Measuring the Impact of Fault-tolerant CORBA on the Performance of Distributed Systems

Paper ID : 322 Number of Pages : 12

Abstract

Recently, there has been a significant trend towards developing distributed systems based on distributed object computing middleware, such as CORBA. Applications for these systems increasingly require the underlying middleware, operating systems, and networks to provide end-to-end quality of service (QoS) support to enhance their efficiency, predictability, scalability, and fault tolerance. The Object Management Group (OMG), which standardizes CORBA, has addressed these application requirements recently by standardizing the Real-time CORBA and Fault-tolerant CORBA specifications.

This paper provides three contributions to research on fault-tolerant CORBA middleware for performance-sensitive applications. First, we describe a framework called DOORS, which is a CORBA service that provides end-to-end application-level fault-tolerance. Second, we outline how the DOORS' reliability and fault-tolerance model has been incorporated into the standard OMG Fault-tolerant CORBA specification. Third, we describe results of experiments that illustrate the performance impact of using Fault-tolerant CORBA to provide fault-tolerance to CORBA applications. In particular, we show how our patterns-based design enables application developers to tradeoff various QoS aspects by customizing key parameters that control the degree of fault-tolerance and performance.

Our results comparing the performance of a DOORS-enabled CORBA application indicates that DOORS provides fault-tolerance without faulty or missed detections with only 5% degradation in performance, compared to a configuration of the same application that does not use DOORS. Moreover, the failover times after detecting faults was only ∼200 milliseconds. The work presented in this paper represents one of the first results of middleware and benchmarks based on the Fault-tolerant CORBA specification. These results provide a baseline for other researchers and developers that are empirically evaluating the capabilities of Fault-tolerant CORBA.

Keywords:

Fault-tolerant CORBA, Middleware protocols, Prototype.

1 Introduction

Emerging trends: Applications for next-generation distributed systems are increasingly being developed using standard services and protocols defined by distributed object computing middleware, such as the Common Object Request Broker Architecture (CORBA) [1]. CORBA is a distributed object computing middleware standard defined by the Object Management Group (OMG) that allows clients to invoke operations on remote objects without concern for where the object resides or what language the object is written in [2]. In addition, CORBA shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects. These features make CORBA ideally suited to provide the core communication infrastructure for distributed systems and applications.

A growing number of next-generation applications demand varying degrees and forms of quality of service (QoS) support from their middleware, including efficiency, predictability, scalability, and fault tolerance. In CORBA-based middleware, this QoS support is provided by Object Request Broker (ORB) endsystems [3], which consist of network interfaces, operating system I/O subsystems, CORBA ORBs, and higher-level CORBA services.

Prior research on CORBA middleware has explored the efficiency, predictability, and scalability aspects of ORB endsystem design, including static [3] and dynamic [4] scheduling, event processing [5], I/O subsystem [6] and pluggable protocol [7] integration, synchronous [8] and asynchronous [9] ORB Core architectures, systematic benchmarking of multiple ORBs [10], and optimization principle patterns for ORB performance [11]. This paper focuses on another dimension in the ORB endsystem design space: providing fault tolerance to CORBA applications *without* adversely affecting other QoS requirements, such as efficiency, predictability, and scalability.

Limitations with current fault tolerance strategies: There is a large body of research literature [12, 13] and tools that focus on improving the reliability and recoverability of application *processes*. For instance, Firstwatch [14], Watchd [15], and Wolfpack are tools designed to improve the fault tolerance of application processes. However, techniques that focus solely on process-based failure detection and recovery are not necessarily applicable to CORBA applications for the following reasons:

• **Overly coarse granularity:** Each application process may contain dozens or hundreds of active CORBA objects and threads. The failure of an object or thread may not cause a process to terminate, however, and hence may not be detectable by a fault tolerance strategy that only detects process failures.

• **Inability to restore complex object relationships:** A distributed object communicates with other objects via object references. When a failed object is recovered, not only the object references it held just before failure must be recovered, but other objects holding object references to it must recover their references, as well. Restoring complex object reference relationships is a complex and unique problem in CORBA that is not handled by process recovery strategies.

• **Restrictions on process checkpointing and recovery:** Distributed objects often maintain state that must be checkpointed periodically to survive crash failures. Fault tolerance strategies used for process checkpointing may incur excessive overhead or may be unable to checkpoint the desired state for all objects. Thus, a finer level of granularity for checkpointing process state is needed to permit individual objects in a process to checkpoint their internal state independently of each other.

As a result of the limitations with process-based reliability and recovery outlined above, it is necessary to explore new strategies for providing high reliability and availability to CORBA-based distributed object computing applications.

Solution approach → supporting fault tolerance for CORBA objects: Fault tolerance strategies for distributed object computing middleware, such as CORBA, address the limitations of process-based fault tolerance. In general, research on fault tolerance for CORBA ORBs and its applications can be divided into the following three strategies:

1. Integration: In this strategy, an ORB endsystem is layered on top of a reliable group communication subsystem. Orbix+Isis [13] and Electra [16] are examples of the integration strategy.

2. Interception: In this strategy, requests made by client objects are captured "external" to the ORB and mapped to a reliable group communication system. The Eternal system [17] and the AQuA framework [18] are examples of the interception strategy.

3. Service: In this strategy, a set of interfaces and objects are defined as a CORBA service that provides the policies and mechanisms for delivering fault tolerance to applications. The Distributed Object-Oriented Reliable Service (DOORS), which is our focus in this paper, is an example of this strategy.

All of these strategies have been incorporated in the OMG Fault Tolerant CORBA (FT-CORBA) specification [19], which was adopted as a standard in January, 2000. Section 4 describes related work on these strategies in more detail.

