Proactive Recovery in Distributed CORBA Applications [∗]

Soila Pertet and Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3890
spertet@ece.cmu.edu, priya@cs.cmu.edu

Abstract

Unanticipated runtime events, such as faults, can lead to missed deadlines in real-time systems. While it is not always possible to know when a fault will occur, we can sometimes exploit pre-fault "symptoms" to initiate proactive (rather than reactive) fault-recovery. In this paper, we describe the design and implementation of a proactive recovery strategy for distributed CORBA applications in the presence of resource-exhaustion faults. We analyze the effect of different proactive recovery schemes on client/server response times, and we demonstrate a significant reduction, both in jitter and in the number of client-side failures.

1. Introduction

Middleware platforms, such as CORBA [11] 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. The push for Commercial-Off-The-Shelf (COTS) products, along with the recent support for "-ilities" within middleware, have furthered the adoption of middleware within mission-critical applications.

Mission-critical systems, like avionics applications and drive-by-wire systems, have stringent reliability and availability requirements, and need to exhibit both real-time and fault-tolerance properties. The Object Management Group (OMG) CORBA standard has attempted to address the needs of such applications by developing separate Real-Time [12] and Fault-Tolerance [11] specifications. However, due to some inherent conflicts in providing simultaneous real-time and fault-tolerance support [9], 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 arises because real-time systems require predictable end-to-end schedules and bounded temporal behavior in order to meet task deadlines; however, faults are unanticipated system events that can disrupt the predefined real-time schedule and result in missed task deadlines. Our preliminary measurements in this paper demonstrate that faults can disrupt a Real-Time CORBA application, and do lead to unbounded and variable fault-detection and fault-recovery times.

The MEAD (Middleware for Embedded Adaptive Dependability) system [10] 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 the MEAD system is its use of a *proactive dependability* framework to lower the impact of faults on a distributed application's real-time schedule. Proactive dependability involves designing and implementing mechanisms that can predict, with some confidence, when a failure might occur, and compensating for the failure even before it occurs. For instance, if we knew that a node had an 80% chance of failing within the next 5 minutes, we could migrate all of its hosted processes to another working node in the system, with the aim of meeting the application's real-time deadlines. This proactive process-migration would incur a lower penalty than if we had first waited for the process to crash before taking any action. The proactive approach is in sharp contrast to the reactive fault-tolerance strategy adopted by the Fault-Tolerant CORBA standard, which first waits to detect a fault, and then initiates fault-recovery mechanisms. In this paper, we show that, by developing such a proactive strategy, we can not only mask the occurrence of faults altogether from the clients of our

[∗] This work is partially supported by the NSF CAREER Award CCR-0238381, the DARPA PCES contract F33615-03-C-4110, and the General Motors Collaborative Laboratory at CMU.

application, but that we can also reduce the jitter and the latency "spikes" which manifest themselves when faults occur in the middle of a real-time operation.

There are two aspects to proactive dependability: first, the ability to predict failures and, second, the mechanisms to compensate for the failures, hopefully before the failures can actually occur. Of course, we recognize that some failures might occur so abruptly that we cannot possibly hope to predict them; for example, if someone accidentally unplugs the power supply of a node! However, a number of interesting computer-related faults are often preceded by a visible pattern of abnormal behavior that favors the use of some form of prediction. Typically, these failures result from gradual degradation faults, like resource exhaustion [5], or from transient and intermittent hardware faults, like disk crashes [7] or telecommunication equipment failures [16]. Because it is not always possible to predict failures for every kind of fault, proactive dependability complements (and for certain kinds of faults, out-performs), but does not replace, the traditional reactive fault-tolerance schemes.

Our hope is that we can provide bounded fault-recovery times for distributed real-time applications, in the face of a predictable class of faults, thereby allowing the support of both fault-tolerance and real-time properties, for at least these kinds of faults. In this paper, we focus on the mechanisms needed to implement proactive recovery in a distributed CORBA-based system. We do not attempt to develop a new failure-prediction technique; rather, we exploit relatively simple failure-prediction mechanisms within the MEAD system, and we show how to use these resulting predictions to develop a proactive recovery strategy that minimizes both the jitter and the "latency" spikes experienced by distributed applications in the presence of faults. This paper makes the following concrete contributions:

- Describes our *development* of a transparent, proactive recovery infrastructure for distributed CORBA applications,
- Discusses the *design choices* that we faced in implementing a proactive strategy, along with the impact of these choices on client-to-server response times and on client-side fail-over times, and
- Presents our *empirical evaluation* and measurements to quantify the overhead and performance of our proactive strategy, as compared with the classical reactive fault-tolerance approach.

