A Fault Tolerance Framework for CORBA

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

Abstract

We describe a Fault Tolerance Framework for CORBA that provides fault tolerance management and core services, implemented above the ORB for ease of use and customization, and fault tolerance mechanisms, implemented beneath the ORB for transparency and efficiency. Strong replica consistency is facilitated by a multicast engine that provides reliable totally ordered delivery of multicast messages to the replicas of an object. Transparency to the application allows application programmers to focus on their applications rather than on fault tolerance, and transparency to the ORB allows existing commercial CORBA ORBs to be used without modification. The Fault Tolerance Framework adheres to CORBA's objective of interoperability by ensuring that different implementations of the specifications of the framework can interoperate and that non-fault-tolerant objects can interwork with fault-tolerant objects.

1 Introduction

The Object Management Group (OMG) has developed the Common Object Request Broker Architecture (CORBA) [11] as a standard for distributed object computing. The CORBA standard is based on a client/server, object-oriented style of computing. The application programmer defines an interface for each application class in the OMG's Interface Definition Language (IDL), a declarative language that is independent of the particular programming language in which the classes are implemented. The Object Request Broker (ORB) of CORBA locates the server object on behalf of a client object and packages the client's method invocations, and the server's responses, into messages, defined by the General Internet Inter-ORB Protocol (GIOP) and by the Internet Inter-ORB Protocol (IIOP), the mapping of GIOP onto TCP/IP.

*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.

Although CORBA provides portability, location transparency, and interoperability of applications across heterogeneous platforms (architectures, operating systems, and languages), it lacks support for fault tolerance. Recently, the OMG has recognized the need for fault tolerance through its Request for Proposals (RFP) [12] for a fault tolerance standard for CORBA. In this paper, we describe the key components of the Fault Tolerance Framework [8] that we have proposed to the OMG in response to that RFP.

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

On the other hand, fault tolerance for CORBA could be provided through mechanisms within or underneath a CORBA ORB, but that makes it difficult for the application to interface to, and manage, the operation of the framework. Moreover, such an approach exploits complex ORB or operating system facilities that are difficult for the application programmer to understand and customize. However, a framework implemented within or underneath the ORB has the advantage of transparency due to its minimal visibility at the application level. Additionally, a framework based on this approach avoids the ORB overheads and therefore should be more efficient.

The Fault Tolerance Framework that we have developed achieves the benefits of both approaches in a novel manner, through the combination of mechanisms implemented underneath the ORB for transparency and efficiency, and services implemented above the ORB for application-level control and ease of use.

2 Architectural Overview

Figure 1 shows the Fault Tolerance Framework, which provides interfaces for the following services and mechanisms:

- **Fault Tolerance Management Services** that allow application designers to describe the static fault tolerance properties of their applications. The replication and fault tolerance of the application objects are automatic and transparent to such users, who run their unmodified applications on the unmodified commercial ORBs of their choice.
- **Fault Tolerance Core Services** that allow the application program to exercise dynamic control over replication and recovery from faults, for example, by requiring an application object to be replicated on specific processors. These services operate at the level of application objects, without exposing how object replication and recovery are implemented.
- **Fault Tolerance Mechanisms** that allow the application program or the Fault Tolerance Core Services to exercise precise control over the creation and location of individual object replicas, and direct control over recovery. The interfaces to the mechanisms are critical for interoperability within a fault-tolerant system.

The Fault Tolerance Management and Core Services are implemented as CORBA objects above the ORB. The Fault Tolerance Mechanisms are implemented as pseudoobjects (native code) beneath or within the ORB. Several components of the Fault Tolerance Framework have both service and mechanism counterparts, *e.g.*, the Replication Service and the Replication Mechanism. The intent is not duplication of effort or code but, rather, the separation of a component into service-level and mechanism-level modules that together provide the required functionality. The service-level module provides a user-accessible and customizable interface, while the mechanism-level module provides an efficient implementation of the infrastructure required by the service-level counterpart.

The Fault Tolerance Framework employs a Multicast Engine that provides reliable totally ordered delivery of multicast messages in a model of virtual synchrony to maintain strong replica consistency. Space constraints preclude a description of the Multicast Engine.

