recover from faults, and sustain operation in all components of a partitioned system, should a partition occur.

The Eternal Resource Manager monitors the system resources, and maintains the initial and the minimum number of replicas. The Eternal Evolution Manager exploits object replication to support upgrades to the CORBA application objects. The Replication Manager, the Resource Manager and the Evolution Manager are themselves implemented as collections of CORBA objects and, thus, can benefit from Eternal's fault tolerance. The Eternal system implements the new Fault Tolerant CORBA standard [14].

*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.

[1] Unlike CORBA's ORB-level portable interceptors, Eternal's Interceptor is an IIOP message interceptor that is not part of the ORB stack and is located outside the ORB, at the ORB's socket-level interface to the operating system.

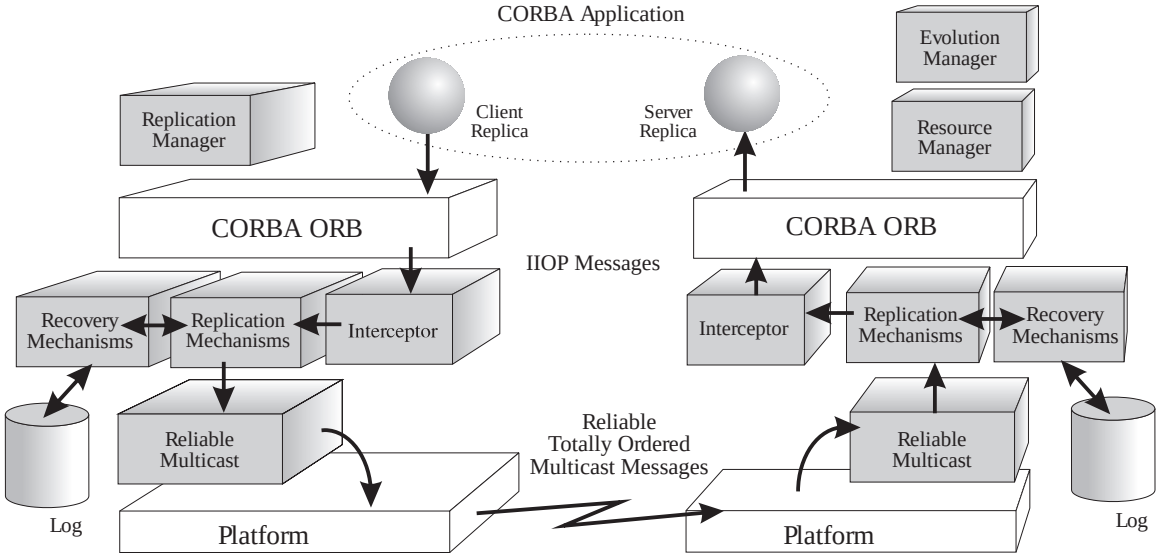


Figure 1: The structure of the Eternal system.

### 2.1 Strong Replica Consistency

For ensuring strong replica consistency of the application, application objects must be *deterministic* in their behavior so that if two replicas of an object start from the same initial state, and have the same sequence of messages applied to them, in the same order, the two replicas will reach the same final state. Challenges in maintaining replica consistency include:

- **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 group communication system 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, may lead to duplicate operations. For example, when 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 that enable the Replication Mechanisms 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. Eternal provides mechanisms [11] to ensure replica consistency, regardless of the multithreading of the ORB or the application.
- **Recovery.** Replicating an object allows it to continue providing useful services when one of its replicas fails. For true fault tolerance, it must be possible to recover a failed replica, and to reinstate it 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. The focus of this paper is on Eternal's mechanisms for providing state transfer and recovery for strongly consistent replicated CORBA objects.

## 3 Supporting Replication Styles

The mechanisms required for the consistent recovery vary with the replication style – active replication, warm passive replication, cold passive replication – of the replicated object. For active replication, as shown in Figure 2(a), each server (client) replica responds to (invokes) every operation. For passive replication, one of the replicas, designated the primary replica, responds to (invokes) every operation. With warm passive replication, as shown in Figure 2(b), the remaining passive replicas, known as backups, are synchronized periodically with the primary replica's state. With cold passive replication, a backup replica is loaded into memory and its state initialized from a log only if the existing primary replica fails.

### 3.1 Recovering an Active Replica

Masking the failure of an active replica is relatively simple. If an active replica fails while performing an operation, the remaining active replicas of the object continue to perform the operation and return the result.

