Proactive Fault-Recovery in Distributed Systems

by
Soila M. Pertet

A dissertation submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

Date: May 1st, 2004

Advisor: Prof. Priya Narasimhan
Second Reader: Prof. Philip Koopman

Department of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, Pennsylvania, USA

Abstract

Supporting both real-time and fault-tolerance properties in systems is challenging because real-time systems require predictable end-to-end schedules and bounded temporal behavior in order to meet task deadlines. However, system failures, which are typically unanticipated events, can disrupt the predefined real-time schedule and result in missed task deadlines. Such disruptions to the real-time schedule are aggravated in asynchronous distributed systems by two main factors: first, delays in failure detection, and second, increased latencies due to the *reactive* fault-recovery routines that are set into motion once a failure is detected.

In this thesis, we present a general framework for *proactive* (rather than the classical reactive) fault-recovery that reduces the latencies incurred by the fault-recovery routines. Proactive fault-recovery is a technique that exploits fault prediction mechanisms in order to compensate for failures even before they occur, thereby providing bounded temporal behavior in real-time and fault-tolerant systems for certain classes of faults. In our framework, we also show how to exploit knowledge of the underlying system topology to apply the benefits of proactive fault-recovery to multi-tiered distributed systems.

We evaluate the impact of the design choices we faced when implementing a prototype of this framework in a distributed CORBA application. Our preliminary results show a promising 76% reduction in the worst-case fault-recovery latencies in our application. This demonstrates that proactive fault-recovery can indeed provide bounded temporal behavior in the presence of certain kinds of faults, thereby facilitating the development of real-time, fault-tolerant distributed systems.

Keywords: Proactive fault-recovery, Real-time, Fault-Tolerance, Distributed Systems, Middleware, CORBA

Table of Contents

Acknowledgements

I thank God for guiding me through this thesis. I would also like to thank my advisor, Professor Priya Narasimhan, for her constant guidance and support. Thank you so much, Priya, for believing in me and for the wealth of knowledge and experience that you have so freely shared with me. Thanks are also due to the members of my research group, Deepti Srivastava, Tudor Dumitras, Charlie Reverte and Joe Slember, for sharing their code with me and for the many insightful discussions that we had. Professor Philip Koopman, thank you for introducing me to the field of embedded systems and for teaching me; I enjoyed your classes immensely.

Special thanks are due to my parents, Fred and Anne Pertet, who have sacrificed a great deal to make my education possible. They have always stood by me and have been outstanding role models. I would also like to thank my brothers, Robert and Emmanuel, for all their support. Lastly, I express my gratitude to the Electronic-Integration Group at General Motors Research, particularly Pradyumna K. Mishra, Thomas Fuhrman, Alan Baum, Sanjeev Naik, Rami Debouk and Lawrence Peruski, for giving me the opportunity to work on fascinating real-time and fault-tolerant systems on their drive-by-wire test-bench during my summer internship.

I would also like to acknowledge the sponsors of this research. This work has been partially supported by the General Motors Collaborative Laboratory at CMU, the NSF CAREER Award CCR-0238381, the DARPA PCES contract F33615-03-C-4110 and the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University.

1 Introduction and Objectives

Middleware platforms, such as the Common Object Request Broker Architecture (CORBA) standard [25] and Java, are increasingly being adopted because they simplify application programming by rendering transparent the low-level details of networking, distribution, physical location, hardware, operating systems, and byte order. Since CORBA and Java have come to support many "-ilities" (*e.g.*, reliability, real-time, security), these middleware platforms have become even more attractive to applications that require a higher quality of service, such as firm real-time systems [2]. Firm real-time systems that need to exhibit both real-time and fault-tolerance properties (such as some control and multimedia applications) can tolerate occasional missed task deadlines; however, if a task is not completed by its deadline, the task is considered to be of no value and is discarded.

The CORBA standard [25] has attempted to address the needs of such applications by incorporating separate Real-Time [28] and Fault-Tolerance [26] specifications. However, due to some inherent conflicts in providing simultaneous real-time and fault-tolerance support [21], it is simply not possible for today's CORBA applications to obtain both real-time and fault-tolerance guarantees through a straightforward adoption of the CORBA standards. The difficulty in supporting both real-time and fault-tolerance properties arises because real-time systems require predictable end-to-end schedules and bounded temporal behavior in order to meet task deadlines. On the other hand, faults are unanticipated system events that can disrupt predefined real-time schedules and result in missed task deadlines.

Furthermore, these difficulties in supporting both real-time and fault-tolerance properties are compounded in highly interconnected distributed systems, where messages may pass through multiple tiers before reaching their final destination. For instance, in an Unmanned Air Vehicle (UAV) data distribution system [13], several UAVs can stream video feeds to a stationary distributor, which, in turn, forwards the video to user hosts for display or to an automated target recognition system. Faults that occur in any one of these tiers sometimes propagate to the successive tiers involved in the computation, resulting in increased fault-detection and fault-recovery times across the distributed system.

The MEAD (Middleware for Embedded Adaptive Dependability) system [20] that we are developing at Carnegie Mellon University attempts to reconcile the conflicts between real-time and fault-tolerance properties in a resource-aware manner. One novel aspect of MEAD is its use of a *proactive dependability framework that lowers the impact of faults on a distributed application's real-time schedule*. The aim here is to design and implement mechanisms that can predict, with some confidence, when a failure might occur, and that can compensate for the failure even before it occurs.

Proactive fault-recovery is a technique that exploits fault-prediction mechanisms in order to compensate for failures even before they occur, thereby providing bounded temporal behavior in real-time and fault-tolerant systems, for certain classes of faults. Some faults, such as resource-exhaustion faults [3, 9, 11] and some intermittent hardware faults like disk crashes [16], exhibit a pattern of abnormal behavior that favors fault-prediction. MEAD's proactive dependability framework uses these predictions to initiate fault-recovery that would incur a lower penalty than a reactive fault-recovery strategy in which we first waited for the process to crash before taking any action. For instance, if we knew that a node had an 80% chance of failing within the next five minutes, we could gracefully migrate all of its hosted processes to another working node in the system, with the aim of meeting the application's real-time deadlines. *Because it is not always possible to predict failures for every kind of fault, proactive dependability complements, but does not replace, the classical reactive fault-tolerance schemes.*

1.1 Contributions of this Thesis

The research presented in this thesis addresses the problem of supporting both real-time and fault-tolerance properties in asynchronous distributed systems. To achieve this goal, this thesis introduces a novel proactive fault-recovery framework that provides bounded fault-recovery times for distributed real-time applications, in the face of a class of predictable faults. We focus on the mechanisms needed to implement proactive fault-recovery in a distributed systems, and do not attempt to develop a new failure-prediction technique. Instead, we exploit relatively simple failure-prediction mechanisms within the MEAD system, and demonstrate how to use the resulting predictions to initiate fault-recovery that minimizes both the jitter and the "latency" spikes experienced by distributed applications in the presence of faults. This thesis makes the following concrete contributions:

- A general framework for proactive fault-recovery in distributed systems;
- Exploiting knowledge of the system topology when applying proactive fault-recovery to distributed multi-tiered applications;
- Implementing this framework transparently for distributed CORBA applications;
- Empirical evaluations and measurements that quantify the overhead and performance of our new approach, as compared with the classical reactive fault-tolerance strategy.

While we employ CORBA as the vehicle for our investigations in this thesis, our techniques are, for the most part, independent of CORBA, and can be readily extended to non-CORBA-based distributed applications.

1.2 Structure of this Thesis

The remainder of this thesis is organized as follows: Chapter 2 provides background information on CORBA, and discusses related work. Chapter 3 introduces our novel proactive fault-recovery framework and discusses the factors that determine the effectiveness of our approach. This chapter also explains how to exploit knowledge of the system topology when extending our approach to multi-tiered distributed applications. Chapter 4 presents an in-depth description of our system architecture, while Chapter 5 highlights the experimental results that we obtained from applying our prototype to a distributed CORBA application. We then conclude by summarizing our main contributions.

2 Background

CORBA [25] is a middleware specification that simplifies application programming in distributed systems by rendering transparent the low-level details of networking, distribution, physical location, hardware, operating systems, and byte order. A key component of CORBA is the Object Request Broker (ORB) that enables clients to request object implementations from servers seamlessly over the network. The ORB locates object implementations using an Interoperable Object Reference (IOR) that contains the address of the server hosting the object, as well as an object key that uniquely identifies the object. Object keys can be either transient or persistent. Transient object keys contain host-specific information such as timestamps, and are typically only valid during the lifetime of a specific server instance. On the other hand, permanent object keys (used in our approach) transcend the lifetime of a specific server instance, and assign identical keys to objects independent of the hosting server.

CORBA specifies a generic communication protocol known as General-Inter-ORB Protocol (GIOP). GIOP defines a set of eight message formats and common data representations for communications between ORBs. The GIOP messages that are of most relevance to our work are *Request* messages sent from the client to the server, and the *Reply* messages sent by the server in response to a client's invocation. Each *Request* message encapsulates a *request_id* field that allows the client ORB to match incoming responses from a server to the outgoing invocations from the client. The *Reply* message from the server contains a *reply_status* that notifies the client of the completion status of its invocation, *e.g.*, whether or not an exception occurred while the server processed the invocation. Our architecture exploits two values of GIOP *reply_status* field:

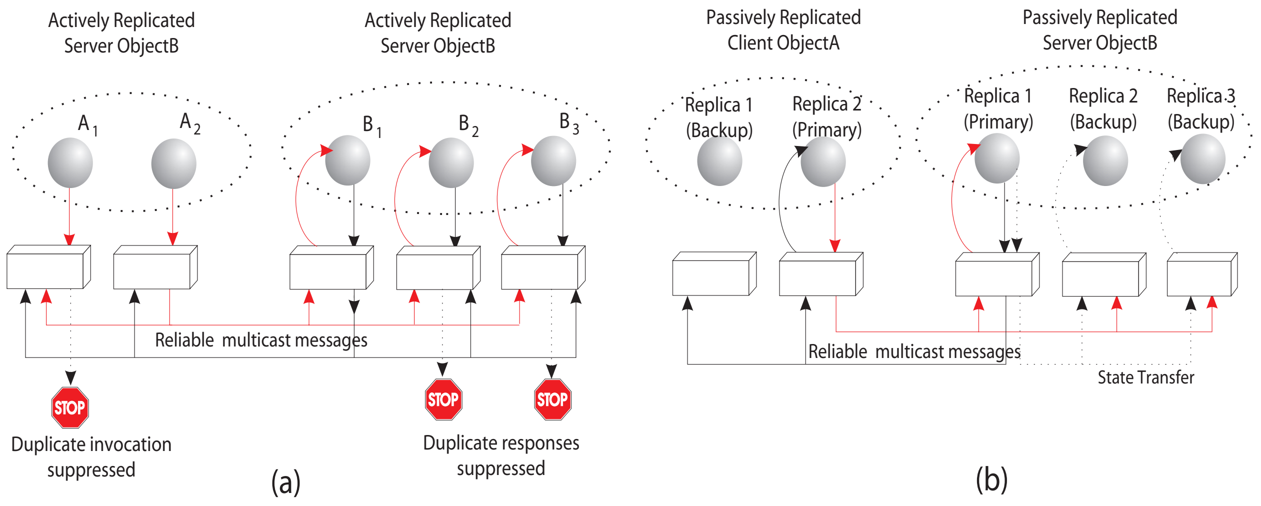
- *LOCATION_FORWARD,* which indicates to the client that it should reissue its request to a server at some other location, and
- *NEEDS_ADDRESSING_MODE*, which directs the client to supply more addressing information and usually prompts the client ORB to re-send the request (transparently to the client).