The work presented in this paper is based on the service strategy described above. We focus on this strategy because it allows application developers greater flexibility to *efficiently* select the protocol(s) used to communicate between collaborating ORB endsystems. The CORBA specification defines a standard transport protocol, IIOP, which runs atop TCP/IP. However, there are many use-cases, particularly in telecommunication and real-time systems, where ORBs must communicate over non-TCP/IP protocols, such as VME, ATM, and shared memory. Moreover, researchers have devised other messaging protocols, such as HTTP-NG [20] or GIOP-lite [7], that have a smaller "wire footprint" than the standard CORBA GIOP messaging protocol.

As explained above, applying a service strategy for fault tolerance offers significant flexibility compared with alternative strategies. However, next-generation distributed applications, such as telecommunication and command and control systems, also require real-time QoS support *and* high reliability. To address these requirements, therefore, a CORBA fault tolerance framework based on the service strategy should possess the following characteristics:

1. The overhead incurred by the fault tolerance framework should maintain application QoS requirements, such as efficiency, predictability, and scalability, within designated bounds end-to-end.
2. The fault detection and failovers incurred by servers should be transparent to clients.
3. Response time to the client should be efficient and predictable, irrespective of server failovers.

The DOORS framework described in this paper is designed to satisfy the requirements mentioned above. Our benchmarking studies that measured the impact of DOORS on bandwidth-intensive applications reveal only a 6% performance degradation. Moreover, DOORS allows applications to customize the heartbeat and polling intervals to enable them to optimize performance *and* achieve the requisite fault tolerance.

Paper organization: This remainder of this paper is organized as follows: Section 2 describes the recently adopted Fault Tolerant CORBA (FT-CORBA) standard and illustrates the concepts from DOORS that were incorporated in this standard. In addition, this section outlines the design of DOORS, which applies patterns and frameworks to enable developers to configure DOORS to optimize performance, as well as to customize its reliability and fault tolerance strategies; Section 3 describes empirical benchmarks we conducted to compare the performance of a bandwidth-intensive application with and without using the DOORS FT-CORBA framework; Section 4 describes related work; and Section 5 presents concluding remarks.

2 Overview of DOORS and the Fault Tolerant CORBA Specification

The Fault Tolerant CORBA (FT-CORBA) [19] specification defines a standard set of interfaces, policies, and services that provide robust support for applications requiring high reliability. The mechanism by which fault tolerance is provided is based on *entity redundancy* [21] in the form of replicated CORBA objects, failure detection, and recovery. The Distributed Object-Oriented Reliable Service (DOORS) is a precursor to the FT-CORBA standard that implements a core subset of the functionality defined by the standard. Thus, through our contributions and active participation in the OMG working group on Fault Tolerant CORBA, the concepts pioneered in DOORS have been incorporated in the FT-CORBA specification.

This section provides an overview of the FT-CORBA standard and compares and contrasts the DOORS framework with the FT-CORBA standard. It presents the interaction protocol among the DOORS modules. In addition, we outline how the framework components applied to DOORS enable applications to select various tradeoffs between performance and reliability. Section 3 then describes the results of our experiments that quantify the performance overhead of employing the functionality provided by DOORS.

2.1 Overview of Fault Tolerant CORBA (FT-CORBA)

Fault tolerance for CORBA objects is achieved via *replication*. Replicas of a CORBA object are created and managed as a "logically singleton" [22] composite object. In addition to standardizing the interoperable object references (IOR)[1] used for the individual replicas, FT-CORBA defines an IOR for composite objects called the *interoperable object group reference* (IOGR).

FT-CORBA servers can publish IOGRs to clients. Clients use these IOGRs to invoke operations on servers. The client-side ORB transmits the request to the appropriate server-side object that handles the request. The client application need not be aware of the existence of server object replicas. If a server object fails, the client-side ORB cycles through the object references contained in the IOGR until the request is handled successfully by a replica object. If all server objects fail, all the references in the IOGR are consider invalid and an exception is returned to the client application.

Figure 1 illustrates the key components in the FT-CORBA architecture. All components shown in the figure are implemented as standard CORBA objects, *i.e.*, they are defined using CORBA IDL interfaces and implemented using servants. The functionality of each component is described below:

ReplicationManager: This component is responsible for managing replicas and contains the following three components:

1. PropertyManager: Which allows properties of an object group to be selected. Common properties include the replication style, membership style, consistency style, and initial and maximum number of replicas. Examples of replication styles are ACTIVE, COLD_PASSIVE, and, WARM_PASSIVE. Membership of a group and data consistency of the group members can either be controlled by the FT-CORBA infrastructure or can be controlled by applications. The FT-CORBA specification standardizes both application controlled and infrastructure controlled membership and consistency styles.

2. GenericFactory: For the infrastructure-controlled membership style, the `ReplicationManager` uses the `GenericFactory` to create object groups and individual members of an object group.

3. ObjectGroupManager: For application-controlled memberships, applications use the `ObjectGroupManager` interface to create, add, or delete members of an object group.

Fault Detector and Notifier: `FaultDetectors` are CORBA objects responsible for detecting faults via a *pull-based* monitoring mechanism that periodically polls applications to determine if their objects are "alive." The FT-CORBA standard requires application objects to implement a `PullMonitorable` interface that exports an `is_alive` operation. In addition, other types of `FaultDetectors`, such as those supporting *push-based* monitoring mechanisms (also known as "heartbeats"), can be implemented. In this scheme, applications implement a `PushMonitorable` interface and send periodic heartbeats to the `FaultDetector`.

`FaultDetectors` report faults to `FaultNotifiers`, which propagate these notifications to a `ReplicationManager`, which performs recovery actions. In addition, other applications in the system that are interested in monitoring fault activity can register with the `FaultNotifiers` to receive such events. In addition, more complex applications can provide `FaultAnalyzers` to expand, correlate, condense, and analyze fault reports. The functionality provided by `FaultAnalyzers` is usually

[1] An IOR is a flexible addressing mechanism that uniquely identifies a CORBA object.