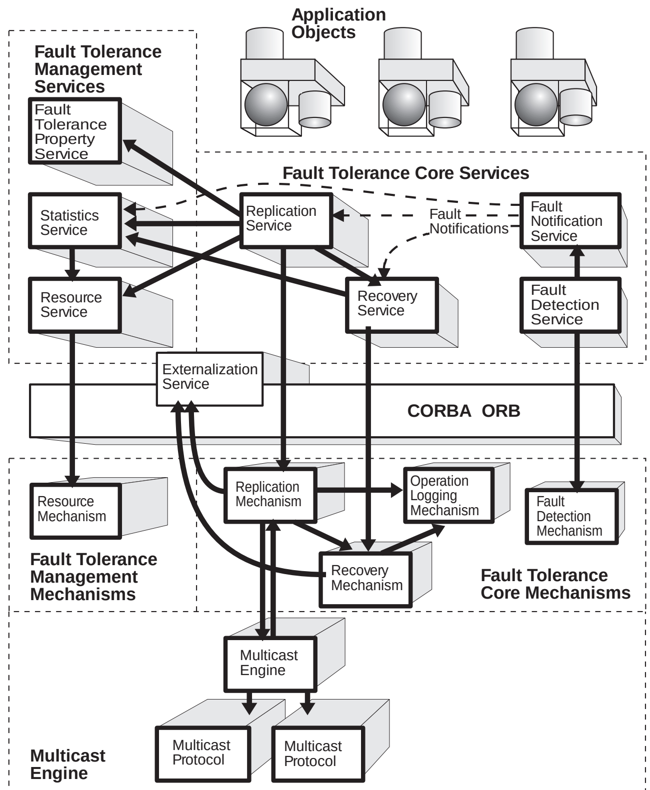


Figure 1: The structure of the Fault Tolerance Framework.

2.1 Replication Domains

A *replication domain* is a set of replicated objects under the control of a single implementation of the Fault Tolerance Core Services. Multiple implementations of the Fault Tolerance Core Services can coexist, and can even share the same processors and use the same Fault Tolerance Core Mechanisms. For example, in a wide-area application, there can be Fault Tolerance Core Services at each physical site that handle objects replicated within the local area, and Fault Tolerance Core Services that handle objects replicated across several sites within the wide area. Each replication domain has a unique replication domain identifier, provided by the application designer, who records this in the Fault Tolerance Property Service.

2.2 Object Replication

The Fault Tolerance Framework supports the individual object as the basic unit of replication. The *replicas of an object*, also referred to as *object replicas* or more simply *replicas*, implement the same IDL interface and have the same implementation source code. The behavior of each object must be deterministic.

An *object group*, also referred to as a *replicated object*, is the set of replicas of an object. It has no physical manifestation and is not located on any specific processor (because that processor might fail). The application does not access the object replicas directly (because a replica might fail). The replicated object continues to exist, even though processors or individual object replicas fail. Each object group has an object group identifier, which is unique within a replication domain.

Cold, warm and hot passive replication, and active replication with and without majority voting, are supported. The appropriate choice of replication policy is made for each object individually, concentrating fault tolerance where it is needed, without incurring unnecessary costs where less stringent fault tolerance or slower recovery is acceptable. Resistance to arbitrary faults requires active replication with majority voting and a more robust multicast protocol [10]; we focus our discussion here on protection against crash (fail-silent) faults.

2.3 Object Group Addressing

The Fault Tolerance Framework, uses two different kinds of Interoperable Object References (IORs), object IORs and object group IORs. An object IOR is an ordinary IOR, generated and used by an ORB, to address an object that it hosts; the ORB, however, has no knowledge of whether the object is a replica. An object group IOR is generated by the Replication Service and is used by the application to address a replicated object.

An object group IOR is like an ordinary IOR except that it contains the replication domain identifier and the object group identifier, rather than the identifier of a physical processor and a port number. Some ORBs send messages to verify that the host actually exists and that the object exists on that processor. The Replication Mechanism intercepts such messages and performs the required checks against the mappings established by the Replication Service.