We emphasize here that, while we employ CORBA as the vehicle for our investigations of proactive fault-tolerance, our techniques are, for the most part, independent of CORBA. In those cases where we exploit CORBA-specific mechanisms to accomplish our goals, we highlight our use of these mechanisms so that our proactive strategy can be readily extended to non-CORBA-based distributed applications.

2. Related Work

Huang et al [5] proposed a proactive approach, called software rejuvenation, for handling transient software failures. Software rejuvenation involves gracefully halting an application once errors accumulate beyond a specified threshold, and then restarting the application in a clean internal state. One example of this is killing and relaunching a process periodically in order to eliminate memory leaks. Subsequent work in software rejuvenation has focused on constructing rejuvenation policies that increase system availability and reduce the cost of rejuvenation [2, 4, 20]. We found that simply restarting a faulty server that has ongoing client transactions can lead to unacceptable jitter and missed real-time deadlines at the client. Our work focuses on the mechanisms needed to gracefully hand-off existing clients on faulty CORBA servers, at the onset of the rejuvenation threshold.

Castro and Liskov [3] describe a proactive recovery scheme for Byzantine fault-tolerant systems. In their system, clients detect a faulty primary 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 [14] consider the hand-off (similar to fail-over) problem in wireless CORBA systems. They develop a proxy platform that uses a modified CORBA Object Request Broker (ORB) to transparently hand-off mobile CORBA clients to wired CORBA servers. Instead, our approach implements transparent fail-over in wired environments using interception mechanisms [6] that do not require us to modify the ORB.

There is also a whole array of statistical fault-prediction techniques. Lin and Siewiorek [7] 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. Rubino [13] proposed a model that employs observations of the system behavior made during the operational phase in order to refine the predictions that system designers made concerning the system's dependability. Xu et al. [19] studied failure behavior in a networked Windows NT system by analyzing the system-reboot information contained in the event logs. Their results showed a strong likelihood of error propagation between servers within the same domain.

We reiterate that our research in this paper does *not* focus on fault-prediction techniques, but rather on how to exploit fault prediction in systems that have real-time deadlines; thus, while it might be possible for us to improve the performance of our system using more accurate, statistical (rather than heuristic) fault-prediction techniques, this is outside the scope of this paper, and forms a part of our future