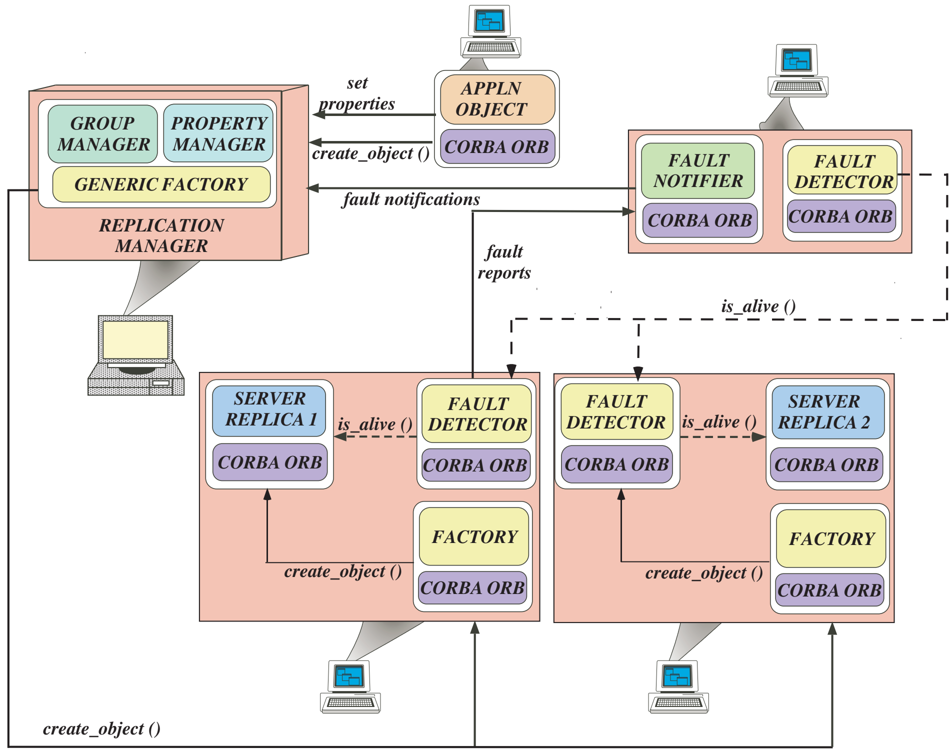

Figure 1: The Architecture of Fault Tolerant CORBA

platform- and application-specific, e.g., a sequence of fault reports can be correlated to identify a single failure condition.

Logging and Recovery: FT-CORBA defines a logging and recovery mechanism that is responsible for intercepting and logging CORBA GIOP messages from client objects to servers. Distributed applications can employ this mechanism via an infrastructure-controlled consistency style. In the event of a failure, a new replica is chosen to become the "primary" and the recovery mechanism executes the operations that were made by the client, but failed due to the primary replica's failure. In addition, it retrieves a consistent state for the new replica. The logging and recovery mechanism ensures that failovers are transparent to applications. For the application-controlled consistency style, it is the responsibility of the application to recover from failures.

The FT-CORBA standard is designed to prevent single points of failure within a distributed object computing system. As a result, each component described above must itself be replicated and a mechanism must be provided to deal with potential failures and recovery.

2.2 Comparing DOORS and FT-CORBA

As mentioned at the beginning of Section 2, DOORS is a precursor to the FT-CORBA standard that implements a core subset of the functionality defined by the FT-CORBA standard. Figure 2 illustrates the interaction of key components in the DOORS framework. As shown in the figure, DOORS has a `ReplicaManager` module that encapsulates most of the property mangement and the group management functionality proposed in the FT-CORBA specification. The `WatchDog` and the `SuperWatchDog` modules support hierarchical fault detection and notification. The `WatchDogs` perform object-level fault detection and `SuperWatchDogs` perform host-level fault detection.

In addition, `WatchDogs` and `SuperWatchDogs` act as `FaultNotifiers` and propagate fault reports to the `ReplicaManager`. `WatchDogs` support both push-based (heartbeats) and pull-based (polling) object monitoring. All FT-CORBA infrastructure modules (*i.e.*, the `Watchdog` and `ReplicaManager`) are monitored by a `SuperWatchdog` via heartbeats. Thus, through replication and monitoring of infrastructure modules, DOORS ensures there is no single point of failure in the system.

Table 1 gives the mapping of the current DOORS implementation to the FT-CORBA specification.