An object group IOR appears to the ORB to be a reference to an object on some other processor, even if one of the replicas of the object group is actually on the same processor. This triggers the ORB to communicate with the object by using IIOP, which can be intercepted, rather than by using local invocations.

2.4 Strong Replica Consistency

The Fault Tolerance Framework employs several mechanisms to maintain strong replica consistency. In particular, it employs a Multicast Engine that provides reliable totally ordered delivery of multicast messages. The invocations of methods on the replicas, and the corresponding responses, are contained in multicast messages. Because the messages are delivered in the same total order at each object replica, the methods are executed by each replica in the same order. This consistent ordering of messages is the basic mechanism for ensuring that the states of the replicas remain consistent.

In addition, to maintain strong replica consistency, the Fault Tolerance Framework detects and suppresses duplicate invocations (responses) that are generated by two or more replicas of an object. It also provides transfer of state between replicas, ensuring that all of the replicas agree on which method executions precede the state transfer and which follow it. Moreover, it guarantees consistent scheduling of concurrent threads of execution by enforcing a single thread of control inside each object and disallowing direct access by one object to the data of another object.

Figure 2: A prototype GUI for the Fault Tolerance Property Service. Selection of application classes and instances are shown in the left panel, editing of fault tolerance properties in the center panel, and display of faults and utilization of resources in the right panel.

2.5 Replication of Fault Tolerance Services

The Fault Tolerance Services are implemented as standard CORBA objects that are actively replicated and thus protected against faults. It may not be obvious that such a service object can control its own replication and protect itself against faults. The technique is, however, well-understood and was first used in the SIFT aircraft flight control computer [16].

If one or more, but not all, of the replicas fail, the Fault Tolerance Services can invoke the methods that recover, or create, replicas of their own objects. As long as at least one replica is operational, the Fault Tolerance Services can operate normally, protecting themselves and the system against faults.

3 Management Services/Mechanisms

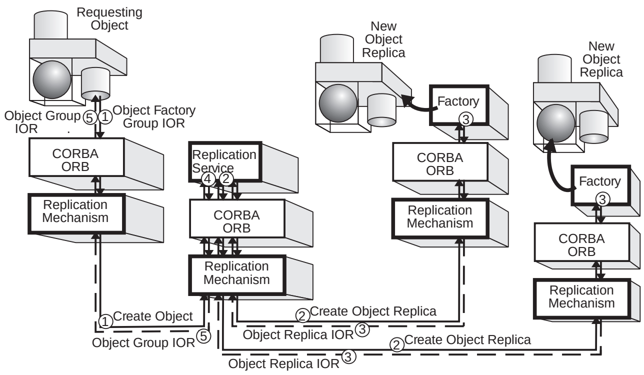
3.1 Fault Tolerance Property Service

System management for the Fault Tolerance Framework is the selection, or creation, of policies that alter the behavior of the system, primarily fault tolerance behavior. Most of the properties are set at design/deployment time, and some of them can be modified later.

A graphical user interface (GUI) for the Fault Tolerance Property Service, shown in Figure 2, allows the application designer or system administrator to define static fault tolerance properties for an application class within a replication domain.

3.2 Resource Service and Mechanism

The Resource Service exploits the Resource Mechanism located on each processor to determine the available resources, *e.g.*, processing and memory resources. The Resource Service provides a list of the available resources to the Replication Service, when the Replication Service



When the application invokes a factory to create a new object:

1. The Replication Mechanism intercepts the invocation and routes it to the Replication Service.
2. The Replication Service decides where to create the object replicas and then invokes the factories on those processors to create the object replicas.
3. The factories return object replica IORs for the replicas to the Replication Service.
4. The Replication Service creates a unique object group IOR and establishes a mapping between the object group and the object replicas. It communicates this mapping to the Replication Mechanism.
5. The Replication Service returns the object group IOR to the application that requested the creation of the object.

Figure 3: The creation of a new (replicated) object.

invokes it. The Resource Mechanism exploits native operating system mechanisms to determine current resource utilizations.

