Applying Patterns to Improve the Performance of Fault Tolerant CORBA

Balachandran Natarajan
bala@cs.wustl.edu
Dept. of Computer Science

Washington University
One Brookings Drive
St. Louis, MO 63130

Aniruddha Gokhale, Shalini Yajnik
{agokhale, shalini}@lucent.com
Bell Laboratories

Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974

Douglas C. Schmidt
schmidt@uci.edu
Dept. of Electrical
and Computer Engineering
University of California
616E Engineering Tower
Irvine, CA 92697

This paper has been submitted as an invited paper to the 7^{th} International Conference on High Performance Computing, ACM/IEEE, Bangalore, India, Dec. 17-20, 2000.

Abstract

An increasing number of mission-critical, embedded, telecommunications, and financial distributed systems are being developed using distributed object computing middleware, such as CORBA. Applications for these systems often 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 many of these QoS requirements the recent Real-time CORBA and Fault Tolerant CORBA specifications.

This paper provides two contributions to the study and design of CORBA middleware that provides multiple QoS properties. First, we describe results of experiments conducted to measure the performance of a fault-tolerant CORBA services framework called DOORS and illustrate how common implementation pitfalls can adversely affect its performance. Second, we describe the patterns we are incorporating into the DOORS fault-tolerant CORBA service to simultaneously improve its performance and fault-tolerance.

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 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 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]. ORB endsystems consist of network interfaces, operating system I/O subsystems, CORBA ORBs, and higher-level CORBA services.

Addressing middleware research challenges with patterns: Our prior research on CORBA middleware has explored many 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], optimization principle patterns for ORB performance [11], and measuring performance of a CORBA fault-tolerant service [12]. This paper focuses on another dimension in the ORB endsystem design space: *applying patterns to improve the performance of Fault Tolerant CORBA (FT-CORBA) implementations*.

A *pattern* names and describes a recurring solution to a software development problem within a particular context [13]. Patterns help to alleviate the continual re-discovery and reinvention of software concepts and components by documenting and teaching proven solutions to standard software development problems. For instance, patterns are useful for documenting the structure and participants in common communication software micro-architectures like *Active Objects* [14] and Brokers [15]. These patterns are generalizations of object-structures that have been used successfully to build flexible, efficient, event-driven, and concurrent communication software, including ORB middleware.

In general, patterns can be categorized as follows:

Design patterns: A design pattern [13] captures the static and dynamic roles and relationships in solutions that occur repeatedly when developing software applications in a particular domain. The design patterns we apply to improve the performance of FT-CORBA include: *Abstract Factory, Active Object, Chain of Responsibility, Component Configurator, and Strategy*.

Architectural patterns: An architecture pattern [14] expresses a fundamental structural organization schema for software systems that provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. The architectural patterns we apply to improve the performance of FT-CORBA include: *Leader/Followers and Reactor*.

Optimization principle patterns: An optimization principle pattern [11] documents rules for avoiding common design and implementation mistakes that degrade the performance, scalability, predictability, and reliability of complex systems. The optimization principle patterns we applied to improve performance of FT-CORBA include: *optimizing for the common case, eliminating gratuitous waste, and storing redundant state to speed up expensive operations*.

Paper organization: The remainder of this paper is organized as follows: Section 2 summarizes the recently adopted Fault Tolerant CORBA (FT-CORBA) specification. Section 3 describes empirical benchmarks we conducted to compare the performance of a load-balancing application implemented with and without using DOORS, which is our FT-CORBA middleware; Section 4 describes the patterns we are using to improve the performance of the DOORS FT-CORBA service; and Section 5 presents concluding remarks.

2 Overview of the Fault Tolerant CORBA Specification and DOORS

The Fault Tolerant CORBA (FT-CORBA) [16] specification defines a standard set of interfaces, policies, and services that provide robust support for applications requiring high reliability. The fault tolerance mechanism used to detect and recover from failures is based on *entity redundancy* [17]. Naturally, in FT-CORBA the redundant entities are replicated CORBA objects.

Replicas of a CORBA object are created and managed as a "logical singleton" [13] composite object. Figure 1 illustrates the key components in the FT-CORBA architecture. All

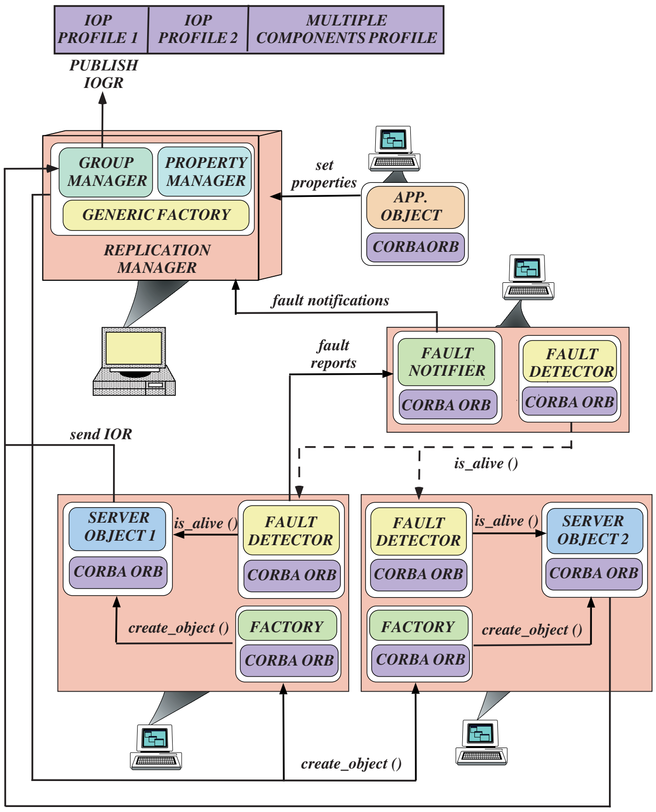

Figure 1: The Architecture of Fault Tolerant CORBA

components shown in the figure are implemented as standard CORBA objects, *i.e.*, they are defined using CORBA IDL interfaces and implemented using servants that can be written in standard programming languages, such as Java, C++, C, or Ada. The functionality of each component is described below.