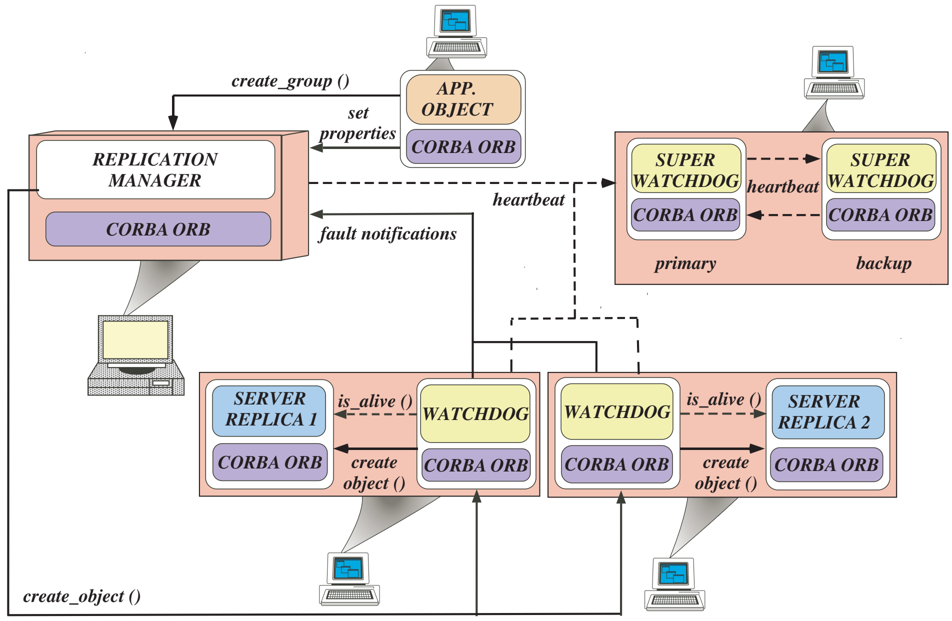

Figure 2: Components in the DOORS Architecture

Function	DOORS	FT-CORBA
Property Management	`ReplicaManager`	`ReplicationManager`
Group Creation/Management	`ReplicaManager`	`ReplicationManager`
Fault Detection	`WatchDogs/SuperWatchDogs`	`FaultDetector`
Fault Notification	`WatchDogs/SuperWatchDogs`	`FaultNotifier`
Logging/Recovery Policy	Application-controlled	Both Application and Infrastructure controlled
Group Creation Policy	Infrastructure controlled	Both Application and Infrastructure controlled

Table 1: Comparison of DOORS with Fault Tolerant CORBA

In summary, DOORS uses the service strategy to provide fault tolerance to CORBA applications. The DOORS model has been incorporated into the recently adopted Fault Tolerant CORBA standard. The service-level management modules of the FT-CORBA specification are derived directly from service-level management modules of DOORS, such as `ReplicaManagers` and `WatchDogs`.

2.3 Overview of the CORBA/DOORS Fault Tolerance Protocol

Figure 3 illustrates the protocol interactions between the components of the DOORS framework when an application uses a WARM_PASSIVE scheme. As shown in the figure, establishing a replica group involves the following steps:

1. An application manager asks the `ReplicaManager` to create a replica group and provides it with a set of properties the replicas will possess.
2. The `ReplicaManager` delegates the task of creating individual replicas to `WatchDogs`, which then start monitoring them.
3. Each `WatchDog` creates a replica with the desired properties.
4. The newly created replica registers its IOR with a Naming Service.
5. The `ReplicaManager` collects all the IORs of the individual replicas, creates a IOGR for the group, and designates one of the replicas as a primary.
6. The `ReplicaManager` checkpoints the IOGR and

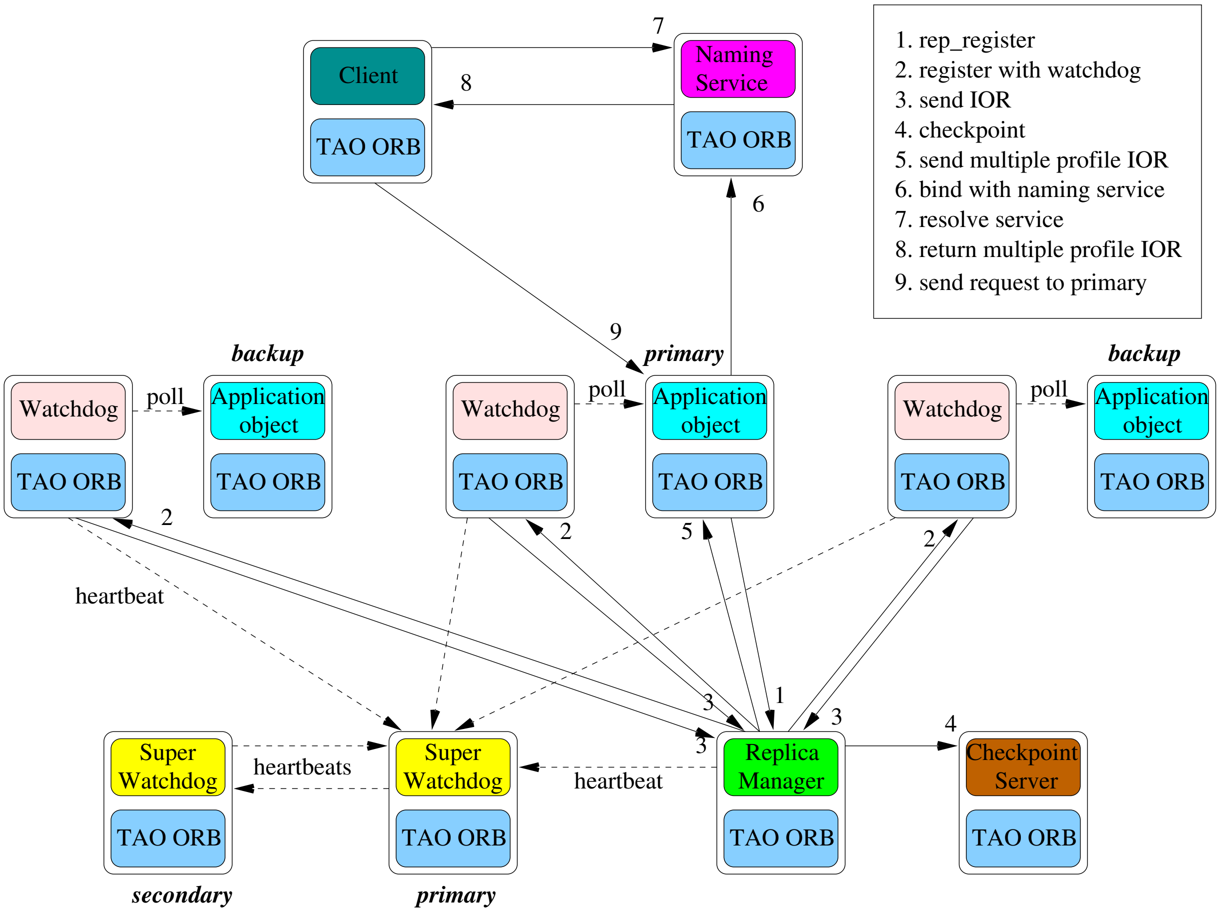

Figure 3: Interaction of Components in DOORS

other state.