The failure of a single active replica is relatively easy to mask, and is transparent to the other replicated objects involved in the nested operation. Thus, active replication yields substantially more rapid recovery from faults. When a failed active replica is recovered, the state of the new or recovering replica must be synchronized with the consistent state of an existing operational replica of the object.

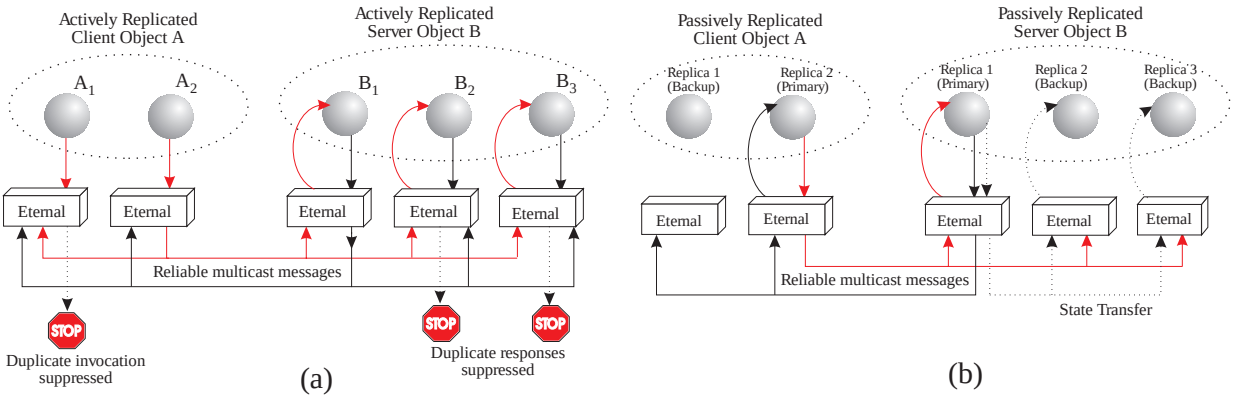


Figure 2: Replication styles supported by the Eternal system (a) active replication and (b) passive replication.

### 3.2 Recovering a Passive Replica

If a backup replica fails, it can be simply removed while the operation continues to be performed by the primary replica. On the other hand, if the primary replica fails, one of the backup replicas must be promoted to be the new primary replica.

Before the new primary replica can become fully operational (and start processing normal invocations and responses), its state must be synchronized with the state that the old primary replica had just before it failed. However, because the old primary replica is no longer available once it has failed, an operational primary's state must be periodically captured and logged so that it is available for reinstating a new primary replica.

### 3.3 Logging Checkpoints and Messages

Because the system continues to operate during the recovery of a replica, a recovering replica may be the target of normal invocations and responses from other objects in the system, even as it is having its state restored. Eternal does not discard these normal invocations and responses, but instead, enqueues them (in the order of their receipt) at the Recovery Mechanisms hosting the recovering replica. Once the replica is recovered, the Recovery Mechanisms dispatch the enqueued invocations and responses to the now-operational replica.

For passive replication, Eternal periodically captures the primary's state in the form of checkpoints. Eternal logs each checkpoint and the ordered messages that follow that checkpoint, until the next checkpoint (which overwrites the previous checkpoint) occurs. If the primary replica fails, the recovery action depends on the replication style – warm or cold passive replication. For warm passive replication, the backup replicas' states are already initialized to the primary's last checkpoint; Eternal delivers the messages (that have been logged since the last checkpoint) to the new primary replica before allowing it to become operational. For cold passive replication, Eternal must first launch the new primary replica before providing it with the primary's last checkpoint, and the logged messages, in that order.

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.

## 4 Consistent State

Unfortunately, the state required to recover a failed CORBA replica consistently is not located in a single place. For the purposes of recovery, 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 system that aims to provide strong replica consistency must maintain consistent application-level, ORB/POA-level state and infrastructure-level state across all of the replicas of every replicated CORBA object.

### 4.1 Application-Level State

Application-level state is represented by the values of the data structures of the replicated object, and is completely determined by the application programmer. Of the three kinds of state, the 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 3.

This inherited IDL interface has two methods, $get\_state()$ and $set\_state()$, both of which are intended to be implemented by the application programmer. The $get\_state()$ 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 standardize on, the format of the application-level state of every application object, the application-level state is defined to be of the CORBA type, `any`. A variable of type `any` can "hold" any primitive, structured and user-defined CORBA type.