2.1 Fault-Tolerance for CORBA

Until the recent adoption of the Fault-Tolerant CORBA standard [26], CORBA had no intrinsic support for fault-tolerance. Early fault-tolerant CORBA systems (which preceded the FT-CORBA standard) adopted diverse approaches: the integration approach, where the support for replication is integrated into the ORB (*e.g.*, Electra [17]), the interception approach, where the support for replication is provided transparently underneath the ORB (*e.g.*, Eternal [22]), and the service approach, where support for replication is provided primarily through a collection of CORBA objects above the ORB (*e.g.*, OGS [8]).

The recent Fault-Tolerant CORBA standard (adopted in March 2000) describes the minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for supporting more advanced fault-tolerance. FT-CORBA implementors are free to use proprietary mechanisms (such as reliable multicast protocols) for their actual implementations, as long as the resulting system complies with the specified interfaces and the behavior expected of those interfaces. Despite their differences, the various approaches to fault-tolerant CORBA (including the new FT-CORBA standard) are alike in their use of replication to protect the application against faults. CORBA applications can be made fault-tolerant by replicating their constituent objects, and distributing these replicas across different processors in the network. The idea behind object replication is that the failure of a replica (or of a processor hosting a replica) of a CORBA object can be masked from a client because the other replicas can continue to provide the services that the client requires.

Figure 2-1. Active and Passive Replication.


Actively Replicated Server ObjectB
Actively Replicated Server ObjectB
A_1
A_2
B_1
B_2
B_3
Reliable multicast messages
STOP
STOP
STOP
Duplicate invocation suppressed
Duplicate responses suppressed
(a)
Passively Replicated Client ObjectA
Passively Replicated Server ObjectB
Replica 1 (Backup)
Replica 2 (Primary)
Replica 1 (Primary)
Replica 2 (Backup)
Replica 3 (Backup)
Reliable multicast messages
State Transfer
(b)

There are essentially two kinds of replication styles -- active replication [33] and passive replication [4], as seen in Figure 2-1 [22]. With **active** (also known as state-machine) replication, each server replica processes every client invocation, and returns the response to the client (of course, care must be taken to ensure that only one of these duplicate responses is actually delivered to the client). The failure of a single active replica is masked by the presence of the other active replicas that also perform the operation and generate the desired result. With **passive** replication, only one of the server replicas, designated the **primary**, processes the client's invocations, and returns responses to the client. With **warm passive** replication, the remaining passive replicas, known as **backups**, are preloaded into memory, and are synchronized periodically with the primary replica, so that one of them can take over should the primary replica fail. With **cold passive** replication, however, the backup replicas are "cold", *i.e.*, not even running, as long as the primary replica is operational. To allow for recovery, the state of the primary replica is periodically checkpointed and stored in a log. The primary also maintains a log of activities between checkpoints. If the existing primary replica fails, a backup replica is launched, with its state initialized from the latest checkpoint and activity log, to take over as the new primary. Typically, clients connected to the old primary will **fail-over** to the new primary replica.

Both passive and active replication styles require mechanisms to support state transfer. For passive replication, the transfer of state occurs periodically from the primary to the backups (warm passive), from the existing primary to a log (cold passive), or from the log to a new primary (cold passive). For active replication, the transfer of state occurs when a new active replica is launched and needs its state synchronized with the operational active replicas. Fault-tolerant CORBA systems require the application to be deterministic, *i.e.*, any two replicas of an object, when starting from the same initial state, and after processing the same set of messages in the same order, should reach the same final state. Mechanisms for strong replica consistency (ordered message delivery, duplicate suppression, *etc.*), along with the deterministic behavior of applications, enable effective fault-tolerance so that a failed replica can be readily replaced by an operational one, without loss of data, messages or consistency. Note that the classical reactive fault-tolerance approach would first wait to detect the failure of an active or a passive replica, and then initiate recovery such as the launch of a replica.

The underlying system model for fault-tolerant CORBA systems is an asynchronous distributed system, where processors communicate via messages over a local-area network that is completely connected, (*i.e.*, network partitioning does not occur). Communication channels are not assumed to be FIFO or authenticated, and the system is asynchronous in that no bounds can be placed on computation times or message-transmission latencies. Processors receive their own messages, and have access to local clocks that are not necessarily synchronized across the system. The fault model includes communication, processor, and object faults. Communication between processors is unreliable and, thus, messages may need to be retransmitted. Processors, processes and objects are subject to crash faults, and thus, might require recovery and re-instatement to correct operation.

2.2 FT-CORBA and RT-CORBA

In addition to the FT-CORBA standard, CORBA also supports a Real-Time CORBA (RT-CORBA) standard that [28] provides standard interfaces to meet an application's real-time requirements by facilitating predictable end-to-end scheduling of activities, and by supporting the management of resources in the system. The current FT-CORBA and RT-CORBA standards were developed independently, and make several conflicting assumptions about the behavior of the system. These conflicts are summarized in Table 2-1, and are inherent to real-time and fault-tolerant systems [21]. The proactive fault-recovery framework that we present relaxes the "no advance knowledge of faults" assumption by introducing some notion of "predictability" to system failures. We recognize that some system failures might occur so abruptly that we cannot possibly hope to predict them; we also acknowledge that fault prediction provides only statistical guarantees. For these reasons, proactive fault-recovery complements, but does not entirely replace, the current reactive fault-recovery strategy adopted by FT-CORBA.

Table 2-1. Conflicts between real-time and fault-tolerance.

Real-Time Systems	**Fault-Tolerant Systems**
Typically requires a priori knowledge of events	No advance knowledge of when faults might occur, and of how long fault-recovery might take
Determinism means predictability of the temporal behavior of a single (likely unreplicated) object	Determinism means predictability/coherence of state and responses across the replicas of an object (replica determinism)
Operations ordered to meet task deadlines	Operations ordered to preserve replica consistency
Traditionally synchronous	Traditionally asynchronous
Multi-threading for concurrency and task scheduling	Replica-determinism prohibits use of multi-threading
Use of timeouts and timer-based mechanisms	Replica-determinism prohibits use of local processor time

2.3 Related Work

Huang et al. [11] proposed a proactive approach, called software rejuvenation, for handling transient software failures such as memory leaks. Software rejuvenation involves gracefully halting an application once errors accumulate beyond a specified threshold, and then restarting the application with a clean internal state. Subsequent work in software rejuvenation has focused on constructing rejuvenation policies that increase system availability and reduce the cost of rejuvenation [3, 9, 11]. Our initial results [29] show that simply restarting a faulty server that has ongoing client transactions can lead to unacceptable jitter and missed real-time deadlines at the client. This thesis presents mechanisms needed to gracefully handoff existing clients on faulty servers at the onset of the rejuvenation threshold, allowing the server to reach a quiescent state before it is restarted.

Castro and Liskov [5] developed another proactive fault-recovery system. Their system implemented a proactive recovery scheme for Byzantine fault-tolerant systems that had untrusted clients. In their scheme, faulty servers periodically restarted themselves, and clients detected these failures by timing out and retransmitting their requests to all the replicas in the group. Again, this fail-over process may result in increased jitter at the client. Our system uses proactive notifications to lower fail-over times in systems with trusted clients.

Ruggaber and Seitz [32] considered the hand-off (similar to fail-over) problem in wireless CORBA systems. They developed a proxy platform that uses a modified ORB to transparently handoff mobile CORBA clients to wired CORBA servers. Instead, our approach implements transparent fail-over in wired environments using interception mechanisms [15] that do not require us to modify the ORB.

A variety of statistical fault-prediction techniques have also been proposed in the literature. Lin and Siewiorek [16] developed a failure prediction heuristic that achieved a 93.7% success rate in predicting faults in the campus-wide Andrew File System at Carnegie Mellon University. Vilalta and Sheng [37] used temporal data mining to predict faults in a telecommunication network. Our research does not focus on fault-prediction techniques, but rather on how to exploit fault-prediction in systems that have real-time deadlines. This thesis establishes that proactive fault-recovery is effective and can provide bounded temporal behavior, in the presence of certain kinds of faults, thereby allowing us to support both fault-tolerance and real-time properties in distributed applications.

3 . Proactive Fault Recovery

Proactive fault-recovery is an approach that exploits fault-prediction mechanisms in order to compensate for failures even before they occur, thereby providing bounded temporal behavior in real-time and fault-tolerant systems, for certain classes of faults. Even in the absence of fault-prediction, the techniques we describe may be helpful in solving other research problems in dependable systems such as live software upgrades [36]. This is because our fault-recovery techniques provide a means of achieving a quiescent[1] state at a server process, which is one of the challenges faced during a live software upgrade.

Equation 3-1. Scenario in which proactive fault-recovery is highly beneficial.

$$\boxed{\textit{Reactive Fault-recovery Latency} >> \textit{Proactive Fault-recovery Latency}}$$

The bounded temporal behavior provided by proactive fault recovery is very evident in systems where the latencies incurred by the reactive fault-recovery routines are significant (Equation 3-1). In proactive fault-recovery, we execute the reactive fault-recovery routines *before* the actual system failure, thereby reducing the amount of work the system needs to perform to recover from the fault.

For instance, when a server crashes in a warm-passively replicated system [4], some ongoing client requests may be lost and will need to be retransmitted to the new primary. However, if the server "knew" that it was about to crash, it could gracefully migrate its current clients to a new primary and avoid the performance penalty incurred by request retransmissions. In systems where the reactive fault-recovery latencies are relatively low, such as actively replicated [29] systems, the performance of both the proactive and the reactive schemes should be about the same. In this case, proactive fault-recovery may be used ensure that a minimum level of resilience (*e.g.*, through a minimum number of replicas) is maintained by launching new replica processes whenever the system detects that a given replica is about to fail.