7. The `ReplicaManager` then registers the IOGR with the Naming Service which is published to the outside world.

8. A client interested in the service contacts the Naming Service.

9. The Naming Service responds with the IOGR.

10. Finally, the client makes a request and the client-side ORB ensures that the request is sent to the primary replica.

Once a replica group is established, depending on the monitoring style chosen (*e.g.*, heartbeat or polling), the `WatchDog` will continue monitoring the replicas at periodic intervals. `WatchDogs` and `ReplicaManager` send heartbeats to a primary `SuperWatchDog` at periodic intervals. The `SuperWatchDog` itself is replicated since FT-CORBA does not allow single points of failures. The `SuperWatchDog` replicas send heartbeats to each other at periodic intervals. One such `SuperWatchDog` is designated as a primary and others are backups. If the primary fails, there is an election for a new primary amongst the other backups.

3 Benchmarking Fault Tolerant CORBA

As described in Section 2.3, the protocol used in the CORBA fault tolerance specification contains many interactions between FT-CORBA components. If designed and implemented improperly, this protocol can incur excessive overhead in distributed systems, which could make FT-CORBA unsuitable for applications in the telecommunications and real-time domain that require both fault tolerance *and* real-time QoS guarantees, such as efficiency, predictability, and scalability. Thus, for performance-sensitive applications to use FT-CORBA effectively, it must incur a negligible impact on distributed system QoS, so that applications can reap the benefits of fault tolerance, while operating within the designated QoS bounds.

This section describes the results of benchmarking studies we conducted to measure the impact of FT-CORBA on the performance of distributed applications. We have chosen a bandwidth-intensive application for our experiments to measure the impact of FT-CORBA on its throughput. We use the DOORS framework described in Section 2.2 as the FT-CORBA implementation in our benchmarks. The results presented in this paper are one of the first empirical benchmarks using the recently adopted Fault Tolerant CORBA specification.

A key goal in conducting these benchmarks is to quantify the effect of the heartbeat/polling interval on the (1) *response time* required for clients to connect to a new primary in the event of the failure of an existing primary, (2) *throughput* observed by a client using a DOORS-enabled service, and (3) *detection time* incurred by DOORS to detect failures.

Our experiments have all been conducted with a single-failure model assumption, *i.e.*, no failures occur while the system is recovering from a previous failure condition. Moreover, as discussed in the FT-CORBA specification [19], DOORS assumes that all failures are crash failures.

3.1 Overview of Benchmarking Testbed

This section describes our testbed environment and experimental setup.

Synopsis of hardware/software platform: The experiments described in this section were conducted on a 167 MHz UltraSPARC processor with 128 Mbytes of RAM running SunOS 5.7. We used the Sun C++ Workshop compiler 5.0 with the default optimization level specified by -O option. To improve the performance baseline, all executables were linked using static libraries.

To minimize network latency and jitter, our benchmarks focus primarily on running different service components of DOORS on one endsystem. Moreover, by running all components on a single endsystem, we can measure the worst-case overhead of FT-CORBA that stems from the polling and heartbeat messages generated by the infrastructure and application components.

The DOORS FT-CORBA framework ran on version 1.0.12 of the TAO [3] ORB. We chose TAO because it is a freely available, highly optimized, CORBA-compliant ORB suited for applications with stringent QoS requirements. In addition, it is the first ORB to implement ORB-level enhancements defined by the FT-CORBA standard. These enhancements include (1) support for the interoperable object group reference (IOGR) described in Section 2 and (2) the ability of the client-side ORB to failover transparently from one IOR to other within an IOGR without the client application being aware of this failover. Without these ORB features, it would not be possible to experiment with *transparent* client-side failover and recovery.

Synopsis of performance benchmarks: Our benchmarking testbed is designed to quantify the following metrics:

Fault detection time: which measures the time taken by DOORS to detect faults for different heartbeat/polling interval values. This latency depends on the rate of the `is_alive` ping calls (Figure 3 Step 3) that a `WatchDog` sends to a server replica object.

Recovery time: which measures the time taken by the DOORS framework to establish a new primary if a primary replica fails for different polling intervals. This latency approximates the amount of overhead DOORS incurs when it performs failure recovery on behalf of a server object. It also indicates the amount of time that DOORS needs to get back to a stable state.

Throughput variation: which measures the variation in throughput for different heartbeat/polling intervals.

IOGR cycling time: which measures the time required by the client-side ORB to cycle through the profiles of an IOGR. When a connection breaks, the FT-CORBA model specifies that the client-side ORB should catch and handle the `CORBA::COMM_FAILURE` exception and propagate the same request to the next available server in the IOGR. We measure the latency for the client-side ORB to establish a new connection.

Synopsis of experimental application configuration: Our experimental testbed is based on a CORBA-based load balancing service provided by TAO. This service balances the load on a group of servers by forwarding the requests to different servers in the group according to a particular selection property, such as round-robin or randomized. We adapted the TAO load balancing service to run under the DOORS framework. Moreover, we added a `shutdown` operation to the service that enables an administrator to simulate object crashes.

We selected the load balancing service for our benchmark because requirements for fault tolerance in large-scale distributed systems are often associated with a need for load balancing. To minimize network delay and jitter, we ran the experiments by registering two copies of the load balancing service with a `FaultDetector` (*i.e.*, `WatchDog`) on the same host. This `FaultDetector` monitored the load balancing service objects using the `PullMonitorable` (*i.e.*, polling) style of fault-detection with a WARM_PASSIVE replication style.