Interoperable object group references (IOGRs): FT-CORBA standardizes the format of interoperable object references (IOR) used for the individual replicas. An IOR is a flexible addressing mechanism that identifies a CORBA object uniquely [1]. In addition, it 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. The references in the IOGR are considered invalid only if all server objects fail, in which case an exception is propagated to the client application.

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

1. PropertyManager: This component allows properties of an object group to be selected. Common properties include the replication style, membership style, consistency style, and initial/minimum number of replicas. Example replication styles include the following:

- COLD_PASSIVE – In this replication style, the replica group contains a single primary replica that responds to client messages. If a primary fails, a backup replica is spawned on-demand to function as the new primary.
- WARM_PASSIVE – 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 pre-spawned to handle crash failures. If a primary fails, a backup replica is selected to function as the new primary and a new backup is created to maintain the replica group size constant.
- ACTIVE – In the ACTIVE replication style all replicas are primary and handle client requests independently of each other. To ensure a single reply sent to the client and to maintain consistent state amongst the replicas, a special group communication protocol is necessary.

Membership of a group and data consistency of the group members can be controlled either by the FT-CORBA infrastructure or by applications. FT-CORBA 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 either a *pull-based* or a *push-based* mechanism. A *pull-based* monitoring mechanism periodically polls applications to determine if their objects are "alive." FT-CORBA requires application objects to implement a `PullMonitorable` interface that exports an `is_alive` operation. A *push-based* monitoring mechanism can also be implemented. In this scheme, which is also known as a "heartbeat monitor," applications implement a `PushMonitorable` interface and send periodic heartbeats to the `FaultDetector`.

`FaultDetectors` report faults to `FaultNotifiers`. In turn, the `FaultNotifiers` 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 their events. More complex applications can provide `FaultAnalyzers` to expand, correlate, condense, and analyze fault reports. The functionality provided by `FaultAnalyzers` is usually 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. If a failure occurs, a new replica is chosen to become the "primary." The recovery mechanism then re-invokes the operations that were made by the client, but which did not execute 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, applications are responsible for their own failure recovery.

FT-CORBA 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. Moreover, mechanisms must be provided to deal with potential failures and recovery.

3 Benchmarking the Performance of DOORS Fault Tolerant CORBA

This section describes the results of benchmarking studies we conducted to measure the impact of FT-CORBA on distributed application performance. All tests were run using the *Distributed Object-Oriented Reliable Services* (DOORS), which is our implementation of the FT-CORBA specification. We have chosen a bandwidth-intensive application for our experiments to quantify the impact of FT-CORBA on its performance and availability. The results presented in this paper are one of the first empirical benchmarks of an implementation of the OMG Fault Tolerant CORBA specification, which was adopted recently.

A key goal in conducting these benchmarks is to quantify the effect of the heartbeat/polling interval on the following:

1. The *detection time* incurred by DOORS to detect failures;
2. The *response time* required for clients to connect to a new primary in the event of the failure of an existing primary;

Our experiments assumed a single-failure model, where no nested failures can occur while the system is recovering from a previous failure condition. Moreover, DOORS assumes that all faults arise from object or host crashes, in accordance with the FT-CORBA specification [16]. In particular, the FT-CORBA specification does *not* support:

- *Network partitioning faults* where objects and hosts become unreachable although they themselves may not have crashed;
- *Commission faults* where objects or hosts generate incorrect results; or
- *Design or programming faults* that are caused by poor design or programming errors.

3.1 Overview of Benchmarking Testbed

This section outlines our testbed environment and experimental setup.

Synopsis of hardware/software platform: To minimize network latency and jitter, our benchmarks focus primarily on running different service components of DOORS on a single endsystem–a 167 MHz UltraSPARC processor with 128 Mbytes of RAM running SunOS 5.7. Running all components on one endsystem allows us to measure the worst-case overhead of FT-CORBA that stems from the polling and heartbeat messages generated by the infrastructure and application components. We used the GNU EGCS compiler with the default optimization level specified by the `-O` option. To improve the performance baseline, all executables were linked using static libraries.

The DOORS FT-CORBA framework ran on version 1.1 of the TAO [3] ORB. We chose TAO because it is an open-source, highly optimized, CORBA-compliant ORB suited for applications with stringent QoS requirements. In addition, it is the first ORB to implement the following ORB-level enhancements defined by the FT-CORBA standard:

- Support for the interoperable object group reference (IOGR) described in Section 2; and
- 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.

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

1. 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 at which a `FaultDetector` sends `is_alive` ping calls to a server replica object.

2. Recovery time, which measures the time taken by the DOORS framework to establish a new primary if a primary replica fails. We measured the recovery time 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 re-establish a stable system state.

Synopsis of experimental application configuration: Our experimental testbed is based on TAO's load balancing service. This service balances the load on a group of CORBA 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's FT-CORBA implementation. Moreover, we added a `shutdown` operation to the service so that we could simulate object crashes.

We selected the load balancing service for our benchmark because requirements for fault tolerance in large-scale distributed systems often necessitate load balancing. To minimize network delay and jitter, we ran the experiments by registering two copies of the load balancing service with a `FaultDetector` on the same host. This `FaultDetector` monitored the load balancing service objects using the `PullMonitorable` (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 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 practice, most production applications use the WARM_PASSIVE replication scheme for fault tolerance.