1. Quiescence is often required in dependable systems in order to ensure that one can "safely" extract a process' state without fear of the state undergoing modification at that point in time. Checkpointing and live software upgrades require applications to be quiescent before the necessary mechanisms can be used.

3.1 Challenges Faced in Proactive Fault-Recovery

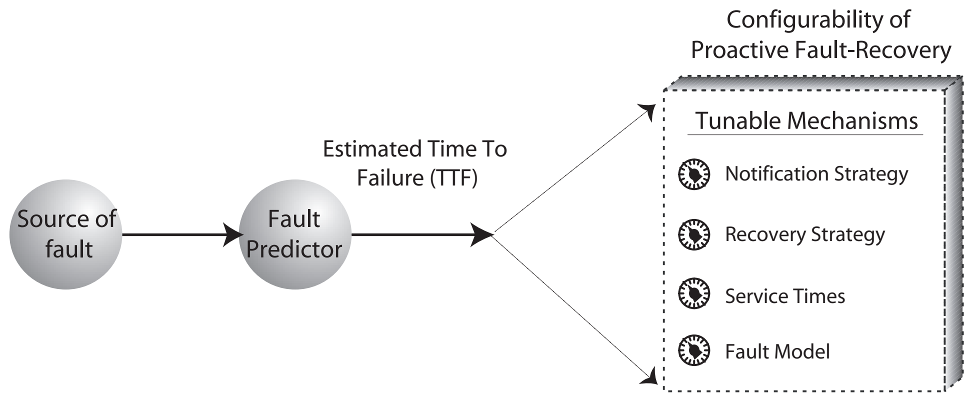
There are two main challenges faced when designing any proactive fault-recovery recovery strategy: the first is determining “when” to initiate proactive fault-recovery, and the second is determining “how” to propagate the proactive fault-recovery notifications through the system.

When to initiate proactive fault-recovery? If we trigger fault-recovery too often, the additional overhead of unnecessarily failing over clients to non-faulty server replicas quickly negates the benefits of using a proactive strategy. On the other hand, if we wait too long to initiate recovery, the very purpose of a proactive strategy is lost because the client does not have enough time to fail-over. In this case, the resulting fault-recovery ends up resembling a reactive strategy. The ideal scenario is to delay proactive fault-recovery so that the system has just enough time to initiate and complete the fault-recovery routines before the server fails.

How to propagate proactive fault-recovery notifications? This choice is greatly influenced by both the reactive fault-tolerance strategy adopted by the system and the current system topology. For example, in multi-tiered distributed systems, we need to determine whether to restrict the proactive fault notifications to neighboring tiers or whether to propagate the notifications to every tier in a given nested invocation.

Figure 3-1 further illustrates the rationale for determining “when” and “how” to initiate proactive fault-recovery. For a system component, MEAD’s fault-predictor provides an estimated Time-to-Failure (TTF), *i.e.*, when the component is likely to fail based on current pre-fault indications. Based on this estimation, the proactive-fault recovery mechanisms opt to either initiate or delay fault-recovery actions. For instance, the fault predictor may inform MEAD that a server replica has an 80% chance of failing within the next ten minutes. At this point, MEAD’s proactive recovery mechanisms need to decide whether to fail-over immediately or to continue processing client requests for, say, another seven minutes, before handing over control over to a new server replica.

Figure 3-1. Tuning proactive fault-recovery.



A number of tunable factors (See Table 3-1) influence the proactive fault-recovery decision:

- *Application-specific characteristics:* These are inherent to the application, for instance, the average time that a server takes to service a client request, as well as the client's invocation rates.
- *Fault-tolerance properties:* These are influenced by the reactive fault-tolerance strategy adopted in the system, for example, the replication style and the fault-model.
- *System topology factors:* These depend on how functionality is distributed across the application, for instance, whether the application is structured as a simple two-tier hierarchy or consists of three or more application tiers.

.

Table 3-1. Factors that influence proactive fault-recovery.

Decision to be made	**Tunable Factor**	**Mechanism/Parameters**
When to initiate recovery	Application-specific	Service times Client invocation rates
	Fault-tolerance strategy	Accuracy of fault-predictor Fault-recovery latencies
	Topology	Network propagation delays
How to perform recovery	Application-specific	Stateless or stateful application
	Fault-tolerance strategy	Replication style (Active or passive) Fault model (Byzantine or fail-stop) Fail-over technique (Client-side vs Server-side)
	Topology	Number of application tiers

3.2 When to Initiate Proactive Fault-Recovery

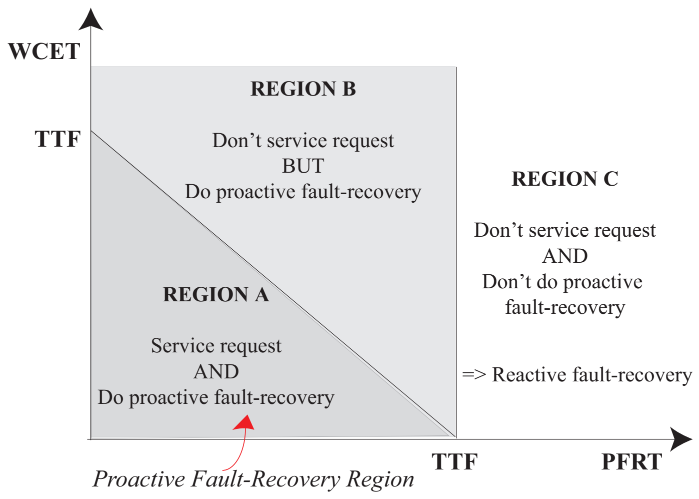
There are two aspects to the question of "when" to initiate proactive fault-recovery: (i) the actual instant in time that we should initiate recovery; which affects how much benefit we obtain from proactive fault-recovery, and (ii) "how often" we should initiate recovery, which affects the amount of overhead introduced by the scheme. For example, we may opt to initiate proactive fault-recovery only after every third predicted fault in order to reduce the overhead in the system. In general, we minimize the overhead of proactive fault-recovery in a system by initiating recovery actions at the point when the system has just enough time to execute and complete all the fault-recovery routines before the faulty server crashes. This implies that, whenever a client request arrives at a faulty server, the proactive fault-recovery mechanisms may need to make one of the following decisions (See Figure 3-2); based on the values of the Time-to-Failure (TTF) when the client's request arrives, the worst-case execution time (WCET) of the request at the server, and the proactive fault-recovery time (PFRT).

1. *Allow the server to execute the incoming request:* This decision is made when based on the TTF, there is sufficient time to execute both the client's request and the proactive fault-recovery routines if required. For example, if a request with WCET=2ms arrives at the server when the TTF=10ms and the PFRT=4ms, then the server can execute the request within 2ms, still leaving enough time (8ms) to execute proactive fault-recovery routines.

2. *Initiate proactive fault-recovery:* This decision is made when the faulty-server does not have enough time to execute *both* the current request and proactive fault-recovery routines, but has sufficient time to execute the proactive fault-recovery routines, at least. For example, if a request with WCET=2ms arrives at the server when the TTF=5ms and the PFRT=4ms, clearly we are better off if we discarded the request and initiated proactive fault-recovery.

3. *Do nothing:* This decision is made, when based on the TTF, there is neither enough time to service the client request or execute the proactive fault-recovery routines. In this case, the system reverts to its reactive fault-recovery strategy. An example would be if TTF=5ms, WCET=6ms and PFRT=10ms.

Figure 3-2. Timing decision for proactive fault-recovery.



```
If ((WCET + PFRT) > TTF) Then
   // Region A
   Service Request
Else If (PFRT > TTF) Then
   // Region B
   Initiate Proactive Fault-Recovery
Else
   // Region C
   Do Nothing. Revert to Reactive Recovery
End If
```

A number of factors influence the estimates of the WCETs and the proactive fault-recovery times. These factors are discussed in detail below.

3.2.1 Application-Specific Factors

WCETs or Service times: The WCETs for the client requests may be computed through the static program analysis of application code [31]. The results of this analysis can be stored in a dispatch table, within the proactive fault-recovery mechanisms, that can be accessed at run-time. This approach assumes that the performance of the hardware is known beforehand, and that no interrupts occur during the execution of the client. However, because the execution time for a task can change at run-time (due to variable system load, environment, *etc.*), run-time monitoring similar to that proposed in the Time-Aware Fault-Tolerant (TAFT) scheduling system [24] might be more appropriate.

Client invocation rates: The influence of client invocation rates on proactive fault-recovery times is subtle and is better explained with an example. The basic idea is that is if the next client invocation is scheduled to occur at a time after the estimated Time-To-Failure, the client should be informed of the impending server failure during the current invocation. For example, if a server receives requests from clients once every 200ms and at the time the server receives a client request, the TTF=150ms, the WCET=20ms, and the PFRT=50ms. Clearly, even though we have enough time to execute the both the client request and the proactive-fault recovery routines, if we delay proactive fault-recovery until the next client invocation, it will be too late and the system will have to run the expensive reactive fault-recovery routines.

3.2.2 Fault-Tolerance Factors

Accuracy of the fault-predictor: We need a fault-predictor with a reasonable accuracy. However, a fault-predictor might malfunction in two ways. The first way is by computing a TTF that is too small. In this case, proactive fault-recovery will be initiated too late to be of any use, and the system will revert to its reactive fault-recovery routines. A second way in which the fault-predictor might malfunction is by computing a TTF that is too large. Although in this scenario the system may still meet its real-time deadlines because the time needed to execute the proactive recovery routines is relatively low, the efficiency of the system is compromised due to the extra overhead of initiating fail-overs unnecessarily.

Fault-recovery latencies: The fault-recovery latencies include the time to find an alternative working server replica, to restore its state consistently, and then to fail-over clients to use this working replica. These fault-recovery latencies depend on the reactive fault-tolerance strategies that are in place within the system. For instance, warm passively replicated applications typically experience higher recovery latencies than actively replicated applications.

3.2.3 Topology Factors

Network propagation delays: The proactive fault-recovery mechanisms need to take into account the latencies for message transmission from the server to the client. Ideally, the proactive fault-recovery mechanisms should not allow a server to service a client request if the WCET of the request and the network propagation delay exceed the MTTF.