In the WARM_PASSIVE replication style, the replica group contains a single primary replica that responds to client messages. In addition, one or more backup replicas are defined to handle crash failures. If a backup receives a client request, it responds with a standard GIOP LOCATION_FORWARD message to the client indicating the address of the current primary to which the client must send requests. If a primary fails, however, a backup replica is selected to function as the new primary and a new backup is created to maintain a constant replica group size. The recovery mechanism simply restarts the load balancing object.

The DOORS framework focuses on the passive replication style, *i.e.*, only one copy of the replica acts as a primary. In our experiments we did *not* test the COLD_PASSIVE replication style since it is a degenerate case of WARM_PASSIVE where only the primary replica runs and all the backups are dormant. In addition, most applications tend to use the WARM_PASSIVE replication scheme for fault tolerance.

3.2 Measuring Fault Detection and Recovery Times

Below, we describe experiments conducted to measure the fault detection and recovery times experienced by clients.

Methodology: To measure the effect of failures, as well as to compute the total recovery time, we allowed the clients to connect to the primary replica. After clients connected to the primary replica, we terminated the primary replica by invoking the server object's `shutdown` operation. For the client-side failover measurement, we start a timer in the client when `shutdown` is invoked while the client-side ORB continues to invoke subsequent requests to the primary replica.

After the primary is terminated, the client-side ORB receives an exception. At this point, it begins cycling through the other IORs that make up the IOGR, sending a request to each IOR. Until a new primary is selected, however, the client-side ORB continues to receive a `location_forward` exception from the backups informing the client-side ORB to send the request to the dead primary.

When the DOORS `FaultDetector` detects a failure it reports this failure to the `ReplicaManager`, which then selects a backup copy amongst the replicas and promotes it to become the new primary. Simultaneously, the `ReplicaManager` creates a new backup to maintain a constant replica group size. It then notifies the new primary of its change in status by invoking the `become_primary` operation. This operation enables the new primary to respond to client requests. At this point, we stop our client-side timer and compute the failover time observed by the client.

Fault detection and recovery time measurements: We measured the recovery time as outlined above for two different scenarios. The first scenario set the interval between successive pings to the server object to be the same as the global value of the heartbeat interval. The global value of the heartbeat interval is the time period in which the different components of DOORS send heartbeats to its immediate monitors. The recovery times measured for this benchmark are shown in Table 2.

In another scenario, we reduced the time interval between successive `WatchDog` calls to the server object to be half the global heartbeat interval value. The recovery times obtained for this case are shown in Table 3. These two measurements illustrate that by reducing the interval between successive `is_alive` calls will not necessarily yield better recovery times. Although different implementations may have different threshold values, there is a lower limit to the recovery times.

Heartbeat Interval (sec)	Recovery Time (sec)		
	Min	Avg	Max
5	6.0482	6.0553	6.06417
10	11.3877	11.5153	11.7877
15	16.5469	16.8965	17.0876
20	20.0058	20.0373	20.0868
25	25.0162	25.0321	25.0483
30	30.0136	30.4668	30.5119

Table 2: Effect of Different Polling Intervals on Recovery Times

Heartbeat Interval	Recovery Time (sec)		
	Min	Avg	Max
5	6.0483	6.0557	6.0864
10	6.9520	7.0560	7.0878
15	8.0629	8.0540	8.0967
20	11.0845	11.3567	11.9876
25	14.0382	14.0580	14.2041
30	19.0323	19.2350	19.4529

Table 3: Effect of Different Polling Intervals on Recovery Times

These results underscore the importance of choosing the correct heartbeat interval to optimize performance. For instance, a large heartbeat interval yields long detection and recovery times. Conversely, a short heartbeat can yield false alarms. As an exmaple, our experiments showed that reducing the heartbeat interval to 2 seconds caused the `SuperWatchdog` to report a false failure of its `ReplicaManager`. This false failure caused the `SuperWatchDog` to erroneously launch a new instance of the `ReplicaManager`.

3.3 Measuring Throughput

Methodology: To compare the performance of FT-CORBA application with the performance of the same application that did not use FT-CORBA, we created a load balancing service configuration that did not use DOORS. Next, we flooded this service with 100,000 requests, transferring 1,024 bytes of data per request. We then measured throughput as the number of calls per second that was processed by the server. To measure the impact of FT-CORBA on the performance, we configured the same service with DOORS and the same flooding traffic pattern as before.

Throughput measurements: The client throughput baseline measured for a non-DOORS server was ~700 events/sec. Table 4 shows the throughput of a DOORS-enabled server when the polling interval equals the heartbeat interval.

Table 4 illustrates that the throughput of the DOORS-enabled server decreases by only a small amount (6%).

Heartbeat Interval	Throughput (events/sec)
5	656.30
10	657.82
15	662.42
20	664.65
25	668.96
30	672.24

Table 4: Effect of Different Polling Intervals on Throughput

3.4 Measuring Client-side ORB's IOGR Cycling Time

In DOORS, for each group of n replicas the `ReplicaManager` creates an IOGR with $n+1$ object references. The last profile contains a reference to the `ReplicaManager`, with the object group's identifier encapsulated inside the reference. When an IOGR is published to the client, the client starts making calls on the IOGR. With every failure, the `ReplicaManager` starts a new backup as a corrective action. At that instant the IOGR held by the client becomes a "stale" object reference because the constituents of the groups have changed. After all replicas in IOGR are used, the client ORB uses the last reference to make a call on the `ReplicaManager`, which then uses the GIOP LOCATE_FORWARD mechanism to forward the latest IOGR to the client ORB.