3.2 Measuring Failure Detection Time on the Server-side

Rationale: We define the failure detection time on the server-side as the time taken by the fault detector *i.e.*, the

`FaultDetector`, to detect a failure. In general, failure detection time is non-constant 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.

Methodology: A manager program requests the `Replica Manager` to start the replicas. After the replicas are running, the `FaultDetectors` begin polling them at constant intervals of time. We then allow a client to connect to the primary replica. At this point, we invoke the server object's `shutdown` operation, which initiates the fault detection process in the `FaultDetector`.

We measure the failure detection time as the time between the failure of the primary replica and the time when the `FaultDetector` actually detects a failure. To measure the failure detection time, we recorded two time stamps:

1. The first time stamp was recorded when the primary was killed.
2. The second timestamp was recorded when the `FaultDetector` detects the failure of the primary.

We conducted several iterations of this experiment by killing the primary replica at randomly selected times. To measure the impact on fault detection times, however, we killed the replica only after allowing the system to stablize from the previous fault. We allow the system to stabilize before injecting a new fault since DOORS–like most fault tolerance frameworks–cannot handle *nested failures*, *i.e.*, additional failures that occur while failure recovery is in progress.

This single recovery restriction stems from the highly complex nature of handling additional faults while the system is recovering and stabilizing itself from previous failures. It does not, however, preclude DOORS from handling faults for entities not directly involved in the recovery process. Moreover, as explained in Section 4, we are continually optimizing DOORS to reduce its recovery and stabilization overhead.

Failure detection time in DOORS: Figure 2 shows the minimum, average, and maximum failure detection times for different polling intervals.

Analysis: Figure 2 indicates the following:

- As the polling interval increases, the failure detection time also increases.
- On average, the failure detection time is half the polling interval.
- The best-case failure detection occurs when the primary is killed just before the next poll message is sent by the detector to the killed replica.

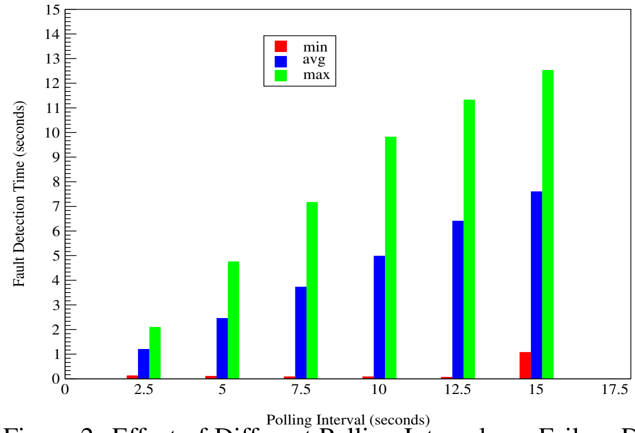

Figure 2: Effect of Different Polling Intervals on Failure Detection Times

- The worst-case behavior is depicted when the primary is killed just after a poll message is sent by the detector to the killed replica.

These results suggest that for applications requiring high availability, the polling intervals should be minimized. Overly small polling intervals increase the number of messages in the network, however, which may be problematic over low-speed network links.

3.3 Measuring Fault Detection and Recovery Times on the Client-side

Rationale: A client invoking a remote operation will experience some delay if its server fails during the operation. This delay has two parts:

1. The time taken by the infrastructure to detect the fault; and
2. The time taken by the infrastructure to promote a backup to become the primary.

Below, we describe experiments conducted to measure the combination of these times, which is the actual delay experienced by a client.

Methodology: As described in Section 3.2, to measure the effect of failures, and to compute the total recovery time, we allow clients to connect to the primary replica. After clients connect to the primary replica, we terminate 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 the `shutdown` operation is invoked.

When the DOORS's `FaultDetector` detects a failure it reports this failure to the `ReplicationManager`. In turn, the ReplicationManager selects a backup copy amongst the replicas and promotes it to become the new primary. Simultaneously, the ReplicationManager creates a new backup to maintain a consistent replica group size. It then notifies the new primary of its change in status by invoking the primary's become_primary operation, which 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. We set the polling interval to be half 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 their immediate monitors. The recovery times measured for this benchmark are shown in Figure 3.

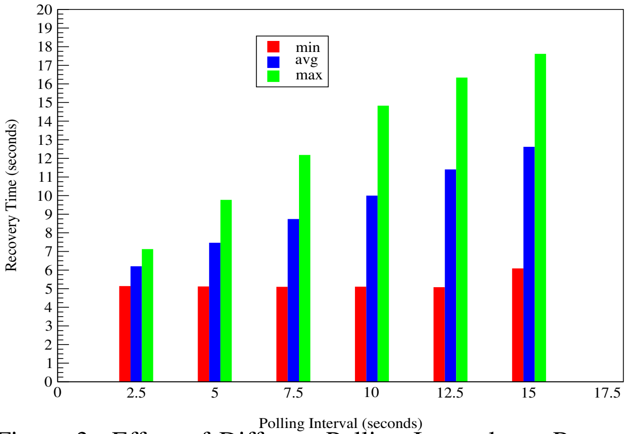

Figure 3: Effect of Different Polling Intervals on Recovery Times

Analysis: The results depicted in Figure 3 illustrate the client's observed recovery time, which comprises:

1. The *fault detection time*, which in the average case is roughly half the polling interval, as shown in Section 3.2 and illustrated by Figure 2; and
2. The *replica group management time*, which is the time to elect a new backup as the primary and start a new backup to maintain the replica group size constant.