3.2.4 Critique

The factors listed above are not exhaustive, but they give a good representation of the timing information that the proactive fault-recovery mechanisms would ideally need to collect. Care must be taken to ensure that the overhead of monitoring and extracting this timing information does not “clog” the system. The proactive fault-recovery mechanisms need to determine the minimum amount of information that they need in order to make a reasonably accurate decision about when to initiate proactive fault-recovery. For instance, if the network propagation delays in the system are minimal, they can be ignored when computing the fault-recovery times, resulting in lower overheads. Poor timing estimates will either cause us to initiate proactive recovery too late or too often. Initiating proactive fault-recovery too late will cause us to revert to a reactive fault-recovery strategy, whereas initiating recovery too often will lead to increased system overheads.

3.3 How to Implement Proactive Fault-Recovery

The proactive fault-recovery notifications can be propagated through the system in several ways. The mechanisms that we outline in this section will mainly focus on passively replicated systems because implementing proactive fault-recovery in actively replicated systems is the more trivial case because these systems typically have low fault-recovery latencies. The factors that affect how proactive fault-recovery is implemented in a system are outlined below.

3.3.1 Application-Specific Factors

Stateless vs. Stateful applications: For stateless applications, the proactive fault-recovery problem is very similar to the problem of load-balancing or high availability. This is especially true if the application assumes a fail-stop fault-model. Such stateless applications, the failing replica can continue to service client requests for a certain "grace period" after the proactive fault recovery decision has been made. This "grace period" gives the newly-elected primary replica some time to recover any system state so that the effect of the fail-over is masked from the client. However, in stateful applications, the failing primary replica must yield complete control to the newly elected primary because we need to maintain consistent state across all of the replicas in the system. The process by which control is handed over to the new primary is listed below:

1. *Take a proactive state-checkpoint:* A proactive state-checkpoint involves recording the internal state information on the failing replica, and broadcasting this information to the other replicas in the group so that they reflect the most recent server state. Proactive checkpoints lower the fault-recovery times by reducing the amount of work that needs to be done during recovery and fail-over, in the event of a system failure.

2. *Elect the new primary:* The replicas then agree amongst themselves who the next primary will be. They exclude the failing replica from consideration in the re-election process.

3. *Newly-elected primary starts servicing client requests:* The recovery latencies for stateful applications will be higher because the servicing of client requests will be put on hold until the newly-elected primary attains a consistent state.

3.3.2 Fault-Tolerance Factors

The reactive fault-tolerance strategy adopted by the system greatly influences how proactive fault-recovery is implemented in the system. These fault-tolerance factors are as follows:

Replication style: Systems usually implement either active [33] or passive replication [4], or a variation of these two styles, *e.g.*, semi-active replication [30]. Passively replicated applications typically use fewer system resources but have higher fault-recovery latencies when compared to actively replicated systems. Proactive fault-recovery can help lower the fault-recovery latencies associated with passively replicated systems, thereby coupling the benefits of low resource usage in passive replication with the faster fault-recovery times associated with active replication. In actively replicated systems, proactive fault-recovery can be used to ensure that a certain minimum resilience (through the number of replicas) by launching new replicas whenever it detects that a given replica is about to fail.

Fault model: The types of faults that a system can handle can range from fail-silent to Byzantine faults [14]. Proactive fault-recovery systems designed under the fail-silence assumption are simpler because the fault-recovery decision can be localized to proactive fault-recovery mechanisms collocated at a given node. However, if systems need to handle Byzantine failures, the proactive fault-recovery mechanisms on each replica need to share information with each other, and agree on the current view of the system; any proactive fail-over decision must be based on the output of at least $3k+1$ nodes where k is the number of Byzantine failures tolerated. This scheme leads to increased system complexity.

Fail-over technique: Fail-over to a new server replica may occur either at the client-side or the server-side. In client-side fail-over, the client is aware of the address of the new server replica and establishes an explicit connection to it during fail-over. With server-side fail-over, the client typically addresses the server replicas as a group, and is not explicitly made aware of the addresses of the individual server replicas. All the fail-over mechanisms occur at the server-side (at each individual server replica), and changes in server group membership are completely masked from the client.

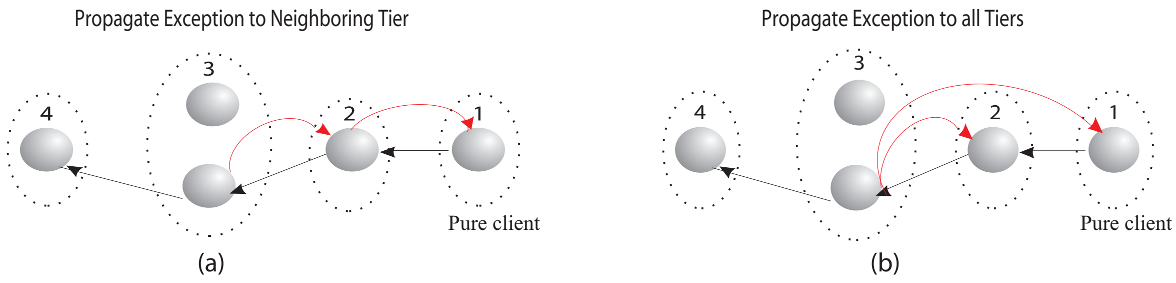
3.3.3 Topology Factors

A distributed system based on the client/server model may be implemented as a simple two-tier application or it may support nested invocations that span more than two tiers. Most distributed applications nowadays consist of multiple tiers, and we, therefore need to extend our proactive approach to such applications. In multi-tiered systems, we can either limit the proactive fault-notifications to just the neighboring tiers, or take globally proactive actions across all the tiers involved in the nested invocation. The simplest case is to restrict the proactive fault-recovery notifications to the neighboring tiers, thereby representing the nested invocation problem as a series of "independent" client/server pairs as shown in Figure 3-3.

If, on the other hand, we decide to invalidate the entire end-to-end operation, we need to propagate an exception simultaneously to all of the tiers involved (as opposed to letting the exception propagate sequentially from tier-to-tier at the application level). For example, when a fault occurs at tier 3, we would broadcast the exception to tiers 1 and 2. We would not propagate an explicit exception to tier 4 because it violates the client/server model (an exception is a type of a *Reply* message, and clients do not send replies to servers). Instead, we would propagate the exception implicitly to the server in tier 4 by resetting the connection established between tier 3 and tier 4. To implement this scheme, we need some knowledge of the system topology. We also need to determine if this invalidation will be transparent to the application, as discussed below:

- *Determining system topology:* For instance, the proactive fault-recovery mechanisms at tier 3 need to know that the invocation that they are dealing with was initiated by tier 1 and passed through tier 2 before reaching tier 3. We could piggyback information about the topology information on the messages that are passed from tier to tier. The disadvantage of this approach is that messages can grow arbitrarily long. The other option is to map the logical dependencies in an application to some type of run-time representation that can be queried by the proactive fault-recovery mechanisms. For instance, Ensel and Keller [7] have proposed a system that tracks service dependencies using XML (eXtensible Markup Language). We could also envision doing this through static program analysis.
- *Transparency at the application layer:* We believe that global exception-handling [19] should be visible to the programmer. This is because the programmer needs to specify how to invalidate any partial state (at the application layer) that has been generated during the transaction.

Figure 3-3. Proactive fault-recovery in multi-tiered distributed systems.


Propagate Exception to Neighboring Tier
4
3
2
1
Pure client
(a)
Propagate Exception to all Tiers
4
3
2
1
Pure client
(b)

4 System Architecture

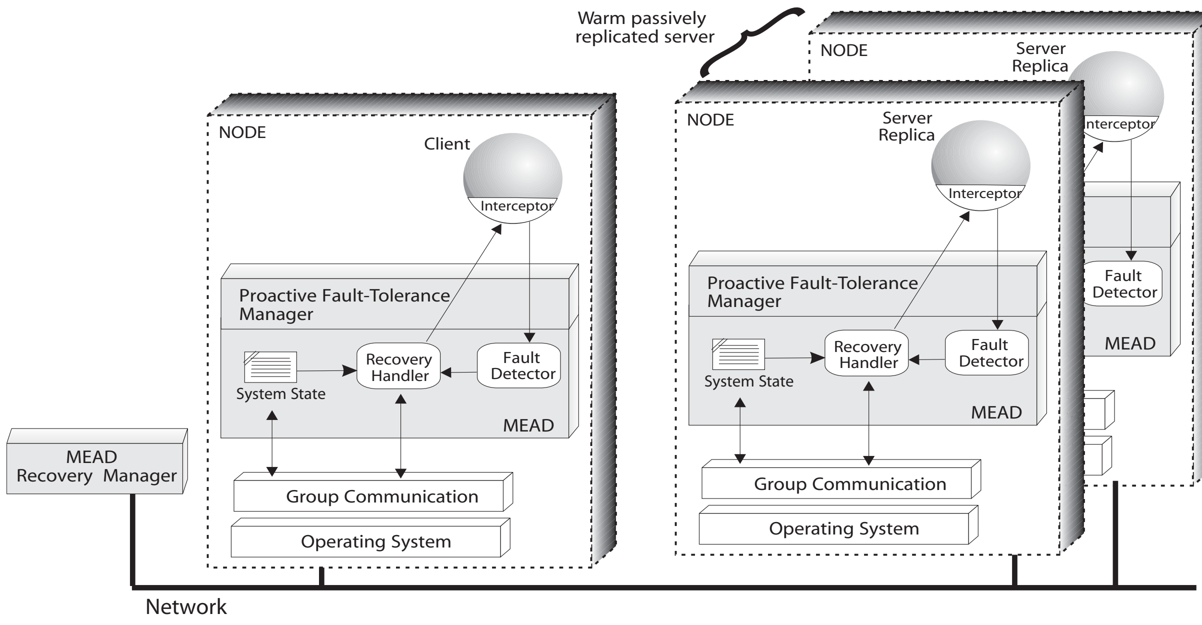
The proactive fault-recovery framework, which we are developing within the MEAD project, helps to lower the impact of faults on real-time schedules. Our main objective when designing this framework was to build a transparent fault-tolerant infrastructure that masked failures at the application, lowered the average fail-over time, and incurred a reasonable overhead. Our framework uses replication and proactive fault-notifications to protect applications against faults. Although we use CORBA to develop our system, the concept of proactive notifications can be extended to other types of middleware. Our system builds upon the transparency provided by CORBA, and makes use of some of the notification messages already supported by the CORBA standard.

4.1 Design Assumptions

In our development of MEAD's proactive fault-recovery framework, we make the following assumptions:

- Operation in an asynchronous distributed system;
- Independent failures across the server replicas and the nodes;
- A fault model that covers process crash-faults, node crash-faults, message-loss faults and resource-exhaustion faults;
- Deterministic, reproducible behavior of the application and the ORB. This also implies that there is no multi-threading in either the server or client processes (similar to the assumptions made in the FT-CORBA standard). This is because multi-threading may lead to different orders of execution across the replicas in the system, and result in replica inconsistency.