Methodology: To measure the client-side ORB's IOGR cycle time, we wrote a simple server application and launched multiple copies of this server. A manager object created IOGR as follows: IOR_1 consists of object references from $server_1$, $server_2$, $server_3$, and a reference to new IOR_2. IOR_2 is made from the object references of $server_4$ and itself. The reason we append IOR_2 is to simulate the role played by the `ReplicaManager`. Finally, this IOGR is published to the client and the time required by the client to make calls on every IOR within the IOGR was then measured. To ensure that the client ORB selects the next IOR in the IOGR, we killed the primary by the `shutdown` call.

IOGR profile cycling time for the TAO ORB: The time taken by the client ORB to cycle through all the IOR in the IOGR is shown in Table 5. The results in this table are explained below:

Profile index in a Multiple profile IOR	Time to switch (nanosecs)
First profile	23935173
Second profile	27241314
Third Profile	27221948
Replica Manager	46927928

Table 5: IOGR Profile Cycle Time

- The data presented for the first profile indicates the latency experienced by a newly launched client to make the first call on the server. We measure this time to determine the overhead TAO incurs to establish connections. After connections are established, TAO has many performance optimizations that improve its throughput and latency on the critical data path [11]. However, these optimizations do not necessarily improve its connection establishment time. Therefore, we measure this overhead to determine the time required to reconnect to the each IOR profile in an IOGR.
- The data for the second profile indicates the time required by the ORB to detect a failure, handle a COMM_FAILURE exception at the application level, and re-issue the request.
- The data for the third profile was also measured for the whole cycle outlined above. After a primary fails, TAO holds a reference to the next possible target address. When the second profile fails, then TAO shifts over to the next possible target address. It also marks the targets on which it has not been able to establish connections. So, it is only natural that the time as observed by the client would be the same for successive failures till it reaches the IOR of RM.
- The data for the fourth profile (`ReplicaManager`) shows the latency experienced by the client when third IOR fails. This latency includes detection of client-side failures, re-issuing of the request after handling the exception, the server sending a LOCATION_FORWARD IOR to the client, and the client re-invoking a request on the new server's IOR.

3.5 Measuring Failure Detection Time

Methodology: We define the failure detection time as the time take by the detector *i.e.*, the `WatchDog`, to identify a failure. As described earlier, the `WatchDog` polls the server object at constant intervals of time. Therefore, the time difference between the failure of the primary and the time difference when the `WatchDog` actually detects a failure is considered the *failure detection time*.

In general, the failure detection time is not a constant value for the FT-CORBA infrastructure or an application since it depends on the type and rate of failures an application encounters. Moreover, failure detection time depends on the polling interval. The *recovery overhead* is computed as an average estimate of the detection time subtracted from the recovery time experienced by the client.

Failure detection time in DOORS: We measured the failure detection times by recording two time stamps. The first time stamp was recorded when the primary was killed. The second was recorded when the `WatchDog` detects a failure. Table 6 shows the average failure detection times during our experiments for various polling intervals.

Polling Interval	Average Fault Detection Time (sec)
2.5	1.0492
5	2.04612
7.5	3.2775
10	5.6023
12.5	7.4922
15	12.0910

Table 6: Effect of Different Polling Intervals on Recovery Times

3.6 Measuring Time-to-Stability

Methodology: Beyond detecting failures and taking corrective actions, a FT-CORBA infrastructure must perform additional activities, including (1) starting new servers on the nodes where they have failed, (2) migrating backups to a different node because of a node crash, and (3) informing clients of change in object references. The latter activity can be complex since the FT-CORBA specification strives to make system failures transparent to clients. In particular, after a client obtains an object reference, it can invoke calls on this reference repeatedly and if there are changes to the references, the FT-CORBA infrastructure is responsible for performing any necessary redirections to replicas.

Thus, it is necessary to determine this type of overhead on the fault tolerant system since it affects the time the system requires to return to a stable state. Each fault tolerant implementation has a time period to attain stability after a failure for every replication style that it supports. Determining the time-to-stability is important if a system experiences new faults before stability is restored because any intervening error detection and recovery activities may become erroneous.

Time-to-stability measurements: We determine the time-to-stability in DOORS by measuring the time required to (1) promote a backup to a primary after detecting a fault, (2) start a new back up copy, (3) update our internal reference list, and (4) update the Naming service with new group IOR (IOGR) that we create. The sum of all these reveals DOORS' time-for-stability. Table 7 shows the time-for-stability in DOORS. This table illustrates that DOORS takes on an average 17 seconds to regain stability after a failure. It time-to-stability is relatively constant, which indicates that this overhead does not depend on the polling intervals.

Polling Interval	Primary Setup Time (sec)		
	Min	Avg	Max
2.5	15.1748	15.3616	16.3818[1]
5.0	15.1568	15.2295	18.2780
7.5	15.157345	15.3421	17.3605
10.0	15.161933	16.6101	19.5751
12.5	15.168172	15.3371	17.2783
15.0	15.153734	15.2125	17.2638

Table 7: Variation of the Time-to-Stability After a Failure for Different Polling Intervals

4 Related Work

CORBA is increasingly being adopted as the middleware of choice for mission-critical, large-scale, and heterogeneous distributed systems. Thus, the need to develop highly reliable CORBA applications has motivated research on policies and mechanisms for fault-tolerant CORBA. Recent research efforts [23, 16, 24, 25, 26] devoted to enhancing the fault tolerance of CORBA ORBs and/or CORBA applications can be categorized into three strategies: an *integration* strategy, an *interception* strategy, and a *service* strategy [23].

The *integration* strategy layers the ORB on top of a reliable multicast communication subsystem. Examples of this approach include Orbix+Isis [24, 13] and Electra [24, 16]. Orbix+Isis was the first commercially available system that supported fault-tolerant CORBA-compliant applications. The fault-tolerance of server implementations is achieved using active (hot) replication, with reliable multicast support from low-level group communication system [13]. Electra was another early implementation of reliable CORBA. It differed from Orbix+Isis mainly with respect to its ease of use and its adaptability to other communication subsystems, such as Horus and Ensemble. However, the integration strategy requires modifications to ORB interfaces and language mappings, which reduces the portability of server and client applications.

