

Lessons Learned in Building a Fault-Tolerant CORBA System

P. Narasimhan

Institute of Software Research International, School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213-3890

L. E. Moser and P. M. Melliar-Smith

Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

The Eternal system pioneered the interception approach to providing transparent fault tolerance for CORBA, which allows it to make a CORBA application reliable with little or no modification to the application or the ORB. The design and implementation of the Eternal system has influenced industrial practices by providing the basis for the specifications of the Fault-Tolerant CORBA standard that the Object Management Group adopted. In this paper, we discuss our experience in developing the Eternal system, with particular emphasis on the challenges that we encountered and the lessons that we learned.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [3] is middleware that supports distributed client-server applications, with client objects invoking server objects that return responses to the client objects after performing the requested operations. CORBA's Interface Definition Language (IDL) allows server objects to make public the services that they offer and to allow client objects to send requests to the server. The key component of CORBA, the Object Request Broker (ORB), acts as an intermediary, enabling the client and server objects to communicate, despite any differences in their programming languages (language transparency) or their physical locations (location transparency). CORBA's TCP/IP-based Internet Inter-ORB Protocol (IIOP) promotes interoperability by allowing the client and server objects to communicate, even if their operating systems and hardware architectures differ.

Enhancing CORBA with fault tolerance while maintaining CORBA's transparency, interoperability and simplicity of application programming is a challenge. The Eternal system [1, 2] addresses this challenge by providing fault tolerance for CORBA applications, without requiring the

CORBA application programmer to be concerned with the difficult issues of fault tolerance. The CORBA application code requires minimal modification to benefit from the fault tolerance that Eternal provides. Thus, Eternal enables existing CORBA applications to be rendered fault-tolerant easily and quickly by application programmers who have little or no prior experience in fault tolerance.

Eternal provides fault tolerance for CORBA applications by replicating the objects of the application. Multiple identical copies, or *replicas*, of an object allow the object to continue to provide service, even if some of its replicas (or the processors hosting some of its replicas) fail. Eternal maintains *strong replica consistency*, even as objects and processors fail, as objects receive and process method invocations, and as objects are created and destroyed. Our experience in designing and implementing Eternal led to our participation in establishing the Fault-Tolerant CORBA standard [4], which corresponds closely to our fault tolerance implementation. The focus of this paper is on the challenges that we encountered while developing Eternal, and on the approaches that we used to address those challenges. Our intention here is to share our knowledge and the lessons that we learned for the benefit of other fault-tolerant middleware researchers and practitioners.

2 Lessons Learned

Most of the challenges arose from our requirement to provide strong replica consistency, despite any vagaries of current ORB implementations that might compromise it. The majority of these challenges will be faced, and will need to be resolved (although not necessarily using our approach, which provides one possible solution) in the development of any system that provides robust fault tolerance for CORBA.

2.1 Implementation Repositories

CORBA allows vendors to provide custom Implementation Repositories for the run-time registration and discov-

ery of server objects. The Implementation Repository typically takes the form of a vendor daemon, *e.g.*, the `orbixd` provided with Iona Technologies' Orbix ORB that runs on some processor in the system. From a fault tolerance viewpoint, the daemon has the disadvantage of being a single point of failure. Secondly, vendor daemons employ proprietary protocols, instead of the Internet Inter-ORB Protocol (IIOP) mandated by the CORBA standard, in order to increase efficiency or to reduce the use of bandwidth for connection management.

In addition, some ORBs include, in the server's object reference, the host name and port number of the daemon, rather than the host name and port number of the server object. Thus, on obtaining this object reference, clients contact the daemon first, with no knowledge of the server's location. The daemon then facilitates the connection establishment between the client and the server. Typically, the client retains its connection to the daemon on the server's machine, in addition to its connection with the server itself. The challenges here are to find a way to ensure that the daemons do not present a single point of failure, and that they do not compromise the reliability provided to normal client-server IIOP communication.

In order for the Eternal system, rather than a vendor daemon, to remain in control of connection management and server activation and, thereby, to render the CORBA application reliable, it was necessary to develop mechanisms that support the interception of the daemon itself. Fortunately, in the case of Orbix, the daemon is endowed with an IDL interface, `IT_daemon`, which means that it can be regarded as any other CORBA server object. This IDL interface supports methods to register and activate servers, to assign values to connection-specific parameters, and to manage client-server communication. Because the daemon's behavior (as seen through its IDL interface) is similar to that of other CORBA objects, the daemon can be forced to use IIOP for communicating with server and client application objects.