Figure 4-1. MEAD's proactive fault-tolerance architecture.



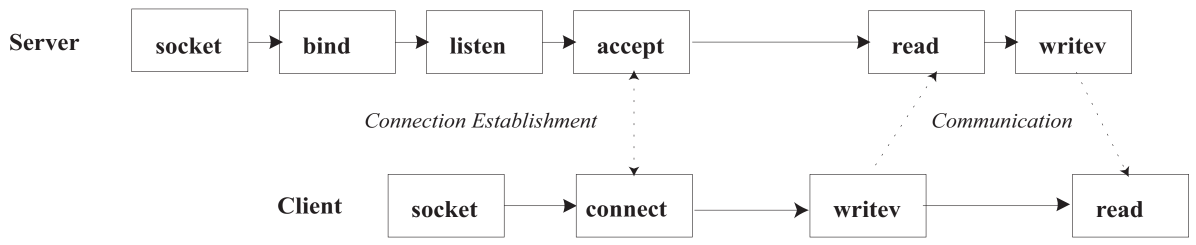
4.2 MEAD Proactive Fault-Recovery Architecture

The MEAD proactive fault-recovery architecture has three main components: the Interceptor, the Proactive Fault-Tolerance Manager, and the Recovery Manager (see Figure 4-1). The MEAD Interceptor transparently modifies the behavior of the application, and serves as the insertion-point for our fault-recovery hooks. The MEAD Fault-Tolerance Manager monitors resource usage and triggers fault-recovery actions when it senses that a server replica is about to fail. Finally, the MEAD Recovery Manager launches new replicas that restore the application's resilience after a server replica crashes. MEAD exploits an underlying totally-ordered reliable group communication system, specifically, the Spread system [1], to obtain the reliable delivery and message ordering guarantees required for consistent node-level and process-level membership.

4.2.1 The MEAD Interceptor

Interceptors are software components that can transparently modify the behavior of the function calls invoked by an application. CORBA provides standardized support for interception through its Portable Interceptors [27], which requires modification of the application code to insert hooks for intercepting the application's request and reply messages. Due to the lack of transparency in Portable Interceptors, and due to some limitations [18] in their usage, such as the inability to block server replies without raising an exception at the application, we favor a transparent library interpositioning approach [15] instead. Library interpositioning provides us with greater flexibility, and allows us to implement proactive recovery transparently for an unmodified CORBA application running over an unmodified Linux kernel.

Figure 4-2. Sequence of steps for connection establishment and communication in CORBA.



MEAD's proactive recovery framework tracks GIOP messages communicated over TCP/IP sockets by intercepting the following eight UNIX system calls: socket(), accept(), connect(), listen(), close(), read(), writev() and select(). We keep track, locally, of each socket that either the CORBA client or server opens based on the sequence of system calls executed (See Figure 4-2). For instance, if a specific socket file descriptor appears within the accept() call, we associate the descriptor with a server-side socket because only the server-side logic of a CORBA application would invoke the accept() call (in order to accept connections from potential clients). Most of our proactive-recovery logic is implemented within the intercepted read(), writev() and select() calls because all of the communication in CORBA is connection-oriented, and these calls capture the message-exchange interface between the CORBA client and server.

4.2.2 The MEAD Proactive Fault-Tolerance Manager

The Proactive Fault-Tolerance Manager is embedded within the server-side and client-side Interceptors and is thus collocated with the CORBA client and server respectively. At the server side, its responsibility is to monitor resource usage and trigger proactive fault-recovery mechanisms whenever it senses that resource usage has exceeded a predefined threshold. At the client-side, its responsibility is to redirect the client away from the failing replica and to a non-faulty server replica. We implement proactive recovery using a two-step threshold-based scheme similar to the soft hand-off process employed in cellular systems [39]. When a replica's resource usage exceeds our first threshold, *e.g.*, when the replica has used 80% of its allocated resources, the Proactive Fault-Tolerance Manager at that replica requests the Recovery Manager to launch a new replica. If the replica's resource usage exceeds our second threshold, *e.g.*, when 90% of the allocated resources have been consumed, the Proactive Fault-Tolerance Manager at that replica can initiate the migration of all its current clients to the next non-faulty server replica. Future implementations of the Fault-Tolerance Manager will incorporate adaptive, rather than preset, thresholds.

4.2.3 The MEAD Recovery Manager

Within our proactive dependability framework, the Recovery Manager is responsible for launching new server replicas to restore the application's resilience after a server replica or a node crashes. Thus, the Recovery Manager needs to have up-to-date information about the server's degree of replication (*i.e.*, the number of replicas). To propagate the replicated server's group membership information to the Recovery Manager, we ensure that new server replicas join a unique server-specific group as soon as they are launched. By subscribing to that group, the Recovery Manager can receive membership-change notifications. For instance, if a server replica crashes, the Recovery Manager receives a membership-change notification, too, and can launch a new replica to replace the failed one. The Recovery Manager also receives messages from the Proactive Fault-Tolerance Manager whenever the Fault-Tolerance Manager anticipates that a server replica is about to fail. These proactive fault-notification messages can also trigger the Recovery Manager to launch a new replica to replace the one that is expected to fail. Thus, the Recovery Manager participates in both reactive and proactive fault-recovery schemes.

4.3 Proactive Fault-Recovery Schemes

The Proactive Fault-Tolerance Manager implements proactive recovery through five different schemes: GIOP LOCATION_FORWARD *Reply* messages, GIOP NEEDS_ADDRESSING_MODE *Reply* messages, MEAD's client-side fail-over messages, MEAD's server-side fail-over messages and finally, through MEAD's topology-aware messages. Each of these messages performs two functions: first, it invokes a proactive state-checkpoint by updating state information across the server replicas using the underlying group communication system, and second, it redirects clients (away from the failing replica) to the newly elected primary replica. The mechanisms for proactive checkpointing are the same across all five schemes. The schemes primarily differ in the way that they redirect clients to new server replicas as explained below.

4.3.1 GIOP LOCATION_FORWARD Messages

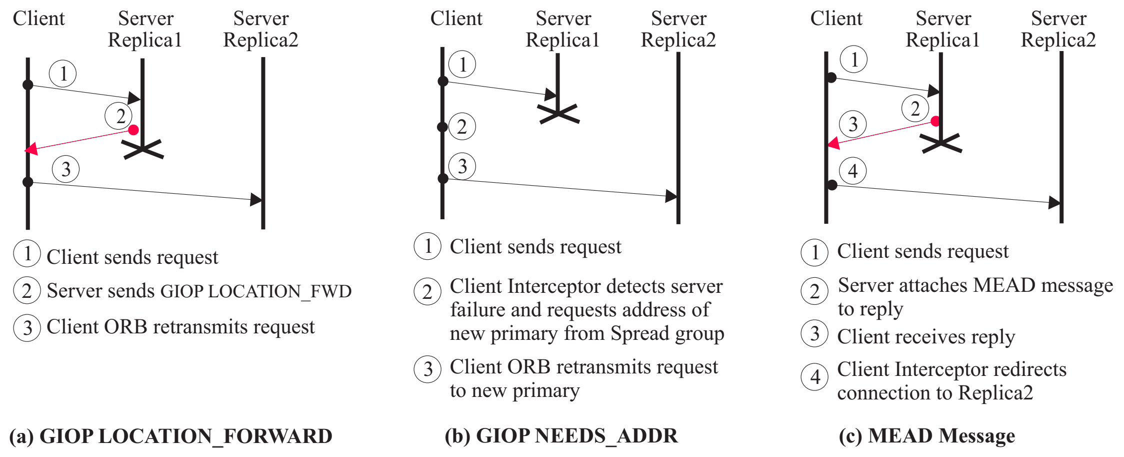
CORBA's GIOP specification [25] defines a LOCATION_FORWARD *Reply* message that a server can use to redirect its current clients to an alternative server location (See Chapter 2). The body of this *Reply* message consists of an Interoperable Object Reference (IOR) that uniquely identifies the CORBA object at the new server location.

To implement this scheme, we intercept the IOR returned by the Naming Service when each server replica registers its objects with the Naming Service. We then broadcast these IORs, through the group communication system, to the Fault-Tolerance Managers collocated with the server replicas. Thus, each Fault-Tolerance Manager hosting a server replica "knows" the references of all of the other replicas of that server.

When the server-side MEAD Fault-Tolerance Manager senses that its replica is about to crash, it suppresses its replica's normal GIOP Reply message to the client, and instead sends a LOCATION_FORWARD *Reply* message containing the address of the next available server replica. The client ORB, on receiving this message, transparently retransmits the client request to the new replica without involving the hosted client application (see Figure 4-3 a).

The main advantage of this technique is that it does not require us to install an Interceptor at the client because the client ORB handles the retransmission through native CORBA mechanisms. However, the server-side Interceptor must maintain some system state because it needs to store the IOR entry of every object instantiated within the server. This scheme also incurs a high overhead because we need to parse incoming GIOP *Request* messages to extract the request identifier field so that we can generate corresponding LOCATION_FORWARD *Reply* messages that contain the corresponding request identifier and the correct object key. One optimization that we add to this scheme is the use of a 16-bit hash of the object key to facilitate the easy look-up of the IORs, as opposed to a byte-by-byte comparison of the object key (that was typically 52 bytes for our test application).

Figure 4-3. Client-side fail-over schemes.


Client
Server Replica1
Server Replica2
1 Client sends request
2 Server sends GIOP LOCATION_FWD
3 Client ORB retransmits request
(a) GIOP LOCATION_FORWARD
Client
Server Replica1
Server Replica2
1 Client sends request
2 Client Interceptor detects server failure and requests address of new primary from Spread group
3 Client ORB retransmits request to new primary
(b) GIOP NEEDS_ADDR
Client
Server Replica1
Server Replica2
1 Client sends request
2 Server attaches MEAD message to reply
3 Client receives reply
4 Client Interceptor redirects connection to Replica2
(c) MEAD Message

4.3.2 GIOP NEEDS_ADDRESSING_MODE Messages

The GIOP NEEDS_ADDRESSING_MODE *Reply* message directs the client to supply more addressing information, and usually prompts the client ORB to resend the request. We used this scheme to investigate the effect of suppressing abrupt server failures from the client application, in case the server does not have enough time to initiate proactive recovery before it fails. We detect abrupt server failures when the read() call at the client Interceptor returns an End-Of-File (EOF) response. At this point, we contact the Fault-Tolerance Manager at the server replicas (using the group communication system) to obtain the address of the next available replica. The first server replica listed in the group-membership list responds to the client's request (see Figure 4-3 b). If the client does not receive a response from the server group within a specified time (we used a 10ms timeout), the blocking read() at the client-side times out, and a CORBA exception is propagated up to the client application. If, on the other hand, we receive the address of the next available replica, we then redirect the current client connection to the new replica at the Interceptor level, and fabricate a NEEDS_ADDRESSING_MODE *Reply* message that causes the client-side ORB to retransmit its last request over the new connection.