For active replication, recovery of application-level state involves the retrieval of the current consistent application-level state via a $get\_state()$ invocation on an existing active replica, and a transfer of the retrieved state via a $set\_state()$ invocation on a new or recovering replica. For passive replication, application-level state is periodically retrieved through a $get\_state()$ invocation on the primary replica, with the returned checkpoint either logged (cold passive replication), or transferred to the backup replicas (warm passive replication). The

```
// Generic definition of application-level state
typedef any 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 3: The `Checkpointable` IDL interface that must be inherited by every CORBA object in the application to enable the checkpointing and transfer of application-level state.

three phases of recovery – state retrieval (using $get\_state()$), state transfer, and state assignment (using $set\_state()$) – must also occur in the totally-ordered message sequence to ensure replica consistency.

## 4.2 ORB/POA-Level State

Ideally, ORBs should be viewable as "black-boxes" that are 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 for the object. The existence of ORB/POA-level state implies that there really are no "stateless" objects from the viewpoint of recovery – a replicated CORBA object with no application-level state will nevertheless have ORB/POA-level state associated with it. 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 the object. Unfortunately, these "pieces" of ORB/POA-level state are not visible at the level of the CORBA object. The internal ORB/POA-level state is not standardized, and thus, not identical across ORBs from different vendors. Indeed, such standardization would be contrary to the Object Management Group's philosophy of standardizing ORB interfaces, and not their implementations.

The vendor-specific form of the ORB/POA-level state renders it a source of non-determinism if different replicas of the same object are hosted on different vendors' ORBs. Thus, for all practical purposes and for the rest of this paper, a strongly consistent replicated object has all of its replicas running over an ORB from the same ORB vendor.

When a CORBA object is replicated, each replica has its own ORB on a distinct processor. For active replication under normal operation, if the object and the ORB are deterministic, both the application-level and the ORB/POA-level state will be automatically consistent across all replicas at the end of every operation. However, under recovery, consistent state is more difficult to achieve. Even if the application-level state of the recovering active replica is synchronized with that of an operational active replica, the two replicas (the existing replica and the recovering replica) will differ in their respective ORB/POA-level states, unless these are also synchronized.

Similarly, for passive replication, under recovery, consistent replication cannot be ensured through the transfer of application-level state (from the old primary replica's logged application-level checkpoints to the new primary replica) alone; unless they are also synchronized, the respective ORB/POA-level states of the old and the new primary replicas will differ.

In this paper, we describe how Eternal handles the recovery of ORB/POA-level state, in particular, GIOP request identifiers and information negotiated between the client and the server.

### 4.2.1 GIOP Request Identifiers

CORBA's General Internet Inter-ORB Protocol (GIOP) incorporates the notion of a request identifier, a number that uniquely identifies a request-reply pair exchanged between a client and a server over a connection. The client-side ORB generates this `request_id` on a per-connection basis, and inserts it into the standard GIOP header of every outgoing request from the client to the server over the connection. On its part, the server-side ORB retrieves the `request_id`, and inserts it into the GIOP header of the corresponding IIOP reply message from the server. Typically, the client-side ORB increments the per-connection `request_id` as the number of requests sent by the client over the connection increases. The `request_id` allows the client-side ORB to match a received IIOP reply with an outstanding IIOP request. Replies whose `request_id`s do not match are discarded by the client-side ORB.