Although the ORB daemon can be handled as a CORBA object, it need not be replicated (unlike the application objects, which must be replicated for reliability) because each daemon maintains information only about the CORBA servers that are located on the same processor as itself. Furthermore, the daemon's role is important only in the "match-making" of a client with a server object; the daemon plays no role in the communication of that client-server pair after it has helped the client and the server to "find" each other. Thus, the crash of the daemon on a processor makes the server replicas on that processor inaccessible only to future clients that have yet to contact the server. If the server is replicated, other replicas of the server exist on operational processors. If a client attempts to contact a daemon that has crashed, our interception of both the client and the daemon

allows the client to be diverted to an operational daemon. Eternal shields the client from the crash of the original daemon (that the client wished to contact); furthermore, the results returned by the operational daemon are valid, and provide the client with adequate information to contact operational server replicas.

Handling the daemon is trickier than handling other CORBA objects. The reason is that, in addition to its IIOP communication, the daemon can initiate proprietary communication that does not form a part of the CORBA standard, and is rarely documented by the ORB vendor. For providing fault tolerance to CORBA applications that employ the Implementation Repository or vendor daemons, the unreliability of the daemon, the proprietary protocols used by the daemon, and the role of the daemon in client-server communication must be taken into account. Therefore, to ensure that the application does not crash due to the failure of the daemon, Eternal uses multiple daemons, along with mechanisms that allow clients to fail-over, or re-connect, transparently to an operational daemon if the original targeted daemon crashes.

2.2 ORB/POA-Level State

Ideally, ORBs should be viewable as stateless "black-boxes". In reality, because the ORB and the Portable Object Adapter (POA) handle all connection and transport information on behalf of a CORBA object that they support, the ORB and the POA must maintain some information for the object. This implies that there really are no "stateless" objects – a CORBA object with no application-level state will nevertheless have associated ORB/POA-level state.

For a given CORBA object, its associated ORB/POA-level state 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. This ORB/POA-level state is modified as the ORB dispatches operations to the object, dispatches threads within the object, establishes connections to/from the object, and processes outgoing messages from the object. Unfortunately, the "pieces" of ORB/POA-level state are not visible at the level of the CORBA object. In addition, the Object Management Group standardizes only the interfaces to, and not the implementations of, the ORB or the POA. Thus, the internal ORB/POA-level state is not guaranteed to be identical across ORBs from different vendors.

When a CORBA object is replicated, each replica has its own ORB on a distinct processor. When a new/failed replica is recovered, consistent state is more difficult to achieve. Even if the application-level state of the recovering replica is synchronized with that of an operational replica, the two replicas will differ in their respective ORB/POA-level states, unless these are also synchronized. Eternal handles the consistent recovery of the various "pieces"

of ORB/POA-level state, including request identifiers and client-server negotiated information.

Request Identifiers. CORBA incorporates the notion of a request identifier, a number that uniquely identifies a request-reply pair exchanged between a CORBA client and a CORBA server over an established connection. The client-side ORB generates this `request_id` on a per-connection basis, and inserts it into every outgoing IIOP 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 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_ids` do not match are discarded by the client-side ORB.

When a new/failed replica is recovered, its client-side ORB must hold the same value for that `request_id` counter as that held by client-side ORBs hosting operational replicas of the same object. Otherwise, the mismatch between the returned `request_id` (contained in replies directed to this object) and the transmitted `request_id` (contained in requests sent by this object) will cause the client-side ORB of one or the other of the replicas of the object to discard a valid reply.

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 reply messages that are sent by the ORB. By parsing each outgoing IIOP request message sent by a client-side ORB, Eternal discovers, and stores, the ORB’s current setting for the `request_id` for each of the ORB’s connections. By transferring this stored value for the `request_id` from the ORB hosting an existing replica to the ORB hosting a new replica, Eternal ensures that outgoing IIOP request messages from both new and existing replicas are consistent.

Client-Server Negotiated Information. CORBA 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. `ServiceContexts` can be encapsulated into every client and server message, but are particularly used in the initial “handshake” between the client and the server. This initial handshake usually serves to establish efficient short-cuts for subsequent client-server communication, or to negotiate a code-set for the exchange of messages between the client and the server. This surreptitious commu-

nication is a source of problems for consistent recovery because a new CORBA server replica is completely unaware of the negotiations that have already taken place between the client and existing replicas of the same object.