The advantage of this technique is that it masks communication failures from the client application, but it sometimes takes the client longer to recover from the failure, as compared to a reactive scheme where we would expose the client to the failure and let it recover on its own. We do not recommend this technique because it sometimes increases the average fail-over time, and it is based on the assumption that an EOF response corresponds to an abrupt server failure, which might not always the case on all operating systems and under all conditions.

4.3.3 MEAD Client-Side Fail-over Messages

In this scheme, the Proactive Fault-Tolerance Manager intercepts the listen() call at the server to determine the port on which the server-side ORB is listening for clients. We then broadcast this information over the group communication system so that the Proactive Fault-Tolerance Manager at each server replica knows the hostname and the port of the other replicas in the group. Whenever group-membership changes occur (and are disseminated automatically over the group communication system), the first replica listed in the group-membership message sends a message that synchronizes the listing of server replicas across the group.

When MEAD detects that a replica is about to fail, it sends the client-side Proactive Fault-Tolerance Manager a MEAD proactive fail-over message containing the address of the next available replica in the group (see Figure 4-3c). We accomplish this by piggybacking regular GIOP *Reply* messages onto the MEAD proactive fail-over messages. When the client-side Interceptor receives this combined message, it extracts (the address in) the MEAD message to redirect the client connection to the new replica so that subsequent client requests are sent to the new replica. The Interceptor then transmits the regular GIOP *Reply* message up to the client application. The redirection of existing client connections is accomplished by the Interceptor opening a new TCP socket, connecting to the new replica address, and then using the UNIX `dup2()`[1] call to close the connection to the failing replica, and point the connection to the new address (an alternative to this scheme would be to use the migratory TCP protocol [34]).

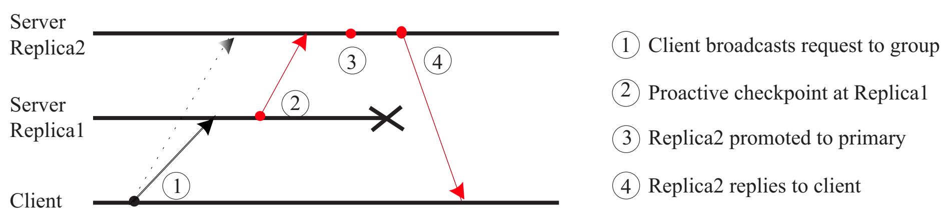
This scheme reduces the average fail-over time because, unlike the previous two schemes, it avoids the retransmission of client requests. The scheme also incurs a low overhead since we do not need to parse GIOP messages and keep track of IORs. However, this scheme does not readily support replicated clients to track the group communication messages (and would make the support of multi-tiered applications harder).

4.3.4 MEAD Server-Side Fail-over Messages

In this scheme, all of the communication between the client and the server is channeled over the group communication system ensuring a consistent ordering of messages across all the server replicas. The interception mechanisms used in this scheme are described in detail elsewhere (specifically, in the paper on MEAD's versatile dependability [35]), and which support both active and passive replication styles. The schemes that we described earlier only support passively replicated systems, so we do not need to provide strict message ordering guarantees because the client only communicates directly with one replica. Clients address the server using the server's group identifier; therefore, no proactive fail-over messages are sent to the client. Instead, the proactive fail-over simply involves invoking a proactive state-checkpoint and is treated in a similar manner to a server replica voluntarily leaving the group.

1. `dup2()` is a UNIX system call that is used to create copies of file descriptors.

Figure 4-4. Server-side fail-over scheme.

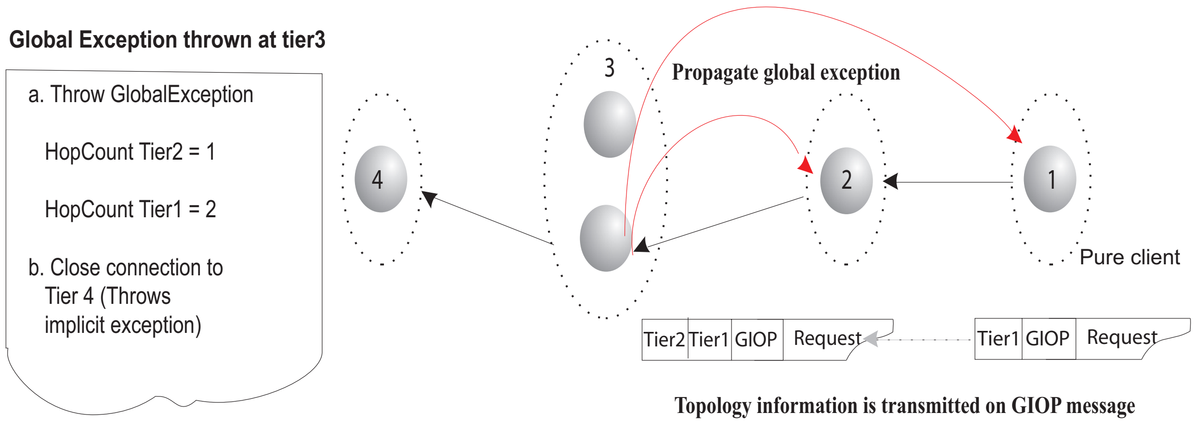

Server Replica2
Server Replica1
Client
1
2
3
4
1 Client broadcasts request to group
2 Proactive checkpoint at Replica1
3 Replica2 promoted to primary
4 Replica2 replies to client

4.3.5 MEAD Topology-Aware Messages

The previous sections (4.3.1-4.3.4) refer to two-tiered client-server applications. Here, we investigate the effect of invalidating an entire invocation in a multi-tiered application by simultaneously broadcasting an exception message to all the tiers involved as opposed to propagating exceptions sequentially from one tier to the next. This global exception-handling strategy should help reduce the latency associated with exceptions propagating sequentially from tier-to-tier. This scheme, unlike the others described in the two-tier case, is not transparent to the application. We introduce a *GlobalException* type that needs to be handled by every CORBA object in the system. The application programmer specifies the error-handling routine that needs to be executed once this exception is received. The *GlobalException* contains a field that flags the tier that failed using a hop-count. For example, if the exception was raised at tier 3, the hop-count at tier 3 is zero, the hop-count at tier 2 is one, and the hop-count at tier 1 is two.

The main challenge of this technique is extracting some knowledge of the system topology. At the Interceptor level, all that the Fault-tolerance Manager sees is a number of incoming and outgoing sockets. It has no idea whether the messages on a particular socket originated from a neighboring client or one farther downstream. To get some notion of system topology in our prototype, we prepend information about the Spread group identifier onto each GIOP message. Because we assume that all processes are single-threaded, if we see an incoming message on one socket followed immediately by an outgoing request on a different socket, we assume that this is a nested invocation and attach the relevant topology information (See Figure 4-5).

Figure 4-5. Learning system topology.


Global Exception thrown at tier3
a. Throw GlobalException
HopCount Tier2 = 1
HopCount Tier1 = 2
b. Close connection to Tier 4 (Throws implicit exception)
4
3
Propagate global exception
2
1
Pure client
Tier2 Tier1 GIOP Request
Tier1 GIOP Request
Topology information is transmitted on GIOP message

A more efficient approach would be to map the logical dependencies in an application to a run-time representation that can be consulted by the proactive fault-recovery mechanisms, as discussed in Section 3.3.3. The method we adopted for this implementation merely serves as a proof-of-concept. When the proactive fault-recovery mechanisms sense that a replica is about to fail, they suppress the request from the server application, and instead broadcast a CORBA *GlobalException message* to all the tiers involved in the invocation.

All of the proactive fault-recovery schemes discussed in this section help systems to meet real-time deadlines by lowering the fault-recovery times for a class of predictable faults. However, they differ in the means of accomplishing this as summarized in Table 4-1.

Table 4-1. A comparion of proactive fault-recovery schemes.

Scheme	**Replication Style**	**Client-side vs. Server side fail-over**	**Amount of state kept by FT-manager**	**Transparent to client application**
LOCATION_FORWARD	Passive	Client-side	High	Yes
NEEDS_ADDRESSING_MODE	Passive	Client-side	Low	Yes
MEAD client-side	Passive	Client-side	Moderate	Yes
MEAD server-side	Passive and Active	Server-side	Moderate	Yes
Topology-aware	Passive	Client-side	High	No

5 Experimental Evaluation

We ran our initial experiments on five Emulab [38] nodes with 850MHz processors, 512MB RAM, and the RedHat Linux 9 operating system. Our test application was a simple CORBA client implemented over the TAO ORB (ACE and TAO version 5.4) [10] that requested the time-of-day at 10ms intervals from one of three warm-passively replicated CORBA servers. Each experiment covered 10,000 client invocations. We activated a specific kind of resource-exhaustion fault, namely, a memory-leak, when the primary server replica responded to its first client request. The resource-exhaustion fault led to approximately one server failure for every 250 client-invocations. The Proactive Fault-Tolerance Manager constantly monitored the memory usage on the faulty server replica, and triggered proactive recovery when the resource usage reached a preset threshold, for instance, when 80% of the allocated memory was consumed. Our multi-tiered application was an extension of our two-tier application in which a client sends requests to a server furthur upstream, *e.g.*, a server in tier 5 (all the intermediate client/server tiers between tier 1 and tier 5 simply forward messages between the client in tier 1 and the server in tier 5).

5.1 Results

We compare the jitter and performance of both the proactive and the reactive fault-recovery schemes in systems that use either client-side or purely server-side fail-over techniques. We also evaluate the impact of throwing *GlobalException*s in multi-tiered systems.

5.1.1 Client-Side Fail-over Systems

We investigate two different reactive schemes. In the first reactive (no-cache) scheme, the client waits until it detects a server failure before contacting the CORBA Naming Service for the address of the next available server replica. In our second reactive (cache-based) scheme, the client first contacts the CORBA Naming Service and obtains the addresses of the three server replicas, and stores them in a collocated cache. When the client detects the failure of a server replica, it moves on to the next entry in the cache, and only contacts the Naming Service once it exhausts all of the entries in the cache. For both the reactive and proactive schemes, we measured the following three metrics (see Table 5-1): the percentage increase in client-server round-trip times over the reactive schemes, the percentage of failures exposed to the client application per server-side failure, and the fail-over times. In our proactive schemes, we additionally measured the effectiveness of failing over clients at different thresholds.