The example of Figure 4 demonstrates the replica inconsistency that ensues if ORB/POA-level state, such as the `request_id`, is not synchronized during recovery.

Figure 4(a) shows an existing replica of an actively replicated client[2] object $A$ that issues an invocation (say, of method $X$ of object $B$). This request carries a `request_id` of 350, assigned by the client-side ORB hosting this replica. Assume that this replica receives the response to this 350th invocation, and that a new replica of the same object, $A$, is now launched, and that its application-level state (but not the ORB/POA-level state, such as the `request_id` stored by the ORB) is synchronized with that of the existing replica.

The last outgoing invocation from any existing replica of $A$ carried a `request_id` of 350, which its client-side ORB "remembers." Unfortunately, the new replica's ORB does not "know" this current value of the `request_id` counter held by an existing replica's ORB. Thus, the new replica's ORB assigns an initial value, typically 0, to its equivalent `request_id` counter. If both replicas now dispatch their next invocation on object $B$, as shown in Figure 4(c), the existing replica's ORB assigns the correct `request_id` of 351 to its outgoing invocation, while the newly-recovered replica's ORB assigns a `request_id` of 0 to its outgoing invocation. Thus, although the two invocations are identical in content, their `request_id`s differ.

[2] For multi-tiered CORBA applications, the middle-tier plays the roles of both client and server; replication of the middle-tier objects involves replicating both the client-side and the server-side code.

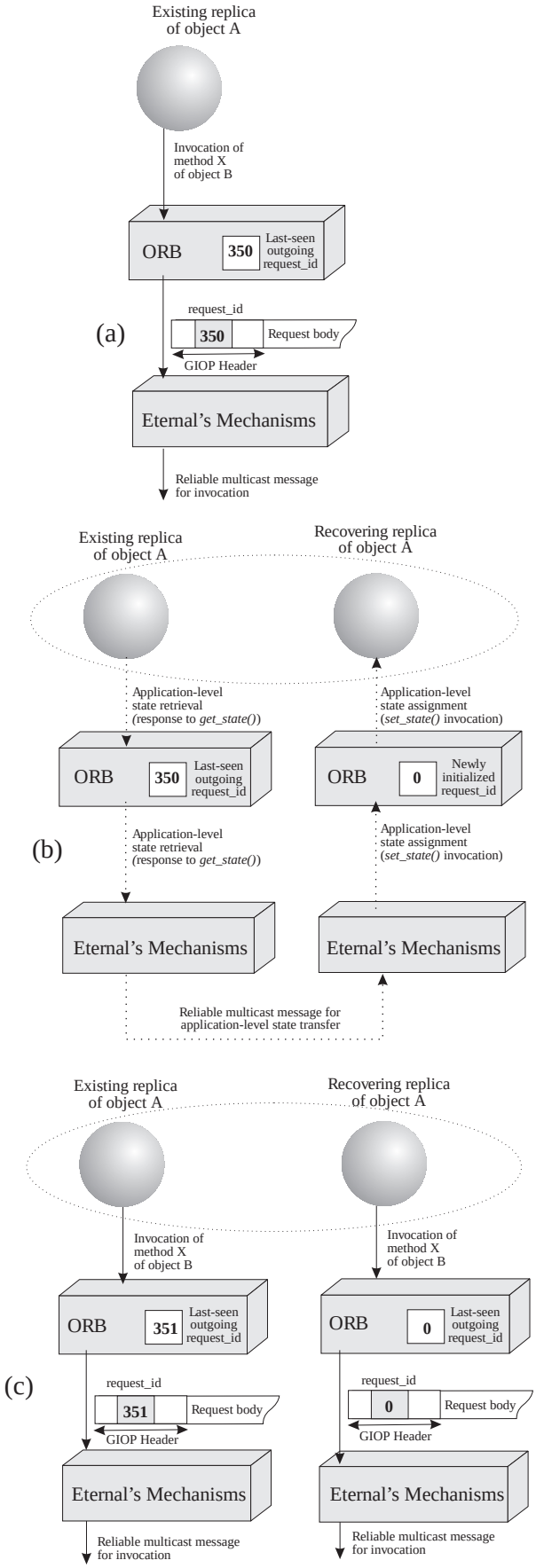


Figure 4: Replica inconsistency due to different GIOP `request_id`s assigned by client-side ORBs hosting an existing and a recovering replica. In this example, only application-level state is being synchronized.

The first of the two invocations to reach $B$ will be delivered, and the other will be discarded as a duplicate. If the invocation delivered to $B$ has a `request_id` of 0, $B$'s server-side ORB will insert a `request_id` of 0 in its outgoing IIOP reply. Unfortunately, when this reply reaches the ORBs hosting the two replicas of $A$, only the ORB that assigned the `request_id` of 0 (which, in this case, is the newly-recovered replica's ORB) will deliver the response to its replica. The ORB that hosts the existing replica (and that assigned a `request_id` of 351) will detect a mismatch between its expected `request_id` (351) and the received `request_id` (0). Thus, this ORB will not deliver the otherwise correct reply to its replica, which will now wait forever for a reply from the server.

To avoid this, a new replica's ORB must hold the same value for `request_id` counter as that held by ORBs hosting operational replicas of the same object. Otherwise, the mismatch between the returned `request_id` (from the server) and the transmitted `request_id` (from the client) will cause one or the other of the client-side ORBs to discard a perfectly valid reply from the server.

This `request_id` information is buried within the client-side ORB, and there are no "hooks" in today's ORBs to retrieve this information. Fortunately, the `request_id` information is visible from outside the ORB, in the IIOP request and response messages that are sent by the ORB.

By parsing every outgoing IIOP request message sent by a client-side ORB, Eternal can discover, and store, the ORB's current setting for the `request_id` for each of the ORB's connections. Furthermore, by transferring this stored value for the `request_id`, at the point of recovery, from the Recovery Mechanisms hosting an existing replica to the Recovery Mechanisms hosting a new replica, Eternal system ensures that the GIOP headers of all outgoing IIOP request messages from both new and existing replicas are consistent.