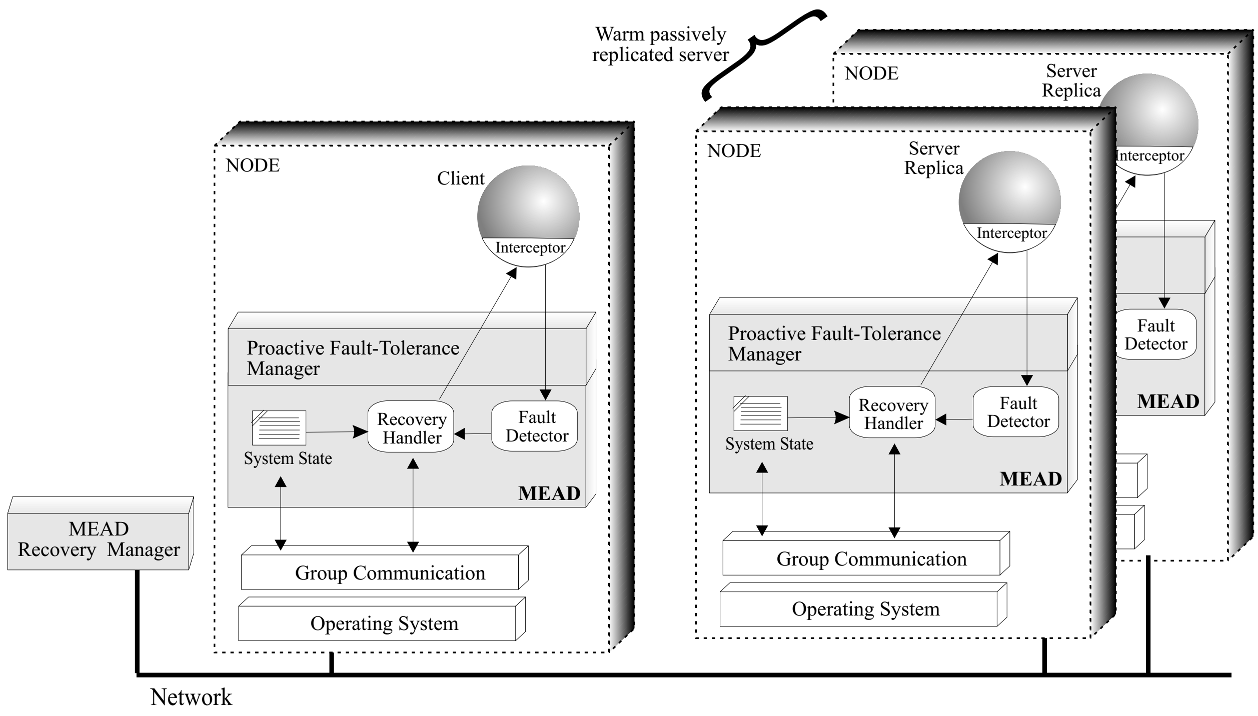


Figure 1. MEAD's proactive recovery framework.

directions. Instead, this paper serves to establish concretely that proactive recovery is effective and that it can provide bounded temporal behavior, in the presence of certain kinds of faults, thereby enabling us to support both fault-tolerance and real-time properties in distributed applications.

3. MEAD's Proactive Recovery Framework

MEAD's proactive recovery framework helps 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.

The CORBA standard specifies an Object Request Broker (ORB) that enables clients to request object implementations from servers seamlessly without having to worry about network-specific details. Our architecture leverages on the transparency provided by CORBA and makes use of some of the notification messages already supported by the CORBA standard. In our development of the MEAD system, we make the following assumptions:

- Operation in an asynchronous distributed system;
- Deterministic, reproducible behavior of the application and the ORB;
- 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.

Our proactive framework has three main components: the MEAD Interceptor, the MEAD Recovery Manager and the MEAD Proactive Fault-Tolerance Manager, as shown in Figure 1. MEAD exploits an underlying totally-ordered reliable group communication system, specifically, the Spread system [1], to obtain the reliable delivery and ordering guarantees required for consistent node-level and process-level membership.

3.1. 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 mechanism [11], 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 also due to some limitations [8] in their usage, such as the inability to block server replies without raising an exception at the application, we favored a transparent library interpositioning approach [6] instead. Library interpositioning provides us with greater flexibility, and allows us to implement proactive recovery transparently underneath an unmodified CORBA application running over an unmodified Linux kernel.

Library interpositioning is based on the runtime dynamic linking/loading facilities supported by most operating systems. At runtime, we can "preload" a custom library (the MEAD Interceptor) ahead of all of the other dynamically linked libraries in a process' address space. The idea is that runtime symbol resolution will first "discover" and use the symbols inside the custom Interceptor library, rather than the default ones provided by the operating system. Thus, if our custom Interceptor library provides an overridden definition of a system call, say `socket()`, then, the intercepted process will end up using our redefinition of the `socket()` call, rather than the default one.

Clearly, we might also need to access the default system definitions (for instance, our overridden `socket()` call might wish to do some additional processing before ultimately invoking the default `socket()` call); through calls such as `dlopen` and `dlsym`, our Interceptor can access the default system libraries, and retrieve the default symbol definitions.

Of direct relevance to fault-tolerance and fail-over are the system calls related to a CORBA application's communication over the network. CORBA uses a connection-oriented communication protocol known as General-Inter-ORB-Protocol (GIOP) to facillitate interoperability between ORBs developed by different ORB vendors. 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 of each socket that either the CORBA client or server opens, and identify the client-server communication sockets based on the sequence of system calls executed. 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.

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. The Interceptor uses the information from the remainder of the socket-related calls simply for differentiating between client and server sockets. Based on whether the invoking process is a client or a server, the intercepted `read()` and `writev()` calls have different overridden implementations.

For server sockets, we use `read()` to keep track of incoming client requests, and `writev()` to trigger our proactive recovery mechanisms. While we could use a separate thread within the Interceptor to implement our proactive-recovery mechanisms instead of embedding our logic in the `writev()` call, multithreading poses its own problems:

- Multithreading introduced a great deal of overhead in some of our earlier implementations, and sometimes led to nondeterministic behavior at the client based on the order in which the Interceptor's proactive-recovery thread executed at the server.
- Multithreading is a timer-driven (rather than event-driven) mechanism, and involved continuous periodic checking of resources. We do not really require this constant resource monitoring because proactive recovery complements, and does not replace, the reactive fault-tolerance mechanisms that handle periodic state transfer and fail-over. Therefore, proactive recovery needs to be triggered only when there are active client connections at the server.