Table 5-1. Overhead and fail-over times.

Recovery Strategy	**Increase in Round-Trip-Time(RTT)** (%)	**Client Failures** (%)	**Fail-over Time**			
			Average (ms)	*Standard Deviation σ (ms)*	*Worst-case - Average + 3σ (ms)*	*% change in Worst-case time*
Reactive without cache	baseline	100%	10.08	1.17	13.59	baseline
Reactive with cache	0%	146%	10.37	1.33	14.36	+5.7%
NEEDS_ADDRESSING_MODE	8%	25%	9.68	1.03	12.77	-6.0%
LOCATION_FORWARD	90%	0%	8.66	0.38	9.80	-27.8%
MEAD message	3%	0%	2.65	0.17	3.16	-76.7%

5.1.2 Number of Client-side Failures

In the no-cache reactive fault-recovery scheme, there was an exact 1:1 correspondence between the number of observed failures at the client and the number of server-side failures. The client-side failures that we observed were purely CORBA COMM_FAILURE exceptions, which are raised when a replica fails after the client has successfully established a connection with the replica. The cache-based reactive scheme experienced a higher failure rate. There was a 1:1 correspondence between the number of server-side failures and the number of COMM_FAILURE exceptions observed by the client. In addition to COMM_FAILURE exceptions, the client also experienced a number of CORBA TRANSIENT exceptions that occur when the client accesses a stale replica reference. Stale cache references occurred when we refreshed the cache before a faulty replica has had a chance to restart and register itself with the Naming Service, thereby leaving its old invalid reference in the cache. This problem can be reduced by staggering the cache-refresh process over time, instead of refreshing all of the cache references in one sweep.

In the NEEDS_ADDRESSING_MODE scheme, which is equivalent to a proactive recovery scheme with insufficient advance warning of the impending failure, we observed eleven client-side failures. These occurred when the client requested the next available replica at the point of the previous replica's crash, but before the replica's crash had been notified to everyone in the server group; because there was as yet no agreed-upon primary replica, the blocking read at the client timed out and the client caught a COMM_FAILURE exception. For the proactive schemes with sufficient warning of the impending failure, *i.e.,* thresholds below 100% (these correspond to the LOCATION_FORWARD scheme and the MEAD proactive fail-over message scheme), the client does not catch any exceptions at all!

5.1.2.1 Overhead

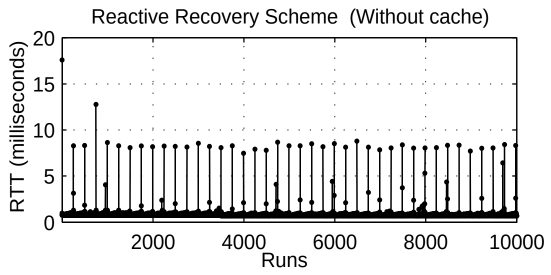
We measured the overhead in terms of the percentage increase in client-server round-trip times (RTT). We define round-trip time as the amount of time that elapses from the time that the client application sends a *Request* to the time that it receives a *Reply* from the server. The overhead in the reactive schemes, which averaged 0.75ms, served as our baseline reference. The scheme which used LOCATION_FORWARD messages to trigger proactive recovery incurred an overhead of about 90% over the baseline round-trip time. This overhead resulted from parsing GIOP messages so that we could keep track of object keys and *request_id*s (see Chapter 2), and fabricate the appropriate GIOP messages that are needed to forward requests to the next available replica. The NEEDS_ADDRESSING_MODE scheme's overhead was only 8% higher than the baseline because we did not need to keep track of object keys. The scheme in which we used MEAD messages introduced an overhead of about 3% over the baseline client-server round-trip time.

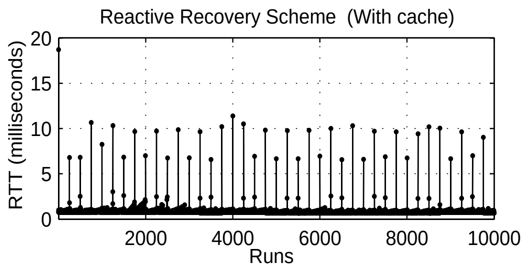
The communication overhead introduced by the proactive schemes depends on the frequency with which proactive recovery is invoked. The additional messages sent by MEAD's proactive fault-recovery framework, in the event of a failure, typically add up to 100-150 bytes per client/server connection. Since systems typically experience more non-faulty, rather than faulty, behavior, the overall communication overhead introduced by our approach is reasonable. The differences in memory and CPU usage for our application were not significant. However, we expect that as the server hosts more objects, the overhead of the LOCATION_FORWARD scheme will increase significantly over the other schemes because it maintains an IOR entry for each instantiated object.

5.1.2.2 Worst-case Fail-over Times

The fail-over time includes both the fault-detection time and the fault-recovery time. The initial transient spike shown on each graph represents the first call to the CORBA Naming Service (see Figure 5-1). In the reactive scheme, where we did not cache server replica references, the client first experienced a COMM_FAILURE exception when the server replica dies; the COMM_FAILURE exception takes about 1.7ms to register at the client. The client then incurred a spike of about 8.4ms to resolve the next server replica's reference, resulting in a worst-case fail-over time of 13.59ms (average $+3\sigma$). In the case where we cache server references, we experienced about one TRANSIENT exception, of about 1.1ms, for every two COMM_FAILURE exceptions. This led to an increased worst-case fail-over time of about 14.36ms.

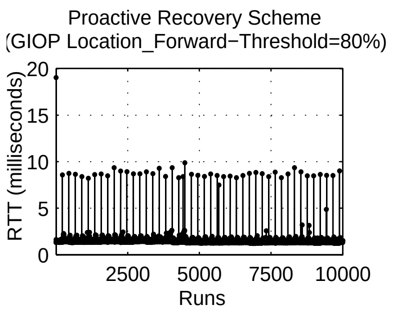
Figure 5-1. Reactive client-side fault-recovery schemes.

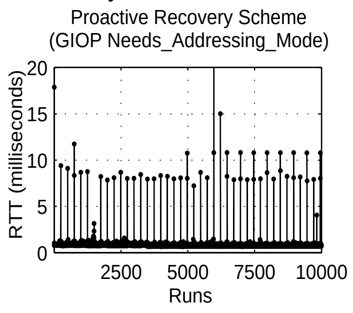


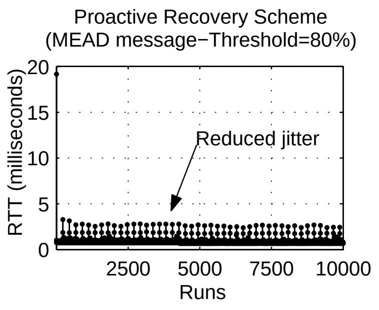


The proactive scheme that used LOCATION_FORWARD messages incurred a worst-case fail-over time of 9.8ms (27% lower than the no-cache reactive scheme) because when the client ORB receives the LOCATION_FORWARD, it transparently resends the request to the next server replica. When we used a MEAD client-side message, the worst-case fail-over time was about 3.16 ms (76% below the no-cache reactive scheme), because we avoided request retransmissions and incurred an overhead only when redirecting a connection to a new server (see Figure 4). Finally, for the NEEDS_ADDRESSING_MODE scheme, the worst-case fail-over time is about 12.77ms (6% below the no-cache reactive scheme), which is the time taken to contact the server group, redirect the client connection, and retransmit the request to the new server.

Figure 5-2. Proactive client-side fault-recovery schemes.

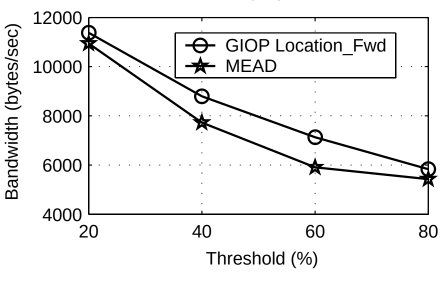






5.1.2.3 Effect of Varying Threshold

For the proactive schemes, we analyzed the effect of varying the proactive-recovery threshold. Our results showed that if the threshold is too low, the overhead in the system increases due to the unnecessary fail-over of clients to a new server. For example, the group communication bandwidth between the servers is about 6,000 bytes/sec at an 80% threshold, but



increases to about 10,000 bytes/sec at a 20% threshold. The increase in bandwidth happens because we are restarting the servers more often at lower thresholds and more bandwidth is consumed in reaching consensus on the current group membership. The best performance is achieved by delaying proactive recovery so that our framework has just enough time to redirect clients away from the faulty server replica to a non-faulty server replica in the system.

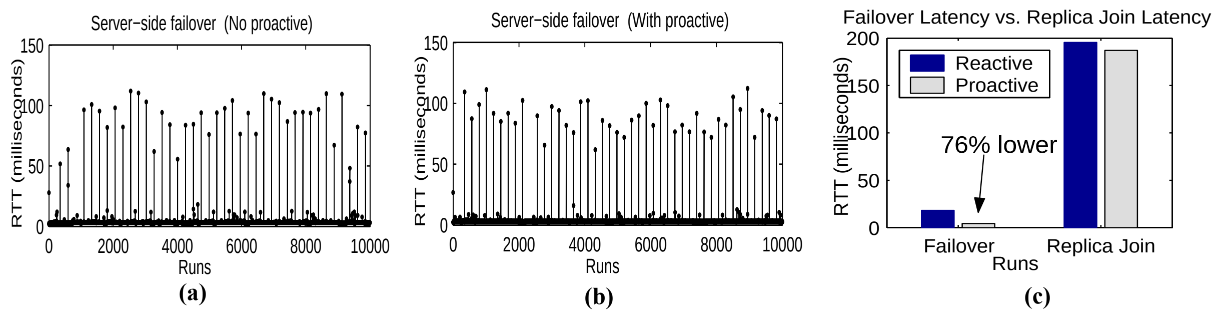
5.1.2.4 Jitter

In both the fault-free and the faulty (reactive and proactive) schemes, we observed spikes that exceeded our average round-trip times by 3 standard deviations. These outliers occurred between 1% and 2.5% of the time. In the fault-free run, the highest spike we observed was 2.3ms (these spikes are potentially due to file system journaling and scheduling done by the operating system.). We also observed one large spike of about 30ms that occurred 0.01% of the time in the GIOP LOCATION_FORWARD scheme. This spike occurred when we set the rejuvenation threshold to below 80%.We suspect that the spike happens when a client sends a request to a newly restarted server that is updating its group membership information. The highest spike that we observed with the MEAD proactive messages was 6.9ms at the 20% proactive-recovery threshold.