### 4.2.2 Client-Server Handshake

CORBA allows client-side and server-side ORBs to exchange vendor-specific information with each other. This does not detract from CORBA's interoperability because vendor X's ORB will not understand vendor Y's ORB-specific information, and can ignore it.

CORBA's GIOP allows vendor-specific information to propagate from the client to the server through the `ServiceContext` field of IIOP request messages. The server-side ORB can examine, modify, and return this `ServiceContext` in its replies to the client. `ServiceContext`s can be encapsulated into every client and server message, but are particularly used in the initial "handshake" between the client and the server.

- **Vendor-specific shortcuts.** The `ServiceContext` information may enable the client-side ORB and the server-side ORB to "recognize" that they are from the same vendor, and to use this awareness for more efficiency using vendor-specific shortcuts. For example, client-side and server-side VisiBroker 4.0 ORBs can initially negotiate a shorter object key for use, instead of the complete object key, in subsequent IIOP requests from the client.
- **Code set negotiation.** A transmission code set is the commonly agreed-upon encoding used for character and wide-

character data transfer between the client's and server's ORBs. The client-side ORB can determine a server's code sets from the code set component that the server-side ORB inserts into its server's published IOR.[3] The client-side ORB uses this information to choose character and wide-character transmission code sets for its subsequent communication with the server. Codeset negotiation is not performed on a per-request basis, but only when a client initially connects to a server.

Regardless of the purpose that the initial client-server handshake serves, both the client-side and server-side ORBs store the results of their initial negotiation, on a per-connection basis. This constitutes ORB/POA-level state that must be appropriately handled during recovery.

Consider a client $A$ communicating with a replicated server $B$ with a replica $B_1$. Assume that $A$ and $B_1$ have already completed their initial handshake, and that $A$'s ORB now stores the negotiated information, and encapsulates it (*e.g.*, in the `ServiceContext` field) in every request that it sends to $B$. When a new replica $B_2$ of object $B$ is launched, the client $A$ does not (and indeed, should not, in the interests of replication transparency) detect the addition of a new server replica and, thus, will not reenact this negotiation with $B_2$.

Unfortunately, $B_2$'s server-side ORB, having missed the initial client-server handshake, is unable to interpret the already-negotiated information in $A$'s requests. Thus, $A$'s requests, when delivered to $B_2$'s ORB, will be discarded. Thus, although $B_2$'s application-level state might be recovered, its ORB/POA-level state is not sufficiently restored as to allow it to process $A$'s requests and function normally, as the existing replica $B_1$ does.

Eternal restores the negotiated ORB/POA-level state to the ORB of a new server replica by storing the client's handshake message (that initiated the client-server negotiation), and by delivering this message to the new server replica's ORB *ahead of* any other IIOP request from the client. This artificial injection of the client's handshake message into the new server replica's ORB causes the server-side ORB to initialize its ORB/POA-level state (in terms of the client-server negotiated information) with that of the ORBs hosting operational replicas of the same server object. The new server replica's response to this artificially-injected handshake confirms, to Eternal's Recovery Mechanisms, the correct synchronization of ORB/POA-level state for the new server replica, and can be safely discarded.

### 4.3 Infrastructure-Level State

Infrastructure-level state.[4] is completely independent of, and invisible to, the replicated object as well as to the ORB and the POA, and involves only information that Eternal needs for maintaining consistent replication. The infrastructure-level state contains information essential for duplicate detection and 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 Recovery Mechanisms to filter duplicate invocations and responses intended for the replica.