Subtracting the failure detection time (shown in Figure 2) from the total recovery time (shown in Figure 3) reveals that maintaining a constant replica group size in the event of a failure requires ∼5 seconds in the best, worst, and average cases, irrespective of the polling interval. These results indicate that there is a lower bound on maintaining a constant replica group size after failure has occurred.

Our results underscore the importance of choosing the correct heartbeat/polling interval to optimize performance. For instance, a large heartbeat/polling interval yields long detection and recovery times. Conversely, a short heartbeat can yield false failure reports. As an example, our experiments showed that by reducing the shortest heartbeat interval from 5 seconds to 2 seconds, the DOORS' SuperWatchDog, which monitors the FaultDetectors for failures, reports a false failure of its ReplicationManager, which causes the SuperWatchDog to erroneously launch a new instance of the ReplicationManager.

3.4 Measuring Time-to-Stability

Rationale: In addition to detecting failures and taking corrective actions, FT-CORBA middleware must perform the following additional activities:

1. Start new servers on the nodes where they have failed;
2. Migrate backups to a different node due to a node crash; and
3. Inform clients of changes to object references.

Activity 3 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. If there are changes to the references, the FT-CORBA middleware is responsible for performing any necessary redirections to the replicas. Thus, it is necessary to determine the overhead on the fault tolerant system since it affects how long the system requires to re-establish a stable state.

Each FT-CORBA fault tolerant implementation has a time period to attain stability after a failure for every replication style that it supports. This "time-to-stability" is the elapsed time from the point of detection of a failure by the FaultDetector, to the point where the ReplicationManager has brought the system into a stable state by updating all the internal tables and states. Determining this time-to-stability is important since any intervening error detection and recovery activities may be erroneous if a system experiences new faults before stability is restored.

Methodology: We determine the time-to-stability in DOORS by measuring the time required to perform the following activities:

1. Promote a backup to a primary after detecting a fault;
2. Start a new backup copy;
3. Update the ReplicationManager's internal tables that hold information such as fault tolerance properties and IOGR of all the managed object groups; and

4. Update the Naming Service with the new interoperable object group reference (IOGR) that we create.

The sum of all these reveals DOORS' time-for-stability. Figure 4 shows the time-for-stability in DOORS.

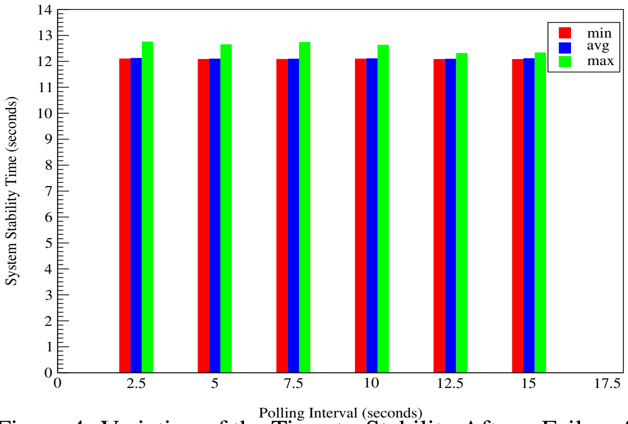

Figure 4: Variation of the Time-to-Stability After a Failure for Different Polling Intervals

Analysis: Figure 4 demonstrates that DOORS's time-to-stability is relatively constant, which indicates that its overhead does not depend on the polling intervals. We also observed during our experiments that if a failure occurs during the recovery phase the `FaultDetector` missed polling cycles because the thread that detects failure of a server also performs the recovery. Thus, after detecting the failure it starts the recovery phase and is not ready to poll the server object and any failures during this phase go undetected.

In general, however, a fault tolerant framework must incur a small time-to-stability after recovering from a fault. For applications that require 24×7 availability, the stabilization intervals incurred by DOORS may be unacceptable. To pinpoint the sources of overhead in our time-to-stability operations, we analyzed the DOORS code and determined the following activities contribute to this overhead:

1. The `ReplicationManager` finds the entry for the failed replica from its internal tables.
2. It then obtains a reference to the `FaultDetector` which reported it the failure from the Naming Service and then requests the `FaultDetector` to de-register the replica.
3. Next, the `ReplicationManager` determines if the failed object was a backup or a primary replica.
4. If a primary replica failed, the `ReplicationManager` selects a backup to promote to the primary.
5. The `ReplicationManager` now proceeds to select a location where it can start a new backup to maintain a constant replica group size. This involves the following steps:
   (a) The `ReplicationManager` obtains the reference to a `FaultDetector` at the selected location from the Naming Service.
   (b) It then tries to determine if a replica at that location already exists. It does this by trying to obtain a reference to a replica at that location and polling it to determine if it is alive.
   (c) If a replica at that location does not exist, the `ReplicationManager` requests the `FaultDetector` at the selected location to start a new replica. The `ReplicationManager` thread is blocked until the `FaultDetector` has started a new replica and the replica has registered itself with the Naming Service and the `FaultDetector`.
   (d) The `ReplicationManager` then queries the Naming Service a finite number of times for the IOR of the newly created replica. In between each query, the `ReplicationManager` in the current experiment is programmed to wait for 10 seconds. Thus, if the `ReplicationManager` is unable to obtain the reference in the first attempt, it will incur a 10 second overhead. We selected 10 seconds to give enough time for the replica to register with the Naming Service. Reducing this to a very low value, however, will increase the traffic between the `ReplicationManager` and the Naming Service since the former would require several attempts to obtain the reference to the newly created replica.
   (e) Finally, the `ReplicationManager` obtains the IORs of all the replicas of the group and proceeds to create a new IOGR which is then registered with the Naming Service. This new IOGR is then sent back to the client ORB for it to use in subsequent requests.