The Resource Service allows the application designer to define a *fault containment region* as a property of a resource. Two resources having the same fault containment region may experience correlated faults, for example, processors in a shared memory multiprocessor. The Replication Service will not assign replicas of the same object to resources within the same fault containment region.

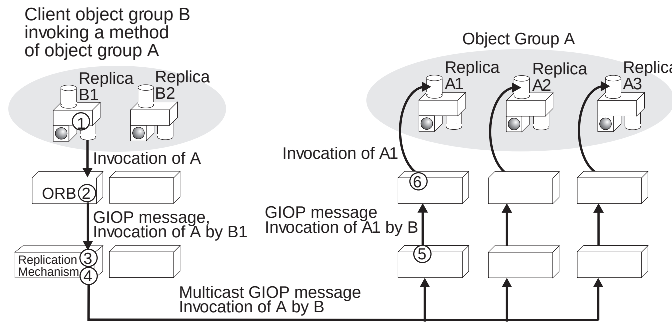
3.3 Statistics Service

A minimal Statistics Service receives event notifications from the other components of the Fault Tolerance Framework and records statistics relevant to fault tolerance, including creation of objects and replicas, occurrence of faults, time to recover from faults and availability and utilization of resources. The application designers may extend or replace this minimal service with custom software, so that it collects information of direct interest to them.

4 Core Services/Mechanisms

4.1 Replication Service/Mechanism

The Replication Service operates in conjunction with the Replication Mechanism to create and delete object replicas



When an object group B invokes a method of object group A:

1. An object replica $B1$ uses the IOR of object group A to dispatch the invocation.
2. The ORB notes that the IOR of the invoked object group A appears to refer to an object on some other processor and constructs a GIOP message, whose destination is object group A and whose source is the individual replica $B1$.
3. The Replication Mechanism, using the mapping tables provided to it by the Replication Service, replaces the invoking replica $B1$ in the source field by object group B.
4. The Replication Mechanism then encapsulates the GIOP message in a multicast message and multicasts it to the processors hosting the replicas of object group A.
5. At each of those processors, the Replication Mechanism replaces the invoked object group A in the destination field of the message by the individual replica $A1$, $A2$ or $A3$ on that processor, and passes the GIOP message to the ORB.
6. The ORB invokes the individual replica $A1$, $A2$ or $A3$ on its respective processor.

Figure 4: The invocation of an object group.

within a single replication domain. The Replication Service obtains its instructions from the application designer via the Fault Tolerance Property Service regarding which application objects to replicate and how to replicate them. It obtains information about the available resources (*e.g.*, processing and memory resources) and their current utilization from the Resource Service. The Replication Service subscribes to the Fault Notification Service to be notified of faults.

The Replication Service interface, provided to the application, allows the creation (deletion) of object groups, and also the creation (deletion) of individual object replicas on specific processors. This interface returns object group IORs, rather than object IORs for the individual replicas. The Replication Mechanism interface, accessible only to the Replication Service, allows access to the object IORs of the replicas. Figure 3 shows the role of the Replication Service and Mechanism when the application creates an object.

To create an object replica, the Replication Service uses the Recovery Service to make the replica *operational*, which entails making the replica's state consistent with that of the other replicas. For a passively replicated object, the Replication Service informs the Recovery Service of the replica that is chosen as the primary replica, and the order in which the other replicas should be used for the selection of subsequent primary replicas in the event of faults.

Figure 4 shows the role of the Replication Service and Mechanism when an object group is invoked. To ensure that the states of the replicas do not become inconsistent because the same invocation is executed multiple times, the Replication Mechanism detects and suppresses duplicate invocations (responses), as shown in Figure 5. Those duplicates that are not detected and suppressed at the source are detected and suppressed at the destinations.

The Replication Service, through the fault reports it receives from the Fault Notification Service, can remove a faulty replica from an object group, or can create a new replica, to maintain the desired fault tolerance requirements. Removal of a replica from an object group by the Replication Service causes the Replication Mechanism to terminate that replica. Immediate removal of a faulty replica is not essential. If the faulty replica continues to generate invocations (responses), the Replication Mechanism will detect and suppress duplicate invocations (responses).