During recovery, the 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 Recovery Mechanisms hosting the new replica.

The Recovery Mechanisms that receive the three kinds of state assign the application-level state first, the ORB/POA-level state next, and finally, 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.

## 5 State Transfer

The frequency of state retrieval or checkpointing is determined on a per-replicated-object basis, by the user, at the time of deploying the application, when all other fault tolerance properties (replication style, number of replicas, location of replicas, etc) for the replicated object are also determined.

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 replica consistency, to deliver a new invocation to the object. Determining whether an object is quiescent is a non-trivial exercise – 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`s, CORBA-supported invocations that do not return responses, introduces additional complications for quiescence. Eternal provides mechanisms to ascertain the quiescence of a CORBA object prior to delivering an invocation to it – these mechanisms are outside the scope of this paper.

### 5.1 Synchronization of State Transfer Messages

During recovery, the current application-level state must first be retrieved from an existing replica or a log before it can be assigned to a new replica. In the transfer of state from an existing replica to a new or recovering replica, it is important that the retrieval of state from the existing replica and the assign-

[3] A server's Interoperable Object Reference (IOR) is a stringified representation of the server's host name, port number, object key, *etc.*

[4] 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.

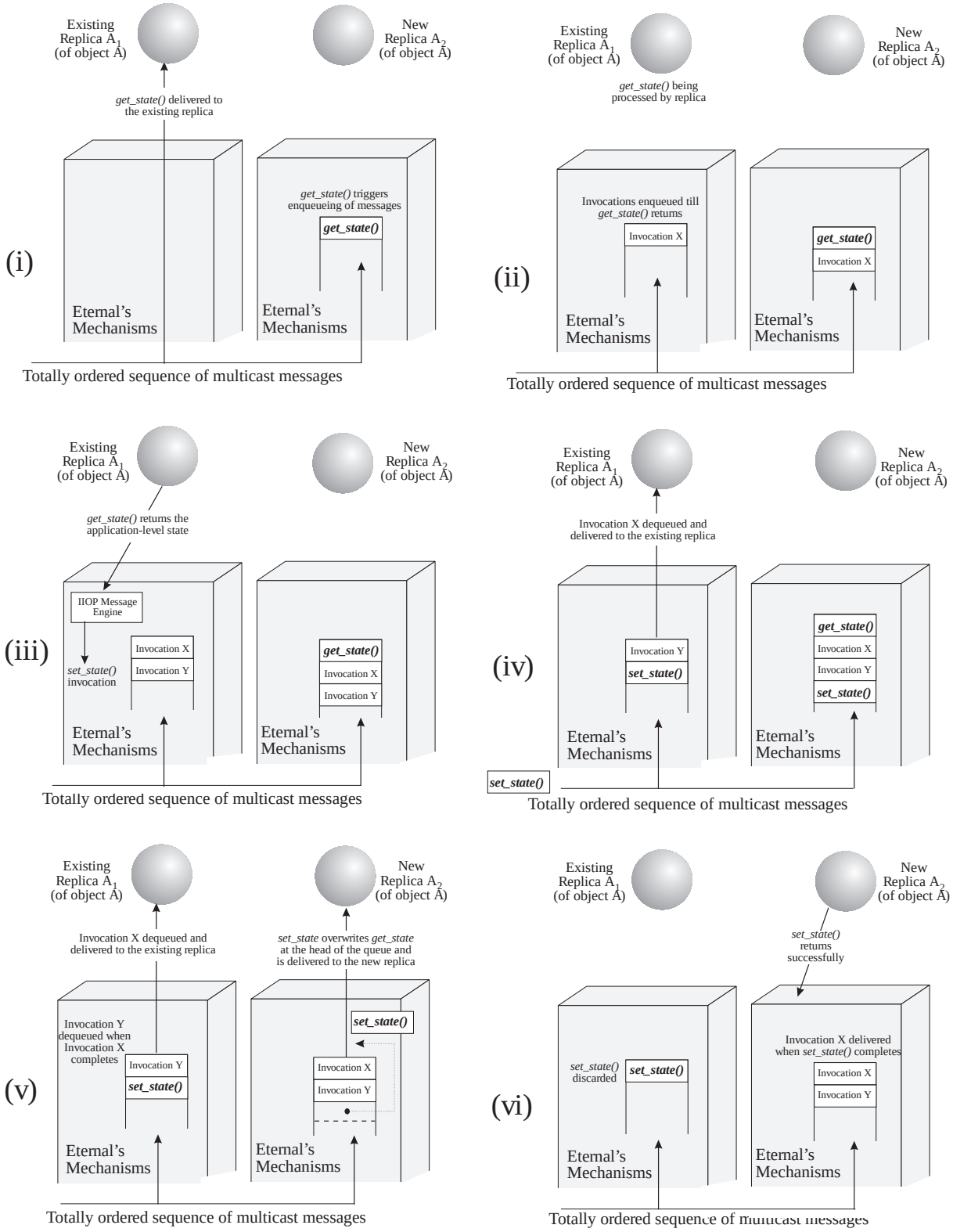


Figure 5: Synchronization of state retrieval and state assignment messages for consistent replication.

ment of the retrieved state at the new or recovering replica be seen to occur at the same *logical* point in time. Otherwise, the state retrieved by the $get\_state()$ invocation will not be the state assigned by the $set\_state()$ invocation. The tricky issues of synchronizing the transfer of state are handled by the Recovery Mechanisms.

The $get\_state()$ invocation must be delivered only to the existing replicas that have the current consistent state of the replicated object; the $set\_state()$ invocation must be delivered only to the new replica. Both invocations are received, in the sequence of multicast messages, by the Mechanisms hosting both the new and the existing replicas of the object. However, the receipt of the invocations results in different actions, depending on whether the receiving Recovery Mechanisms host either an existing or a new replica.

Figure 5 shows two replicas of a replicated object $A$, where $A_1$ is an existing replica, and $A_2$ is a new replica, and the sequence of steps in synchronizing the state transfer of the replicated object $A$.

**Step (i):** At the existing replica $A_1$, Eternal delivers the $get\_state()$ invocation as shown in Figure 5(i). However, because the new or recovering replica $A_2$ has not yet been initialized with the correct consistent state of the replicated object, the $get\_state()$ operation is not delivered to the new replica. Instead, the receipt of the $get\_state()$ invocation triggers Eternal to start the enqueueing of normal incoming IIOP messages at the new or recovering replica.

**Step (ii):** While the existing replica $A_1$ is performing the $get\_state()$ operation, regular invocations (such as Invocation $X$ shown in Figure 5(ii)) might arrive for the replicated object $A$. Because $A_1$ is in the middle of an operation, these incoming invocations are not immediately delivered to $A_1$. Because $A_2$ has not yet been recovered with the correct state, these invocations are not delivered to $A_2$, either. Eternal enqueues such regular incoming messages, at both replicas, for later delivery.