4 Applying Patterns to Improve DOORS Fault Tolerant CORBA Performance

Implementations of FT-CORBA, such as DOORS, are representative of complex communication software. Optimizing this type of software is hard since seemingly minor "mistakes," such as poor choice of concurrency architectures and data structures, lack of caching, and the inability to configure parameters dynamically, can adversely affect performance and

#	Problem	Pattern	Pattern Category
1	Missed Polls in `FaultDetector`	Leader/Followers	Architectural
2	Excessive overhead of recovery	Active Object	Design
3	Excessive overhead of service lookup	Optimize for the common case	Optimization
		Eliminate gratuitous waste	Optimization
		Store extra information	Optimization
4	Tight coupling of data structures	Strategy	Design
		Abstract Factory	Design
5	Inability for dynamic configuration	Component Configurator	Design
6	Property lookup	Chain of Responsibility	Design
		Perfect Hash Functions	Optimization

Table 1: Patterns for Efficient Implementations of FT-CORBA

availability. Therefore, developing high-performance, predictable, reliable, and robust software requires an iterative optimization process that involves (1) performance benchmarking to identify sources of overhead and (2) applying patterns to eliminate the identified sources of overhead. The patterns described in this section are shown in Table 1.

[18, 19] describe a family of optimization principle patterns and illustrate how they have been applied in existing protocol implementations, such as `TCP/IP` and CORBA `IIOP`, to improve their performance. Likewise, our prior research on developing extensible real-time middleware [20, 11] has enabled us to document the design, architectural, and optimization principle patterns used to improve performance and predictability.

This section focuses on the various design, architectural, and optimization principle patterns we are applying to systematically improve the performance of the DOORS FT-CORBA implementation. We focus on these patterns since our benchmarking results in Section 3 revealed they were the most strategic to improve the DOORS FT-CORBA performance.

In the following discussion, we outline the forces underlying the key design challenges that arise when developing high-performance FT-CORBA middleware, such as DOORS. We also describe which patterns resolve these forces and explain how these patterns are used in DOORS. In general, the absence of these patterns leaves these forces unresolved.

4.1 Decoupling Polling and Recovery

Context: In DOORS, FT-CORBA application objects operating under the `PULL`-based fault monitoring style are polled at specific intervals of time by a separate poller thread in the `FaultDetector`. In the event of failure, this poller thread identifies an application object crash and reports the failure to the `ReplicationManager`, which then performs the recovery. Since polling and error recovery are done in the same thread of control, the `FaultDetector`'s poller thread can be blocked from polling other objects until the `ReplicationManager` has recovered from a failure.

Problem: The results of our performance benchmarks of DOORS described in Section 3 reveal that in the event of failure, the `FaultDetector`'s polling thread misses a `poll` since the thread is blocked until the `ReplicationManager` recovers from the failure. This behavior is unacceptable for systems requiring high availability. A naive solution would be to create a separate polling thread for each application object. This strategy does not scale, however, as the number of objects polled by the fault monitor increase. Thus, the force that must be resolved involves ensuring the `FaultDetector` polls all application objects at the specified intervals, even when the poller thread is blocked during failure recovery.

Solution $\rightarrow$ the Leader/Followers pattern: An effective way to avoid unnecessary blocking is to use the *Leader/Followers* pattern [14]. This pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on these event sources.

Using the Leader/Followers pattern in DOORS: Figure 5 illustrates how this pattern is implemented in DOORS's `FaultDetector`. A pool of threads is allocated *a priori* to poll a set of application objects. One thread is elected as the leader to monitor the application objects. When a failure is detected, one of the follower threads is promoted to become the new leader, which then polls the remaining application objects. In contrast, the previous leader thread informs the `ReplicationManager` of the failure and blocks until recovery completes, at which point the previous leader thread becomes a follower. This pattern resolves the force of polling all application objects, even when the poller is blocked on the recovery of a failed object.

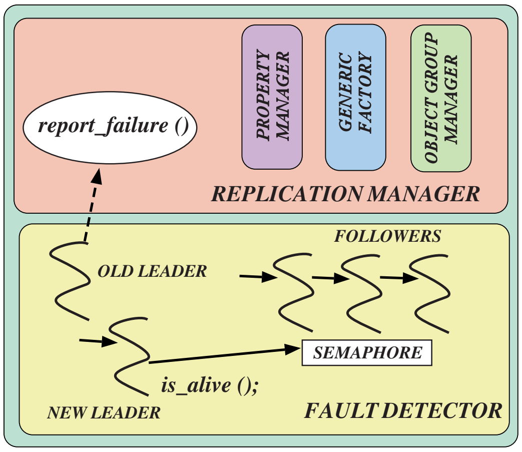

Figure 5: Applying the Leader/Followers Pattern in DOORS

4.2 Decoupling Recovery Invocation and Execution

Context: when the PULL-based monitoring style is used in the DOORS implementation, the poller thread of the FaultDetector is responsible for polling application objects at constant time intervals. Each time an application object fails to respond to the poll message, the FaultDetector must report the failure to the ReplicationManager. In contrast, the ReplicationManager can receive several such failure requests from one or more FaultDetectors. The DOORS's ReplicationManager serializes the failure report requests by handling them sequentially. For performance-sensitive applications with high availability requirements, it is imperative that the ReplicationManager be notified of failures and that recovery occur within a bounded amount of time.

Problem: Benchmarking results presented in Section 3 reveal that the system recovery and stabilization phase after a failure incurs significant overhead in DOORS. As discussed in Section 4.1, a failure notification to the ReplicationManager causes the FaultDetector's poller thread to block, which results in missed polls. In addition, this behavior precludes the propagation of failure reports from other application objects monitored by the same FaultDetector to the ReplicationManager. In DOORS, the ReplicationManager serializes all the failure reports, which degrades its responsiveness. In production systems, a ReplicationManager may receive many failure reports. Handling the failure reports sequentially incurs significant delay in the recovery process for queued requests.