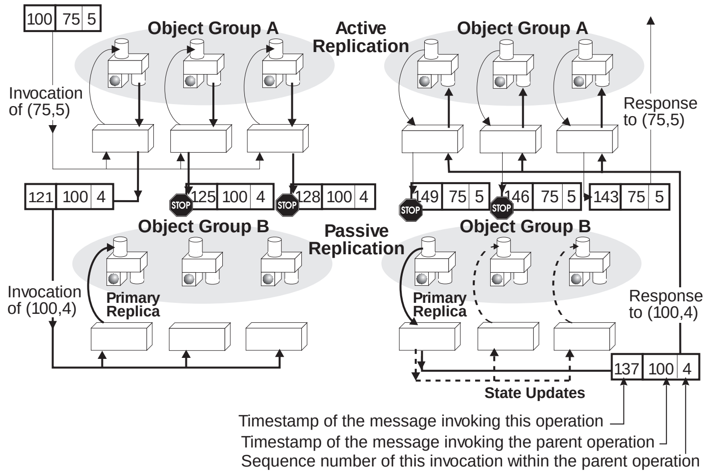
4.2 Recovery Service/Mechanism

The Recovery Service, in conjunction with the Recovery Mechanism, provides fault recovery for passive replication, and activates new objects for both passive and active replication. The Recovery Service subscribes to the Fault Notification Service. If the Recovery Service receives a report that a nonprimary replica has failed during an operation, then no recovery action needs to be taken, because the primary replica continues to perform the operation.

For a cold passively replicated object, when the primary replica fails, the new primary replica does not yet exist. Thus, the Recovery Service must first invoke the Replication Service to create the new primary replica on the appropriate processor. Following this, the Recovery Service instructs the Recovery Mechanism to obtain, from the Operation Logging Mechanism, the most recent state transfer message and the set of invocation and response messages that follow that state transfer message. The Recovery Mechanism then applies the state transfer message to the new primary replica, followed by the set of invocation and response messages, as above. The steps are shown in Figure 6.

For a warm passively replicated object, the state transfer message has already been applied by the backup replica. Thus, the Recovery Manager needs to retrieve only the recent invocation and response messages from the log and apply them to the new primary replica.

Multiple invocations on other objects, and even responses, can be generated by the original and the new primary replicas. The Replication Mechanism detects and suppresses such duplicates.



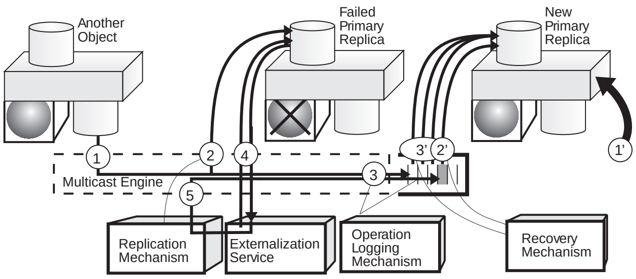
Object group A contains three active replicas and object group B contains three passive replicas.

- Some client object (not shown in the figure) invokes a method with invocation identifier $(100, (75, 5))$ on object group A. Each of the replicas in A executes the method, which results in the invocation of other methods, including the one with invocation identifier $(121, (100, 4))$ on object group B.
- The timestamp of the ''parent'' invocation that resulted in the subsequent invocations is 100. If the method invocation on object group B is the fourth in the sequence of invocations triggered by the execution of the parent invocation then, at each replica in A, the operation identifier for the invocation of B is $(100, 4)$.
- The Replication Mechanism for one of the replicas in object group A multicasts a message containing the invocation identifier $(121, (100, 4))$. When the Replication Mechanism at another replica in object group A receives this message, it suppresses its own replica's invocation also with operation identifier $(100, 4)$.
- The primary replica in object group B then executes the method, after which the Replication Mechanism transfers the state of the primary replica to the nonprimary replicas in object group B and multicasts the response to object group A using the response identifier $(137, (100, 4))$. Note that the invocation identifier $(121, (100, 4))$ and the corresponding response identifier $(137, (100, 4))$ refer to the same operation and thus have the same operation identifier $(100, 4)$.
- At the end of the operation, the Replication Mechanism at one of the replicas in object group A multicasts the response using the response identifier $(143, (75, 5))$. When the Replication Mechanism at another replica in object group A receives this message, it suppresses its own replica's response for $(75, 5)$.