**Step (iii):** The $get\_state()$ invocation completes, as shown in Figure 5(iii). Eternal extracts the return value of the invocation, and uses it as the parameter of a $set\_state()$ invocation that it fabricates. Eternal "piggybacks" the relevant pieces of ORB/POA-level state and infrastructure-level state to the fabricated $set\_state()$ invocation.

**Step (iv):** The $set\_state()$ invocation is multicast, along with the piggybacked ORB/POA-level state and the infrastructure-level state associated with the existing replica $A_1$. Replica $A_1$ is once again free to process invocations; Eternal delivers to $A_1$ the ordered messages (such as Invocations $X$ and $Y$) that arrived for $A_1$ while $A_1$ was processing the $get\_state()$ invocation. Figure 5(iv) shows the first such enqueued invocation $X$ being dequeued and delivered to the existing replica $A_1$.

**Step (v):** When the $set\_state()$ invocation is received by Eternal at the new or recovering replica, as shown in Figure 5(v), this invocation overwrites the message at the head of the message queue (a position previously occupied by the $get\_state()$ invocation). The piggybacked ORB/POA-level state and infrastructure-level state are extracted and assigned to their respective counterparts for replica $A_2$, while the $set\_state()$ invocation containing the application-level state is delivered to $A_2$. At the existing replica $A_1$, the $set\_state()$ invocation is enqueued in the order of its arrival.

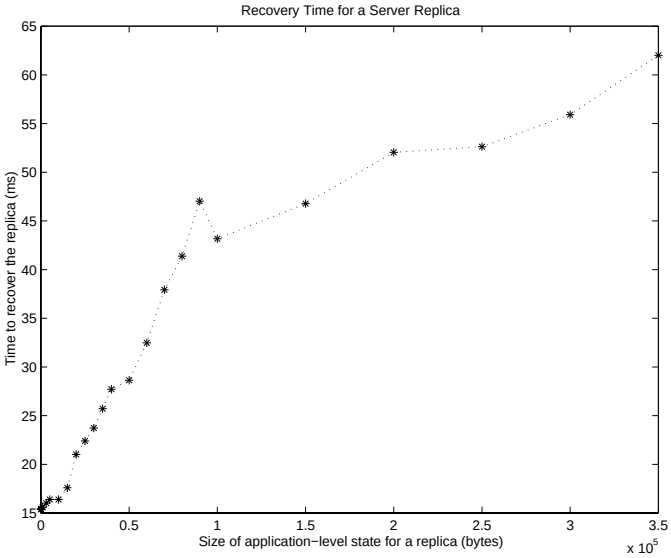


Figure 6: Variation of the recovery time for a server replica with the size of the replica's application-level state.

**Step (vi):** The $set\_state()$ invocation on the new or recovering replica returns a response (without throwing an exception), and the new replica $A_2$ is now recovered and ready to process normal invocations and responses. The enqueued messages are delivered, in order, to the new or recovering replica, as shown in Figure 5(vi). The $set\_state()$ invocation, when it reaches the head of the message queue at the existing replica $A_1$, will simply be discarded because $A_1$ is already recovered.

Thus, the logged $get\_state()$ invocation at the new or recovering replica is used to represent the *state synchronization point*, in the totally ordered message sequence, at which the state assignment must occur through its counterpart $set\_state()$ invocation. This careful synchronization of the positions of the $get\_state()$ and $set\_state()$ messages in the incoming invocation sequence at the new and existing replicas is essential to consistent recovery of every replicated object.

# 6 Implementation and Performance

The Eternal system provides support for the replication of unmodified CORBA objects running over unmodified commercial ORBs, including Inprise's VisiBroker, Iona Technologies' Orbix, Xerox PARC's ILU, Washington University, St. Louis' TAO, Vertel's e*ORB, Expersoft's CORBAplus, Object-Oriented Concepts' ORBacus and AT & T Laboratories' omniORB2. The overheads, under normal fault-free operation, of the interception, multicast and replica consistency mechanisms of our prototype Eternal system are reasonable, within the range of 10-15% of the response time for fault-tolerant CORBA test applications, over their unreplicated counterparts.

The performance of the Eternal system during the recovery of a new or failed replica of an object is shown in Figure 6. The graph 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.

The client object of the test application acts as a packet driver, sending a constant stream of two-way invocations to the actively replicated server object. 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 system, the Ethernet medium necessitates the fragmentation of any IIOP 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 needed to transfer its state, and therefore the time to recover a new or failed replica of the object, 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 constraints placed 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).