A naive solution based on creating a thread per-report failure request scales poorly in a dynamic environment where failure requests may arrive in bursts. In addition, thread creation is expensive and inefficient programming may yield excessive synchronization overhead. Thus, the forces that must be resolved involve ensuring faster response to failure reports and faster recovery. Resolving these forces enables lower time to attain stability and hence higher availability.

Solution → the Active Object pattern: An efficient way to optimize system recovery and stabilization is to use the *Active Object* pattern [14]. This pattern decouples method execution from method invocation to enhance concurrency and to simplify synchronized access to an object that resides in its own thread.

Using the Active Object pattern in DOORS: In DOORS, the invocation thread of the FaultDetector calls the report_failure operation on the proxy object of the ReplicationManager which is exposed to it. Figure 6 shows how the FaultDetector can call the report_failure operation on the ReplicationManager proxy. This call is made in the FaultDetector's thread of control. The proxy then hands off the call to the scheduler of the ReplicationManager, which enqueues this call and returns control to the FaultDetector. The call is then dispatched to the ReplicationManager servant, which executes this call in the ReplicationManager's thread of control.

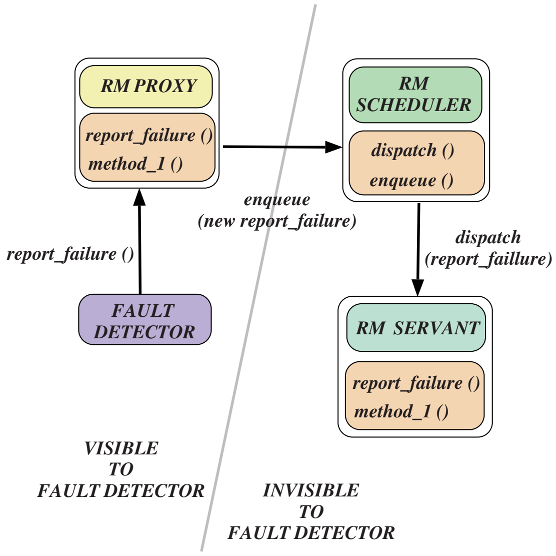

Figure 6: Applying the Active Object Pattern in DOORS

When several FaultDetectors registered with the ReplicationManager report a failure, the Active Object pattern simplifies synchronized access to the internal data structures of the DOORS's ReplicationManager. In addition, this pattern also minimizes extra locking and synchronization overhead.

4.3 Caching Object References of FaultDetector in the ReplicationManager

Context: During the object group creation and the recovery phase, the DOORS `ReplicationManager` delegates the creation of application objects to their associated factories as mandated by the FT-CORBA standard. The FT-CORBA standard does not specify how the `FaultDetectors` learn which application objects they monitor. Therefore, DOORS adopts a strategy whereby the `ReplicationManager` inform the `FaultDetectors` about the application objects they should monitor. To inform the `FaultDetectors` to start monitoring, however, the `ReplicationManager` must first obtain the `FaultDetector` object references. This process involves contacting the CORBA Naming Service or some other reference locator mechanism.

Problem: Section 3 reveals that one source of overhead in DOORS's time-to-stability is the time the `ReplicationManager` spends finding the Naming Service for the `FaultDetector`'s object reference. In the common case the `FaultDetector`'s object reference will not change, unless the `FaultDetector` itself has crashed. Thus, making a remote call to the Naming Service to obtain the `FaultDetector`'s object reference incurs unnecessary overhead and delays the object group creation and recovery process. Hence, the force to be resolved involves minimizing the time spent in obtaining the object references of the `FaultDetectors`.

Solution → Optimize for the common case by storing redundant information and eliminating gratuitous waste: Unless the `FaultDetector` has itself crashed, there is no need for the `ReplicationManager` to obtain the object reference of the `FaultDetector` each time it is needed. Instead, it can cache this information, which avoids the round-trip delay of invoking the `Naming Service` remotely.

Optimizing for the common case in DOORS: During initialization, the DOORS's `ReplicationManager` obtains the `FaultDetector`'s object reference and stores it in an internal table, as shown in Figure 7. The only time DOORS must obtain a new object reference is when the `FaultDetector` crashes, which happens infrequently in a properly configured system. This optimization can improve the time to recovery and system stabilization significantly, thereby enhancing the performance and availability of the application.

4.4 Support Interchangeable Behaviors

Context: As explained in Section 2, the FT-CORBA standard specifies several properties, such as replication styles and fault monitoring styles, and their values, which can

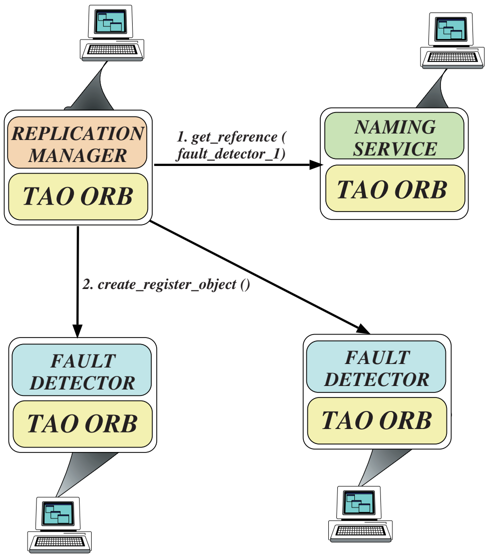

Figure 7: Optimizing for the Common Case in DOORS

be set on a per-object group, per-type, or per-domain basis. In addition, the FT-CORBA standard provides operations to override these properties or to retrieve their values. The `ReplicationManager` that inherits the `PropertyManager` interface implements these operations. Moreover, efficient implementations are possible only when efficient data structures are used to store and access these properties. The choice of data structures depends on (1) the number of properties supported by the `ReplicationManager` and (2) the maximum number of different types of object groups that are permitted.