Figure 5: Detection and suppression of duplicate invocations (responses).

4.3 Fault Detection Service/Mechanism

The Fault Detection Service monitors objects (typically replicas of an object), verifying that they continue to operate, and generates fault reports, using the Fault Detection Mechanism. The Fault Detection Service depends on user-defined timeouts to generate suspicions that objects are faulty, though other mechanisms may be used. An object that is slow in producing a response, or that has a faulty communication link, may be suspected by the fault detector, even though the object itself has not actually failed.



If a cold passively replicated object is invoked and the primary replica does not fail:

1. The Replication Mechanism dispatches the GIOP message containing the invocation of the passively replicated object using the Multicast Engine, which encapsulates the GIOP message in a multicast message and multicasts it.
2. The Recovery Mechanism applies the received message to the primary replica.
3. The Operation Logging Mechanism logs, but does not apply, the message at the other (nonprimary) replicas.
4. The Externalization Service allows the Replication Mechanism to obtain the state of the primary replica.
5. The Replication Mechanism dispatches the GIOP message containing the state of the primary replica to the other (nonprimary) replicas using the Multicast Engine.

If the primary replica fails:

1.′ The Replication Mechanism loads the new primary replica into its processor if it is not already loaded.

2.′ The Recovery Mechanism determines a new primary replica, extracts the most recent state transfer message from the log and then applies it to the new primary replica.

3.′ The Recovery Mechanism extracts subsequent invocation messages from the log and applies them to the new primary replica.

Figure 6: The role of the various services and mechanisms in cold passive replication.

The Fault Detection Service supports both pull monitoring and push monitoring. In the push monitoring model, an object reports periodically to the Fault Detection Service to confirm that it is alive. In the pull monitoring model, the Fault Detection Service periodically invokes a method on an object, which must respond confirming that it is still alive. Objects that fail to report or fail to respond are reported as faulty by the Fault Detection Service.

The Fault Detection Service detects only object faults; other components in the system detect other types of faults. For example, the Multicast Engine detects processor faults and network faults, the Resource Service detects resource overload faults, and the Replication Service detects inadequate replication faults. Some types of faults might be detected by the application objects themselves.

4.4 Fault Notification Service

The Fault Notification Service receives fault reports from the fault detectors and, in turn, generates fault notifications to other objects. Both application objects that need fault notifications, and service objects including the Replication, Recovery and Statistics Services, subscribe to the Fault Notification Service.

The Fault Notification Service might generate multiple fault notifications for a fault report that it receives. For example, a fault report for a processor results in a fault notification for each of the object replicas hosted by that processor. Similarly, a fault report for an object group triggers fault notifications for each of its object replicas. If a fault is reported for the last replica of an object group, a notification is generated for the object group as a whole. An application needs to register selectively for fault notifications to avoid receiving a large number of simultaneous correlated notifications.

4.5 Operation Logging Mechanism

The Operation Logging Mechanism maintains, for each object replica, a log of messages, which contains a record of invocations and responses, as well as state transfer messages, for that object. To achieve reasonable efficiency (in terms of storage and time for logging), the Operation Logging Mechanism is implemented as pseudoobjects beneath the ORB, which are present on each processor. Conceptually, a separate log is maintained for each object replica; physically, the implementation may share storage and retrieval facilities across many logs.

State transfer is performed by the *getstate* and *setstate* methods, which must be coded by the application programmer as part of the application object. Alternatively, if the ORB provides an Externalization, Objects by Value or Persistent State Service, those can be exploited to perform state transfer.