5.1.3 Server-Side Fail-over Systems

The worst-case (*i.e.*, average + 3σ) time needed to execute the fault-recovery routines in the reactive case was 18.135ms, whereas the time taken by the proactive fault-recovery schemes was 4.335ms (about a 76% reduction in fail-over latency). However, this scheme experienced a significant amount of jitter, which typically occurred whenever we launched a new server replica and joined the server replica group, as shown in Figure 5-3 c. The worst-case replica-join time for the reactive and proactive schemes was 195ms and 187ms respectively.

Figure 5-3. Server-side fail-over.

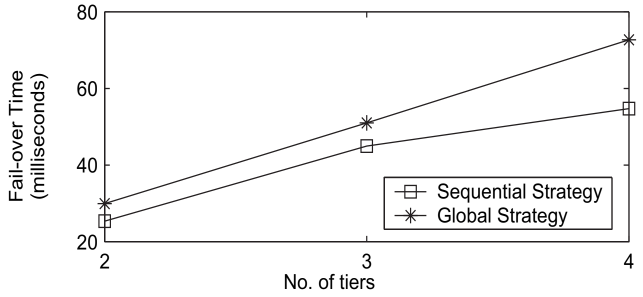


Because this scheme enforces strict message-ordering guarantees between the client and the server in order to ensure consistent behavior across all of the replicas, no messages can be delivered within the system until all the nodes have a consistent view of current group membership. In our experiments, we observed that the latencies experienced when a replica crashes were relatively low. However, when a new replica (even a backup) joins the group, the delays we experienced were significant, and sometimes exceeded 100ms. We believe that this behavior may be due to the flow-control mechanisms that the Spread group communication protocol uses to guarantee consistent message ordering. The flow-control mechanisms adjust the message transmission rate of Spread to match the speed of the slowest receiver so the system may experience a "hiccup" whenever a new replica connects to the group. In addition, we did not tune any of the group communication timeouts; we expect performance improvements with the appropriate setting of Spread's tunable parameters.

5.1.4 Topology-Aware Systems

In this scheme, we compared the effectiveness of invalidating an operation from end-to-end as soon as we detect that a server is about to fail, as opposed to allowing the exception to propagate sequentially from tier-to-tier. For the multi-tiered version of our test application, we imposed a 10ms processing delay at each tier to study the effect that service times had on the effectiveness of our system. We set the rejuvenation threshold to 80%. Our results show that the propagating the exception sequentially performed better than the global proactive notification scheme (see Figure 5-4). However, we expect that as the exception-handling times at each tier increases, a global proactive strategy would perform better.

Figure 5-4. Exception-handling times in a multi-tiered system..



6 Conclusion & Future Work

In this thesis, we describe the development of a transparent proactive fault-recovery framework for CORBA applications, and show that proactive recovery can indeed provide bounded temporal behavior in the presence of certain kinds of faults, thereby enabling the development of real-time, fault-tolerant distributed systems. Our preliminary results show that the use of MEAD’s proactive fail-over messages can yield a promising 76% reduction in worst-case fail-over times over a traditional reactive recovery scheme.

The server-side proactive fault-recovery scheme experienced a significant jitter whenever we relaunched replica processes. We suspect that this may be due to the flow-control mechanisms exerted by the group communication system. We also show that if we trigger proactive recovery too often, the additional overhead of migrating clients too frequently can quickly negate the benefits of proactive recovery. The ideal scenario is to delay proactive recovery so that the proactive dependability framework has just enough time to redirect clients and objects away from the faulty server replica to a non-faulty server replica in the system. This thesis also describes how to exploit knowledge of the underlying system topology to invalidate end-to-end operations in multi-tiered systems.

As part of our future directions, we plan to extend our proactive dependability framework to include more sophisticated failure prediction and use real-world test data to evaluate the effectiveness of our approach. We also plan to integrate adaptive thresholds into our framework rather than relying on preset thresholds supplied by the user. Other directions for proactive fault-recovery would be to target more than resource-exhaustion faults (the primary focus of our failure-prediction strategy in this thesis).

Bibliography

[1] Amir, Y., Danilov, C. and Stanton, J. A low latency, loss tolerant architecture and protocol for wide area group communication. In *International Conference on Dependable Systems and Networks*, pages 327–336, New York, NY, June 2000.

[2] Bernat, G., Burns, A. and Liamosi, A. Weakly hard real-time systems. *IEEE Transactions on computers*, Volume 50-Issue 4, pages 308-321, April 2001.

[3] Bobbio, A. and Sereno, M. Fine grained software rejuvenation models. In *Computer Performance and Dependability Symposium*, pages 4–12, Durham, NC, September 1998.

[4] Budhiraja, N., Marzullo, K., Schneider, F. B. and Toueg, S. The primary-backup approach. In *S. Mullender, editor, Distributed Systems. ACM Press - Addison Wesley*, 1993.

[5] Castro, M. and Liskov, B. Proactive recovery in a Byzantine fault-tolerant system. In *Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[6] Cukier, M., Ren, J., Sabnis, C., Sanders, W. H., Bakken, D. E., Berman, M. E., Karr, D. A., and Schantz, R. AQuA: An adaptive architecture that provides dependable distributed objects. In *IEEE Symposium on Reliable Distributed Systems*, pages 245-253, West Lafayette, IN, October. 1998.

[7] Ensel, C. and Keller, A. Managing application service dependencies with XML and the resource description framework. In *IFIP/IEEE International Symposium on Integrated Network Management Proceedings*, pages 661-674, Seattle, WA, May 2001.

[8] Felber, P. *The CORBA Object Group Service: A service approach to object groups in CORBA*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, 1998.

[9] Garg, S., Van Moorsel, A., Vaidyanathan, K. and Trivedi, K. A methodology for detection and estimation of software aging. In *International Symposium on Software Reliability Engineering*, pages 283–292, Paderborn, Germany, November 1998.

[10] Gokhale, A. and Schmidt, D.C. Techniques for optimizing CORBA middleware for distributed embedded systems. In *Joint Conference of the IEEE Computer and Communications Societies*, pages 513-521, New York, NY, March 1999.

[11] Huang, Y., Kintala, C., Kolettis, N. and Fulton, N. Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, June 1995.

[12] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[13] Kaff, D.A., Rodrigues, C., Krishnamurthy, Y., Pyarali, I. and Schmidt, D.C. Application of the QuO Quality-of-Service Framework to a Distributed Video Application. In *International Symposium on Distributed Objects and Applications,* pages 299-308, Rome, Italy, September 2001.

[14] Lamport, L. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS),* Volume 4, Issue 3, pages 382 - 401, July 1982.

[15] Levine, J. R. *Linkers and Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[16] Lin, T. Y. and Siewiorek D. P. Error log analysis: Statistical modeling and heuristic trend analysis. In *IEEE Transactions on Reliability*, Volume 39, Issue 4, pages 419-432, October 1990.

[17] Maffeis, S. *Run-time Support for Object-oriented Distributed Programming.* PhD thesis, University of Zurich, February 1995.

[18] Marchetti, C., Verde, L. and Baldoni, R. CORBA request portable interceptors: A performance analysis. In *International Symposium on Distributed Objects and Applications*, pages 208–217, Rome, Italy, September 2001.

[19] Miller, R. and Tripathi, A. The guardian model for exception handling in distributed systems. In I*EEE Symposium on Reliable Distributed Systems*, pages 304- 313, Osaka, Japan, October 2002.

[20] Narasimhan, P., Dumitraş, T., Pertet, S., Reverte, C. F., Slember, J., and Srivastava, D. MEAD: Support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, Submitted 2003.

[21] Narasimhan, P. Trade-offs between real-time and fault-tolerance for middleware applications. In *Workshop on Foundations of Middleware Technologies*, Irvine, CA, November 2002.

[22] Narasimhan, P. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

[23] Natarajan, B., Gokhale, A., Yajnik, S. and Schmidt, D.C. DOORS: Towards high-performance fault tolerant CORBA. In *Second International Symposium on Distributed Objects and Applications*, pages 39-48, Antwerp, Belgium, February 2000.

[24] Nett, E., Gergeleit M. and Mock, M. Guaranteeing real-time behavior in adaptive distributed systems. In *Symposium on Large Scale Systems: Theory and Applications*, 1989.

[25] Object Management Group. *The CORBA/IIOP Specification Version 3.0.2.* OMG Technical Committee Document formal/2002-12-02, December 2002.

[26] Object Management Group. *Fault-Tolerant CORBA*. OMG Technical Committee Document formal/2002-09-29, September 2001.

[27] Object Management Group. *Portable Interceptors.* OMG Technical Committee Document formal/ 2001-12-25, December 2001.

[28] Object Management Group. *The CORBA Real-Time Specification Version 2.0.* OMG Technical Committee Document formal/2003-11-01, November 2003.

[29] Pertet, S. and Narasimhan, P. Proactive recovery in distributed CORBA applications. I*n International Conference on Dependable Systems and Networks*, Florence, Italy, June 2004.

[30] Powell, D. Distributed fault tolerance: Lessons from Delta-4. *IEEE Micro,* Volume 14, Issue 1, pages 36-47, February 1994.

[31] Puschner, P. and Koza, C. Calculating the maximum execution time of real-time programs. *Real Time Systems,* Volume 1, Issue 2, pages 159-176, September 1989.

[32] Ruggaber, R. and Seitz, J. A transparent network handover for nomadic CORBA users. In *International Conference on Distributed Computing Systems*, pages 499–506, Mesa, AZ, April 2001.

[33] Schneider, F.B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM computing Surveys (CSUR),* Volume 22, Issue 4, pages 299-319, December 1990.

[34] Sultan, F., Srinivasan, K., Iyer, D. and Iftode, L. Migratory TCP: Highly available Internet services using connection migration. In *International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[35] Dumitraş, T. and Narasimhan, P. An Architecture for Versatile Dependability. *DSN Workshop on Architecting Dependable Systems*, Florence, Italy, June 2004.

[36] Tewksbury, L.A., Moser, L.E. and Melliar-Smith, P.M. Live upgrades of CORBA applications using object replication. In *IEEE International Conference on Software Maintenance*, Florence, Italy, November 2001.

[37] Vilalta, R and Sheng, M. Predicting rare events in temporal domain. In *IEEE International Conference on Data Mining*, pages 474– 481, Maebashi City, Japan, December 2002.

[38] White, B., Lepreau, Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation,* pages 255–270, Boston, MA, December 2002.

[39] Wong, D. and Lim, T. J. Soft handoffs in CDMA mobile systems. *In Personal Communications*, Volume 4, Issue 6, pages 6–17, December 1997.