To handle this, Eternal stores the client-initiated handshake message, and delivers 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 re-set its ORB/POA-level state (in terms of the client-server negotiated information) automatically to correspond to that of the ORBs that host existing replicas of the server object. The new server replica’s reply to this artificially-injected handshake serves to confirm, to Eternal, the correct synchronization of ORB/POA-level state at the new replica, but need not be propagated to the rest of the system, and can be safely discarded.

Although the intention of CORBA is to allow application programmers to be concerned only about interfaces, the implementor of a Fault Tolerant CORBA infrastructure must look beyond interfaces. Thus, if a CORBA object is replicated, the implementor must face the non-trivial challenge of discovering, retrieving and re-setting the ORB/POA-level state (without the luxury of having access to the ORB’s source code or to the ORB developers) to ensure consistent recovery. Another consequence of the ORB/POA-level state is that, for the same set of ORB interfaces, it is possible to have ORB implementations that differ in their internal state. This implementation-dependent nature of the ORB/POA-level state means that different replicas of the same object cannot be hosted on ORBs from different vendors (*i.e.*, it is not possible to have a two-way replicated object with one replica hosted on ORB *X* and the other replica on a different implementation, ORB *Y*) because no assurances can be provided on the equivalence of the ORB/POA-level states of the respective ORBs. For all practical purposes, a strongly consistent replicated object must have all of its replicas running on an ORB from the same ORB vendor.

2.3 Multithreaded ORBs and Objects

Many commercial ORBs are multithreaded, which can yield substantial performance advantages, but unfortunately also results in non-deterministic behavior. The specification of multithreading in CORBA does not place any guarantee on the order of operations dispatched by a multithreaded ORB. The ORB may dispatch several requests for the same object within multiple threads at the same time. This is further complicated by the fact that several different concurrency models (thread-per-object, thread-per-request, *etc.*) are supported by current commercial ORBs. These include thread-per-request, where one thread is spawned for each new invocation on an object, and thread-per-object, where a single thread executes all invocations on an object.

Multithreading in the ORB or the application does not pose a problem unless the threads update *shared state* in the process. Objects that are co-located within the same process or address space might access and update shared variables, and if those objects are replicated, such accesses/updates to the shared state might occur in different orders across different replicas, leading to replica inconsistency. If the ORB and the CORBA application were implemented to be free of shared state, *e.g.*, global variables, to which multiple threads had simultaneous write-access, we could be certain that multithreading would not compromise consistent replication. Our experience with many different ORBs has shown us that this is, unfortunately, not the case for today's ORBs.

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

Eternal employs an *operation scheduler* to control the dispatching of threads and operations within each replicated MT-domain, transparently both to the objects and threads within the MT-domain, and to the multithreaded ORB that hosts the MT-domain. The scheduler dictates the creation, activation, deactivation and destruction of threads, within the replica of a MT-domain, as required for the execution of the current operation “holding” the logical thread-of-control. The scheduler is in a position such that it can override any thread or operation scheduling performed by either the non-deterministic multithreaded ORB within the replica, or by the replica itself. To ensure a single logical thread-of-control within the MT-domain, the scheduler may delay or reschedule invocations and responses on a MT-domain. This is necessary because another operation can assume the MT-domain's logical thread-of-control only when the current operation within the MT-domain completes.

The scheduler at each MT-domain replica receives the same sequence of totally-ordered messages containing invocations and responses destined for the MT-domain. Based on this incoming sequence of messages, each replica's scheduler decides on the immediate, or delayed, delivery of the messages to that replica. Thus, at each MT-domain replica, the scheduler's decisions are identical, and operations within the MT-domain are dispatched identically.

While the MT-domain model may seem somewhat restrictive in terms of the effective concurrency achieved, those restrictions are necessary to achieve consistency for replicated multithreaded CORBA applications. Replica de-

terminism, unfortunately but inevitably, reduces the degree of concurrency within the application. However, if objects are indeed independent of each other, and do not share state within the same address space, they can be assigned to different processes and our operation scheduler will schedule them concurrently without restriction. The memory protection between processes, provided by the operating system, ensures that objects in different processes do not share state (unless they use shared memory), and are indeed independent. In the absence of such guarantees, the non-determinism of the ORB or the application must be overcome, typically through the serialization of operations that enter the ORB.

2.4 System Exceptions

CORBA allows an object to raise *user exceptions* in response to an invalid input; these user exceptions are typically defined as a part of the object's IDL interface description. User exceptions do not pose any challenges for consistent replication because, if an object is deterministic, then all of the replicas of the object will raise the identical user exception in response to an illegal input parameter.