5 Prototype Implementation

We have implemented the Fault Tolerance Mechanisms of the Fault Tolerance Framework in our prototype Eternal system [7, 9], using unmodified commercial ORBs, including Inprise's VisiBroker, Iona's Orbix, Xerox PARC's ILU, Object-Oriented Concept's ORBacus and Washington University's real-time TAO ORB. The implementation is designed for Solaris 2.6 but can also operate on Linux. A port to WindowsNT is in progress. The Fault Tolerance Management and Core Services have been specified and their implementation is underway. The underlying mechanisms are the difficult part of the Fault Tolerance Framework to implement; the services are essential for the practical deployment of the Framework but their implementation is relatively straightforward.

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

The mechanisms that the Fault Tolerance Framework employs to ensure replica consistency are implemented beneath the ORB and, thus, are transparent to the application and to the ORB. The overheads are in the range of 7-15% for remote invocations with triplicated clients and triplicated servers. These low overheads include the cost of interception and replication, as well as that of multicasting GIOP messages using the Totem multicast group communication system [6].

For example, using Sun UltraSPARC2 167 and 200 MHz workstations and 100 Mbit/s Ethernet, a remote invocation and response with an unreplicated client and an unreplicated server running over VisiBroker, without the mechanisms of the Fault Tolerance Framework, requires 0.330 ms. In this case, the client and server communicate using IIOP messages transmitted over TCP/IP.

Using the mechanisms of the Fault Tolerance Framework, for the same platform and application, with three-way actively replicated client and server objects running over VisiBroker, a remote invocation and response required 0.369 ms, which represents an overhead of 12% over the unreplicated case. These measurements involved an actively replicated client object repeatedly invoking an actively replicated server object using deferred synchronous communication without message packing. The Operation Logging Mechanism did not contribute to the measured overheads because it was not required for state transfer.

With three-way passively replicated clients and servers, a remote invocation and response required 0.374 ms, which represents an overhead of 15% over the unreplicated case. These measurements involved a passively replicated client object, with the primary client replica repeatedly invoking the passively replicated server object. In the absence of a standard CORBA service that provides externalization, the state transfers for both client and server objects were hand-coded.

Little difference in the time for invocations and responses was observed between cold (no state transfers), warm (state transfer every fourth invocation) and hot (state transfer every invocation) passive replication, because the state transfers largely overlap the invocations and responses. Cold passive replication, of course, imposes a lower processing load on the processors hosting the nonprimary replicas. With cold passive replication, for the nonprimary replicas, the Replication and Operation Logging Mechanisms accounted for 1.2% of the CPU load, while the ORB and the nonprimary replicas imposed no load on the CPU at all. With hot passive replication, for the nonprimary replicas, the Replication and Operation Logging Mechanisms accounted for 5% of the CPU load, with the ORB and the nonprimary replicas exhibiting similar CPU usage, entirely for state transfers. The main overhead in both cases was due to the operating system and networking software, with CPU usages in the range of 20%.

6 Related Work

Several systems have been developed that augment CORBA application objects with fault tolerance. These systems vary in the level of fault tolerance provided to the application, in the transparency provided to the application and the ORB, and in the performance overheads incurred.

The Electra toolkit implemented on top of Horus provides support for fault tolerance by replicating CORBA objects, as does Orbix+Isis on top of Isis [1, 5]. Both Electra and Orbix+Isis integrate the replication and group communication mechanisms into the ORB and require modification of the ORB. Unlike the Fault Tolerance Framework, Electra and Orbix+Isis are non-hierarchical object systems and support only active replication.

The Maestro toolkit [15] 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.

The AQuA framework [2] employs the Ensemble/Maestro [14, 15] toolkits, the Quality Objects (QuO) runtime, and the Proteus dependability property manager. Based on the requirements communicated by the QuO runtime and the faults that occur, Proteus determines the type of faults to tolerate, the replication policy, the degree of replication, the type of voting to use and the location of the replicas, and dynamically modifies the configuration to meet those requirements. The AQuA gateway translates CORBA object invocations into messages that are transmitted via Ensemble, and detects and filters duplicate invocations (responses). The AQuA framework is more similar to the Fault Tolerance Framework than the other systems described here.