Problem: One way to implement FT-CORBA is to provide only static, non-extensible strategies that are hard-coded into the implementation. This design is inflexible, however, since components that want to use these options must (1) know of their existence, (2) understand their range of values, and (3) provide an appropriate implementation for each value. These restrictions make it hard to develop highly extensible services that can be composed transparently from configurable strategies.

Solution → the Strategy pattern: An effective way to support multiple behaviors is to apply the *Strategy* pattern [13]. This pattern factors out similarities among algorithmic alternatives and explicitly associates the name of a strategy with its algorithm and state.

Using the Strategy Pattern in DOORS: We are enhancing different components of DOORS, such as the `ReplicationManager` and `FaultDetector`, to use the Strategy pattern. These enhancements enable developers of FT-CORBA middleware to configure these components with implementations that are customized for their requirements. Figure 8 illustrates how the Strategy pattern is applied in DOORS. As shown in this figure, different replication styles

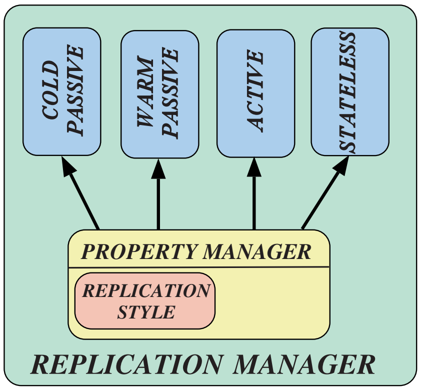

Figure 8: Applying the Strategy Pattern in DOORS

can be configured as strategies that are selectable by applications at run-time. Moreover, new strategies, such as ACTIVE_WITH_VOTING, can be added without affecting existing strategies.

4.5 Consolidating Strategies

Context: Section 4.4 describes how the Strategy pattern can be applied to configure various requirements in the FT-CORBA service. There could be multiple strategies that offer various features, such as fault monitoring style or membership style. It is important to configure only semantically compatible strategies.

Problem: An undesirable side-effect from extensive use of the Strategy pattern in complex software is the maintenance problems posed by the possible semantic incompatibilities between different strategies. For instance, the FT-CORBA service cannot be configured with active replication style and application controlled membership style. In general, the forces that must be resolved to compose all such strategies correctly involve (1) ensuring the configuration of semantically compatible strategies and (2) simplifying the management of a large number of individual strategies.

Solution → the Abstract Factory pattern: An effective way to consolidate multiple strategies into semantically compatible configurations is to apply the *Abstract Factory* [13] pattern. This pattern provides a single access point that integrates all strategies used to configure the FT-CORBA middleware, such as DOORS. Concrete subclasses then aggregate compatible application-specific or domain-specific strategies, which can be replaced *en masse* in semantically meaningful ways.

Using the Abstract Factory Pattern in DOORS: In the DOORS FT-CORBA implementation, abstract factories are used to encapsulate internal data structure-specific strategies in components such as `ReplicationManager` and `FaultDetector`. Figure 9 depicts how the property list in the DOORS `ReplicationManager` uses abstract factories. The property abstract factory encapsulates the different

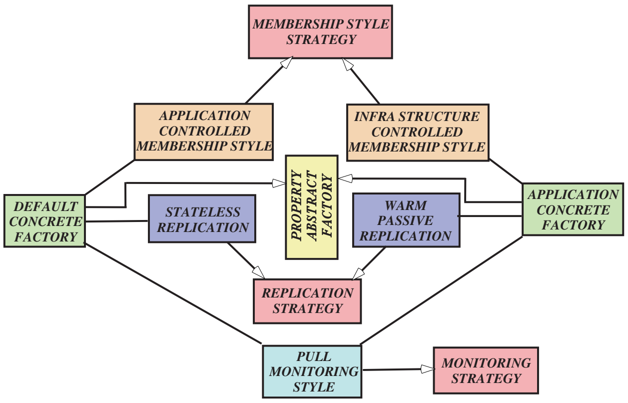

Figure 9: Applying the Abstract Factory Pattern in DOORS

property strategies, such as the membership strategy, monitoring strategy, and replication strategy. By using a property abstract factory, DOORS can be configured to have different property sets conveniently and consistently.

4.6 Dynamically Configuring DOORS

Context: FT-CORBA implementations can benefit from the ability to extend their services *dynamically*, *i.e.*, by allowing their strategies to be configured at run-time. The FT-CORBA standard allows applications to dynamically set certain fault tolerance properties of the application's replica group registered with the `ReplicationManager`. These properties include the list of factories that create each replica object of the replica group or the minimum number of replicas required to maintain the replica group size above a threshold.

Problem: Although the Strategy and Abstract Factory patterns simplify the customization for specific applications, these patterns still require modifying, recompiling, and relinking the DOORS source code to enhance or add new strategies. Thus, the key force to resolve involves decoupling the behaviors of DOORS strategies from the time when they are actually configured into DOORS.

Solution → the Component Configurator pattern: An effective way to enhance the dynamism is to apply the *Component Configurator* pattern [14]. This pattern employs explicit dynamic linking mechanisms to obtain, install, and/or remove the run-time address bindings of custom Strategy and Abstract Factory objects into the service at installation-time and/or run-time.