# 7 Related Work

Much of the early work on systematic message logging [5] in distributed systems was undertaken by Elnozahy and Zwaenepoel in their Manetho system [4]. They devised algorithms for sound uncoordinated logging that avoid cascaded rollbacks during recovery, at the expense of rather complex recovery algorithms. Interesting recent work on logging and recovery has been undertaken by Alvisi and Marzullo. In [1] they investigated conditions under which no process is left in an inconsistent state, while in [2] they addressed the piggybacking of nondeterministic operations onto messages in the message log to ensure deterministic replay.

The Delta-4 system [16] was aimed at providing fault tolerance in a distributed Unix environment, through the use of an atomic multicast protocol to ensure tolerate crash faults at the process level. Delta-4 included support for active replication, passive replication, as well as hybrid semi-active replication of software components on distinct processors. Backward error recovery is achieved by integrating checkpointing with interprocess 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 [6] system aims to provide mechanisms for building fault-tolerant applications in a flexible way through the use of libraries of metaobjects. Separate metaobjects 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 primary replica's state to the backup replicas in the case of passive replication.

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) [7] 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.

Developed at the University of Newcastle, 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 reestablished between them.

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 wishing to contact replicated objects.

The AQuA architecture [3] 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) [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.

OGS, AQuA, Maestro and DOORS deal with the consistency of application-level state by having application objects

inherit from an IDL interface with state retrieval and assignment methods similar to those of our `Checkpointable` IDL interface. To the best of our knowledge, however, none of these fault-tolerant CORBA systems has addressed the issues of ORB/POA-level state and infrastructure-level state that are essential in ensuring strongly consistent replication and recovery.

# 8 Conclusion

The Eternal system provides support for the consistent replication and recovery of unmodified CORBA client and server objects running over unmodified CORBA-compliant off-the-shelf ORBs. Eternal's Recovery Mechanisms include support for the logging of messages and the logging of checkpoints, as well as for the retrieval, transfer and assignment of state.

For every replicated CORBA object that it supports, Eternal maintains the consistency of the three kinds of state – application-level state, ORB/POA-level state and infrastructure-level state – that are inevitably present in a fault-tolerant CORBA system. Eternal ensures that the three kinds of state are synchronized across all of the operational replicas of a CORBA object, regardless of the object's replication style, and in a manner that is transparent to the ORB and to the CORBA application. Eternal's enqueueing and dispatching of the messages for retrieving and assigning the three kinds of state ensures that the recovery of failed replicas is concurrent with the normal operation of existing replicas; thus, Eternal allows the system to continue operating in the presence of faults, and during recovery.

True to the spirit of the new Fault-Tolerant CORBA standard that it implements, Eternal maintains strong replica consistency, as replicas process invocations and responses, as faults occur, causing replicas to fail, and as it recovers replicas after a fault.

# References


[1] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[2] K. Bhatia, K. Marzullo, and L. Alvisi. The relative overhead of piggybacking in causal message logging protocols. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 348–353, West Lafayette, IN, October 1998.

[3] 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.

[4] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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*, 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 (adopted specification). OMG Technical Committee Document orbos/2000-04-04, March 2000.

[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.