However, the ORB and the application can also raise *system exceptions* in response to transient or anomalous local system conditions. An ORB can raise a system exception, for instance, when an invocation fails in allocating dynamic memory (the NO_MEMORY system exception). The different kinds of anomalous events, and the system exceptions that result, are outlined in the CORBA specification. Typically, the anomalous conditions that trigger a system exception are not likely to occur identically on every processor. Furthermore, CORBA system exceptions sometimes do not provide sufficient information about whether the operation was initiated, or completed, before the exception was raised; in fact, CORBA actually defines a COMPLETED_MAYBE status that may be propagated along with a system exception, to signal that the ORB is unable to determine the exact status of the operation.

Thus, in a fault-tolerant CORBA system, it is possible for some of the replicas (or some of the ORBs hosting replicas) of a server object to encounter some anomalous condition on their respective processors/ORBs, and to raise the appropriate system exception, while other replicas of the same object continue to function normally. One consequence is that a client of a server object might receive a system exception in response to a request, and might subsequently terminate its connection to the server, although other server replicas have completed the request successfully, without raising any exceptions (in this scenario, the system exception has reached the client earlier than the successful responses). The damage done can be worse. For instance, the client might attempt to re-establish its connection with the server, and re-issue the same invocation,

hoping for a successful response this time. Unfortunately, some of the target server replicas have already processed this invocation correctly the previous time, without raising any exceptions. These server replicas will execute the same request twice, while other replicas (those that returned system exceptions, previously, for the same request) may execute it only once, and possibly, with success this time. Thus, the state of the server replicas might become inconsistent.

Eternal addresses system exceptions by providing mechanisms at the client-side to collect the incoming responses. If the first received response to the client's request contains a successful response or a user exception, it forwards the response to the client, and discards all subsequently received responses (from the other server replicas) for that request. If the first received response contains a system exception, it retains this exception, but does not forward the exception to the client just as yet. Instead, it waits to see if any of the other responses (from the other server replicas) contains a successful response or a user exception. If the latter occurs, it forwards the successful response or the user exception to the client, and discards the previously-stored system exception response, as well as any other responses that it receives for the same request. If every response from the replicas of the server contains the system exception, there is no choice but to propagate the system exception to the client (even this scenario is tricky because some of the system-exception responses might indicate a COMPLETED_NO or a COMPLETED_YES, *i.e.*, the operation was not initiated, or completed, respectively, while others might indicate a COMPLETED_MAYBE, *i.e.*, it is unclear whether the operation completed). Note that Eternal will never be in a state where it is waiting forever to hear from all of the replicas of the server; it has adequate information about the number of currently operational server replicas and, therefore, the maximum number of server responses to expect for the client's request.

2.5 Oneway Operations

CORBA supports the notion of asynchronous or oneway invocations. When a client invokes a oneway operation, the invocation semantics are best-effort, which does not guarantee delivery of the invocation if TCP/IP were used. With Eternal's conveying oneway invocations through a reliable multicast protocol, their delivery can be guaranteed. However, oneway operations do not return responses or exceptions and, therefore, pose a problem for determining an object's quiescence, *i.e.*, its state of readiness to accept new invocations safely, without violating the shared-state multithreading constraints that were described in Section 2.3. The problem is that even the ORB cannot determine if a oneway operation has completed. Thus, with this incomplete knowledge, even regular IIOP invocations and responses cannot be safely delivered following the dis-

patch of a oneway invocation because different replicas of an object might be at different points of execution of the dispatched oneway invocation.

The extreme approach of avoiding oneway operations altogether in writing CORBA application programs is one solution. Another approach is to provide some means for the application to let the infrastructure know when the oneway completes; this effectively forces the synchronization of the previously unsynchronized oneway operations. However, the oneway might spawn other application-level threads which continue executing, even as the oneway appears to terminate. For all practical purposes, although it is possible to provide some measure of pseudo-synchronization for oneway operations, this approach might not be effective for oneways that perform more complicated background tasks.

2.6 Observations

Our development of the Eternal system provided us with the experience to re-examine, and to invalidate, some of the commonly-held misconceptions associated with fault-tolerant distributed object systems.