The *interception* strategy was pioneered by the Eternal system [23, 27] to construct a reliable ORB environment. Eternal intercepts system calls made by the client object through the lower-level I/O subsystem and maps these system calls to the reliable multicast subsystem. Eternal differs from Orbix+Isis or Electra in that Eternal does not modify the ORB or the language mapping. Thus, it ensures the transparency of fault-tolerance from applications. The AQuA framework [18] also provides a variant of the interception approach. The AQuA gateway acts as an intermediary between the CORBA objects and the Ensemble group communication subsystem and translates GIOP messages to group communication primitives. In addition, AQuA uses the Quality Objects (QuO) [28] framework to allow applications to specify their reliability/availability requirements.

The Maestro toolkit [29] developed at Cornell University, uses a combination of interception and integration approaches. It provides an object-oriented wrapper facade [30] to process-based group communication systems and supports transparent client-side failover through the use of multi-profile object references. The process group exports a multi-profile IOR to the client and the client uses a single profile in that IOR to connect to the primary of the group through an IIOP gateway.

The *service* strategy defines a set of objects and their interfaces to provide fault tolerance [25, 26]. For example, OGS [25] provides a group of services, including membership service, monitoring service, consensus service and messaging service, that work together to enhance the reliability of an application through both active and passive replication. The DOORS framework described in this paper is another example of a service-based approach to fault tolerance.

The research efforts outlined above led to several different strategies to fault tolerance that were not interoperable and did not provide portability to CORBA applications. To address this problem, therefore, the OMG initiated a proposal to standardize Fault-tolerant CORBA (FT-CORBA) and the lessons learned from the varied research prototypes were realized recently in the FT-CORBA standard [19].

5 Concluding Remarks

A growing number of CORBA applications with stringent QoS requirements also require fault tolerance support. The most flexible strategy for providing fault tolerance to CORBA applications is through higher-level CORBA services. To address the fault tolerance requirements, the OMG recently standardized the Fault Tolerant CORBA (FT-CORBA) specification. For FT-CORBA to be useful to applications with stringent QoS requirements, however, it should cause negligible impact on their performance. This paper describes a fault tolerant CORBA framework called DOORS that uses the service approach and whose model is incorporated in the FT CORBA standard.

Our benchmarking studies to measure the impact of using DOORS for bandwidth-intensive applications reveal only a 6% degradation in performance. Moreover, DOORS allows applications to customize the heartbeat and polling intervals to enable them to optimize performance, as well as achieve the requisite levels of fault tolerance.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 edition, June 1999.

[2] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, no. 2, February 1997.

[3] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[4] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[5] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997, ACM.

[6] Fred Kuhns, Douglas C. Schmidt, and David L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, Vancouver, British Columbia, Canada, June 1999, IEEE, pp. 154–163.

[7] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, Apr. 2000.

[8] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

[9] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, Apr. 2000.

[10] Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, Stanford, CA, August 1996, ACM, pp. 306–317.

[11] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999, USENIX.

[12] Kenneth Birman, "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, December 1993.

[13] Kenneth Birman and Robbert van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, 1994.

[14] Veritas, "Veritas FirstWatch," www.veritas.com/us/products/firstwatch, 2000.

[15] Yennun Huang and Chandra Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," in *23rd International Symposium on Fault-tolerance Computing (FTCS)*, Toulouse, France, June 1993, pp. 2–10.

[16] Silvano Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, Monterey, CA, June 1995, USENIX.

[17] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, vol. 32, no. 7, pp. 64–68, July 1999.

[18] 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," in *IEEE Symposium on Reliable and Distributed Systems (SRDS)*, West Lafayette, IN, Oct. 1998, pp. 245–253.

[19] Object Management Group, *Fault Tolerant CORBA Specification*, OMG Document orbos/99-12-08 edition, December 1999.

[20] W3C HTTP-NG Working Group, "W3C HTTP-NG Protocol," www.w3.org/Protocols/HTTP-NG, 1998.

[21] Object Management Group, *Fault Tolerance CORBA Using Entity Redundancy RFP*, OMG Document orbos/98-04-01 edition, April 1998.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[23] Priya Narasimhan, *Transparent Fault Tolerance for CORBA*, Ph.D. thesis, University of California, Dept. Of Electrical and Computer Engineering, Santa Barbara, CA, Dec. 1999, Available as Technical Report UCSB 99-18.

[24] Sean Landis and Silvano Maffeis, "Building Reliable Distributed Systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 31–43, 1997.

[25] Pascal Felber, Rachid Guerraoui, and André Schiper, "The implementation of a corba object group service," *Theory and Practice of Object Systems (TAPOS)*, vol. 4, no. 2, pp. 93–105, Feb. 1998.

[26] J. Schonwalder, S. Garg, Y. Huang, A.P.A. van Moorsel, and S. Yajnik, "A Management Interface for Distributed Fault Tolerant CORBA Services," in *IEEE International Workshop on System Management*, Newport, RI, Apr. 1998, pp. 98–107.

[27] Louise Moser, P. Melliar-Smith, and Priya Narasimhan, "A Fault Tolerance Framework for CORBA," in *International Symposium on Fault Tolerant Computing*, Madison, WI, June 1999, pp. 150–157.

[28] John A. Zinky, David E. Bakken, and Richard Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[29] Alexey Vaysburd and Ken Birman, "The maestro approach to building reliable interoperable distributed applications with multiple execution styles," *Theory and Practice of Object Systems (TAPOS)*, vol. 4, no. 2, pp. 73–80, Feb. 1998.

[30] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*, Wiley & Sons, New York, NY, 2000.