The Object Group Service (OGS) [3] provides fault tolerance for CORBA applications through a set of services implemented on top of the ORB. OGS is itself composed of several CORBA services including a group service, a consensus service, a monitoring service and a messaging service, each of which can be used as a stand-alone CORBA service. The service approach, adopted by OGS, exposes the replication of objects to the application program, but allows the application programmer to modify the class library and customize the services more easily.

The Distributed Object-Oriented Reliable Service (DOORS) [13] adds support for fault tolerance to CORBA by providing replica management, fault detection, and fault recovery as service objects above the ORB. Unlike the above systems and the Fault Tolerance Framework, DOORS focuses on passive replication and is not based on group communication and virtual synchrony. The DoorMan management interface monitors DOORS and the underlying system to fine-tune the functioning of DOORS and to take corrective action by migrating objects, if their hosts are suspected of being faulty.

Using a quite different approach from that of the Fault Tolerance Framework, Killijian *et al* [4] have defined a metaobject protocol for implementing fault-tolerant applications based on the use of inheritance and reflection. Their approach allows control by the user at the metalevel, but is heavily language dependent.

7 Conclusion

The Fault Tolerance Framework, described here, is novel in its use of a combination of services and mechanisms to provide fault tolerance for CORBA. The mechanisms are optimally implemented beneath the ORB for transparency and efficiency, and the services are implemented above the ORB for ease of use and customization by the application.

Strong replica consistency is ensured for CORBA applications by means of a reliable totally ordered message delivery service, detection of duplicate invocations and duplicate responses, transfer of state between replicas of an object ensuring that the replicas of an object agree on which method executions precede the state transfer and which follow it, and consistent scheduling of concurrent threads of execution.

The Fault Tolerance Framework can be used transparently or controlled directly by the application program, depending on the degree of control that the application requires. Transparency to the application allows the benefits of fault tolerance to become available to a much wider range of applications and users, with less effort on the part of the application programmer. Transparency to the ORB allows fault tolerance to be provided using unmodifed commercial implementations of CORBA.

References

[1] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, CA, 1994.

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

[3] P. Felber, R. Guerraoui and A. Schiper, ''The implementation of a CORBA Object Group Service,'' *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 93-105.

[4] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia and S. Chiba, ''A metaobject protocol for fault-tolerant CORBA applications,'' *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, West Lafayette, IN (October 1998), pp. 127-134.

[5] S. Landis and S. Maffeis, ''Building reliable distributed systems with CORBA,'' *Theory and Practice of Object Systems*, vol. 3, no. 1, (April 1997), pp. 31-43.

[6] 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*, vol. 39, no. 4 (April 1996), pp. 54-63.

[7] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, ''Consistent object replication in the Eternal system,'' *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 81-92.

[8] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, *Fault tolerance for CORBA*, Technical Report 98-27, Department of Electrical and Computer Engineering, University of California, Santa Barbara, OMG Technical Document orbos/98-10-08, October 1998.

[9] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, ''Replica consistency of CORBA objects in partitionable distributed systems,'' *Distributed Systems Engineering*, vol. 4, no. 3 (September 1997), pp. 139-150.

[10] P. Narasimhan, K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, ''Providing support for survivable CORBA applications with the Immune system,'' *Proceedings of the IEEE 19th International Conference on Distributed Computing Systems* (May/June 1999), Austin, TX.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.2, OMG Technical Document formal/98-07-01, February 1998.

[12] Object Management Group, *Fault-Tolerant CORBA Using Entity Redundancy: Request for Proposals*, OMG Technical Document orbos/98-04-01, April 1998.

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

[14] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd and D. Karr, ''Building adaptive systems using Ensemble,'' *Software - Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 963-979.

[15] A. Vaysburd and K. Birman, ''The Maestro approach to building reliable interoperable distributed applications with multiple execution styles,'' *Theory and Practice of Object Systems*, vol. 4, no. 2 (1998), pp. 73-80.

[16] J. Wensley, P. M. Melliar-Smith, *et al*, ''SIFT: Design and analysis of a fault-tolerant computer for aircraft control,'' *Proceedings of the IEEE*, vol. 66, no. 10 (October 1978), 1240-1255.