Replication of Clients Is Also Essential. CORBA is a server-centric technology, and only servers possess IDL interfaces, publish object references, and provide useful services to their clients. Thus, fault tolerance is often discussed only in the context of protecting CORBA servers against faults. However, consider the case of a multi-tiered application, where the first tier is a pure client that invokes the second tier which, in turn, invokes the third tier, a pure server. The second tier is of the most interest because it plays a dual role, that of a server for the first tier, and a client for the third tier. When we replicate the objects of the second tier in order to protect them against faults, we undoubtedly replicate both the client and the server application logic. Thus, for the consistent replication of the second-tier objects, we must take both the client-side and the server-side issues into account.

Passive Replication Is Not Always a Cure for Non-Determinism. Using passive replication to replicate a non-deterministic CORBA object is believed to eliminate the side-effects that would have arisen if multiple identical copies of the object had executed simultaneously, as with active replication. This is true only for simple two-tiered applications, where a client invokes a pure passively replicated server which executes a non-deterministic call, and then returns the results to the client. The primary server replica's state is checkpointed into a log, thereby capturing the effects of the non-deterministic operation. If the primary replica fails, and a new primary replica takes over, the new primary's state can simply be over-ridden from the log, and

the client will not detect any differences because the new primary will not re-execute the non-deterministic call.

Consider, instead, a multi-tiered application, where the first tier is a pure client A that invokes the passively replicated second-tier object B which executes a non-deterministic function, say, `getTimeOfDay()`, and subsequently invokes a third-tier object C , propagating its processor's time-of-day to C . In response to this invocation, C decrements the received time-of-day by five minutes, and returns the value to B . Assume that the processor hosting the primary replica of B fails just after invoking C with a value of, say, 13:05 (but without yet receiving the results of the invocation). A new primary replica, on a different processor, will then be chosen from one of the backup replicas of B , and will have its state initialized from the log. The log will, however, not contain the results of the non-deterministic invocation of C because the results of this invocation were not yet received when the previous primary replica of B failed.

Thus, the new primary of B will re-execute the non-deterministic `getTimeOfDay()` call (obtaining a different time-of-day of, say, 12:40, due to the differences in the clocks of the processors), and will re-issue the invocation to C . Because C has already executed the invocation from the previous primary replica of B , it will return the results (containing the value 13:00), and will simply disregard this new invocation. The new primary replica of B , when it receives the result of 13:00 from C , might be confused by the contents of the response, especially because the received time-of-day (13:00) is greater than its own current time-of-day (12:40), and does not match the value (12:35) that it expected to see! This might lead the primary of B to throw an exception to A . In such cases where passively replicated objects propagate their non-determinism to other objects in the system through invocations, passive replication does not eliminate the side-effects of non-determinism.

Active and Passive Replication Require the Same Mechanisms. It is a common misconception that passive replication requires less infrastructural support and fewer mechanisms than active replication. In truth, both active and passive replication require identical support for checkpointing and state transfer; in passive replication, state transfer occurs at a predetermined frequency, while in active replication, state transfer occurs only when a new active replica is launched. The two replication styles equally require mechanisms for duplicate detection and suppression. In the case of active replication, it is easy to see how duplicate messages might arise from the simultaneously executing replicas. In the case of passive replication, duplicate messages might arise when a new primary replica (that takes over from a failed primary replica) re-issues invocations that the old primary replica had already sent, but to which it had not yet received responses.

In the case of active replication, the need for totally-ordered operations is fairly obvious because of the multiple, distributed, simultaneously executing replicas that must receive the same set of invocations in the same order. However, the need for totally-ordered operations exists for passive replication. For instance, if the total order of messages is not preserved, a passively replicated object under recovery (where a new primary replica takes over from a failed one) might invoke or process operations in an order that is inconsistent with the order in which those operations were performed prior to recovery. For strong replica consistency, both active and passive replication require the same set of mechanisms, although they differ in terms of their speed of recovery, their resource requirements, etc.

3 Conclusion

The Eternal system pioneered the interception technique to provide transparent fault tolerance for CORBA, with strong replica consistency, under both normal and fault-free conditions, and despite any non-determinism that ORBs exhibit. Eternal employs novel mechanisms to overcome ORB-specific issues such as multithreading, oneway operations, ORB/POA-level state, and CORBA system exceptions. Our experience in developing Eternal unearthed valuable insights into the fault tolerance pitfalls of CORBA application programming and ORB internals. It is our hope that fault tolerance developers, middleware vendors, and application programmers alike will benefit from the knowledge that we have gained and the lessons that we have learned and shared in this paper.

References

- [1] 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.
- [2] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.
- [3] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. OMG Technical Committee Document formal/2001-12-01, December 2001.
- [4] Object Management Group. Fault tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.