Using the Component Configurator pattern in DOORS: DOORS's `ReplicationManager` and `FaultDetector` use the Component Configurator pattern in conjunction with the Strategy and Abstract Factory patterns to dynamically install the strategies they require without (1) recompiling or statically relinking existing code, or (2) terminating and restarting an existing `ReplicationManager` or `FaultDetector`. Applications can use this pattern to dynamically configure the appropriate replication style, monitoring style, polling interval, and membership style into the DOORS FT-CORBA service. Figure 10 shows how these properties are dynamically linked. The use of the Component Configurator pattern

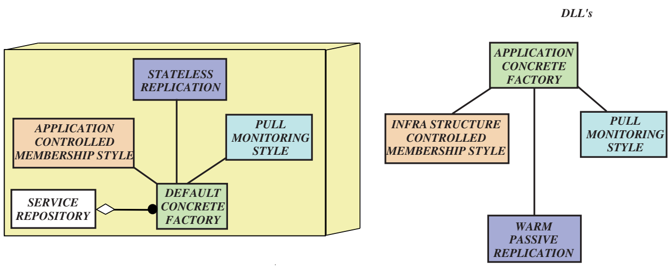

Figure 10: Applying the Component Configurator Pattern in DOORS

allows the behavior of DOORS's `ReplicationManager` and `FaultDetector` to be customized for specific application requirements without requiring access to, or modification of, the source code.

4.7 Efficient Property Name-Value Lookups

Context: The `ReplicationManager` of a FT-CORBA service is required to lookup the fault-tolerant properties of the object groups registered with it during object group creation and recovery. Properties are also located when an application retrieves them or overrides previous values. The FT-CORBA standard defines a hierarchical order in which properties must be found. First, the properties must be located for the object group that is the target of the request. If it is not found, then a lookup is made on a repository that holds properties for all object groups of the same type. If that lookup also fails, another lookup is performed on the domain-specific repository that acts as the default for all the object groups that are registered with the `ReplicationManager`.

Problem: The hierarchical lookup ordering mandated by the FT-CORBA standard underscores the need for an efficient strategy to locate fault-tolerance properties. Thus, the force that must be resolved involves efficient lookups of fault-tolerance properties guided by the order specified in the FT-CORBA standard.

Solution $\rightarrow$ the Chain of Responsibility pattern and perfect hashing: An efficient way to perform hierarchical property lookups is to use the *Chain of Responsibility* pattern [13], which decouples the sender of a request from its receiver, in conjunction with *perfect hashing* [21] to perform optimal name lookups. The Chain of Responsibility pattern links the receiving objects and passes the request along the chain until an object handles the request. Perfect hashing is applicable because the number of properties supported by a `ReplicationManager` can be configured *a priori*.

Using the Chain of Responsibility pattern and perfect hashing in DOORS: Since the fault-tolerance properties supported by a `ReplicationManager` are determined *a priori*, the DOORS service uses a perfect hash function generated by GNU gperf [21] to perform an $O(1)$ lookup on the property name. The Chain of Responsibility pattern is applied by passing the request from one hash table to the other until the property is found or the search fails, as illustrated in Figure 11.

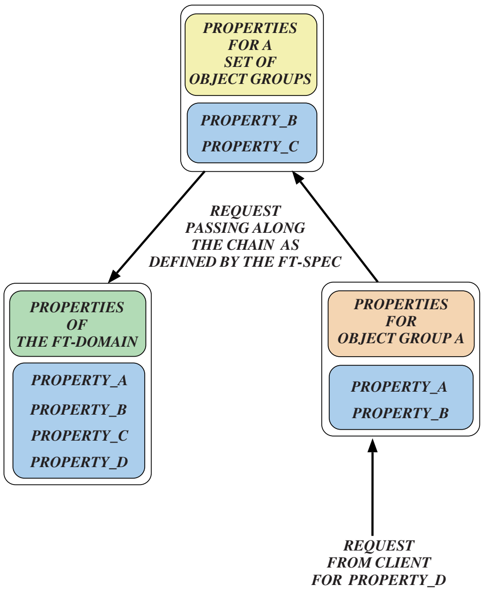

Figure 11: Applying the Chain of Responsibility Pattern in DOORS

5 Concluding Remarks

A growing number of CORBA applications with stringent performance requirements also require fault tolerance support. To address the fault tolerance requirements, the OMG recently standardized the Fault Tolerant CORBA (FT-CORBA) specification. The most flexible strategy for providing fault tolerance to CORBA applications is via higher-level CORBA services.

To make FT-CORBA usable by performance-sensitive applications it must incur negligible overhead. To address these requirements, therefore, an FT-CORBA implementation should possess the following properties:

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

To quantify the impact of different strategies for achieving the properties outlined above, we ran a series of experiments on DOORS, which is our implementation of FT-CORBA. The analysis of these results yielded a number of valuable lessons for implementing an FT-CORBA infrastructure for applications requiring stringent performance and reliability guarantees. We have identified sources of potential problems that implementors of FT-CORBA could face. Finally, we have identified and are applying key design, architectural, and optimization principle patterns to improve the performance, extensibility, scalability, and robustness of the DOORS FT-CORBA implementation.

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*, to appear 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] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, Sept. 2000, OMG.

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

[14] 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.

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, Wiley and Sons, 1996.

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

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

[18] George Varghese, "Algorithmic Techniques for Efficient Protocol Implementations ," in *SIGCOMM '96 Tutorial*, Stanford, CA, August 1996, ACM.

[19] Aniruddha Gokhale and Douglas C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, no. 9, Sept. 1999.

[20] Douglas C. Schmidt and Chris Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, no. 4, April 1999.

[21] Douglas C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, San Francisco, California, April 1990, USENIX, pp. 87–102.