For client sockets, we use the `read()` call to filter and interpret the custom MEAD messages that we piggyback onto regular GIOP messages. We use the `writev()` call to redirect client requests to non-faulty server replicas in the event of proactive fail-over. We also intercept the `select()` call in both the client and server, and add the group-communication socket (which allows our intercepted process to communicate over Spread) into the list of read-sockets examined by `select()`. This enables us to receive group communication messages without resorting to multithreading.

3.2. The MEAD Proactive Fault-Tolerance Manager

The Proactive Fault-Tolerance Manager is embedded within the server-side and client-side Interceptors. Because our focus is on resource-exhaustion faults, the Proactive Fault-Tolerance Manager monitors the resource usage at the server, and triggers proactive recovery mechanisms when it senses that resource usage has exceeded a predefined threshold. Here, "resource" refers loosely to any resource of interest ($e.g.$, memory, file descriptors, threads) to us that could lead to a process-crash fault if it was exhausted.

We implemented proactive recovery using a *two-step threshold-based scheme* similar to the soft hand-off process employed in cellular systems [18]. 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 in the group.

We faced two main challenges when developing the Proactive Fault-Tolerance Manager. The first challenge lay in determining *how often* to initiate proactive recovery. If we triggered fault-recovery too early, the additional overhead of unnecessarily failing over clients to non-faulty servers quickly negated the benefits of using a proactive strategy. On the other hand, if we waited too long to initiate recovery, the very purpose of a proactive strategy is lost because we simply did not allow the system enough time to fail-over client processes to a non-faulty server; in this case, the resulting fault-recovery ends up resembling a reactive strategy. The ideal time to trigger proactive recovery depends on a number of factors, such as the server's fault-rate as well as the amount of time required for fault-recovery. The fault-recovery time includes 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.

The second challenge lay in ensuring that the faulty replica reached a quiescent state before it could be restarted. We found that simply restarting a server replica when the rejuvenation threshold was reached caused clients to experience a large "spike" in their measured round-trip times, due to the resulting CORBA exceptions that are handled on the client side. To reduce these "spikes," we used proactive recovery messages to seamlessly redirect existing clients to the next non-faulty server in the group. These proactive recovery messages are described in detail in Section 4.

3.3. The MEAD Recovery Manager

Within our proactive dependability framework, the MEAD Recovery Manager is responsible for launching new server replicas that 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 ($i.e.$, 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 the same group, the Recovery Manager can receive membership-change notifications. For instance, if a server replica crashes, the Recovery Manager receives a membership-change notification from Spread, and can launch a new replica to replace the failed one.

The Recovery Manager also receives messages from the MEAD 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. We recognize that our Recovery Manager is currently a single point-of-failure – future implementations of our framework will allow us to extend our proactive mechanisms to the Recovery Manager as well.

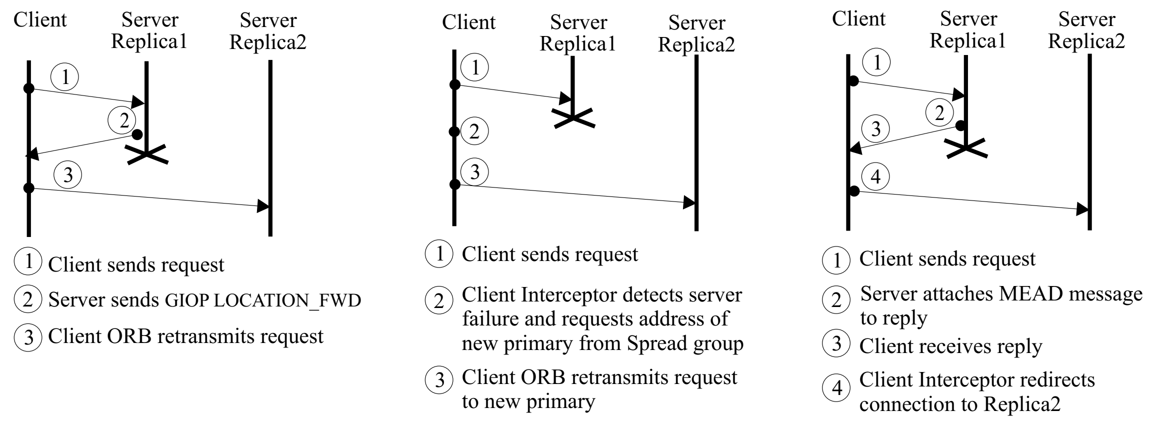
4. Proactive Recovery Schemes

The Proactive Fault-Tolerance Manager implements proactive recovery through three different schemes: GIOP LOCATION_FORWARD `Reply` messages, GIOP NEEDS_ADDRESSING_MODE `Reply` messages and, finally, through MEAD's own proactive fail-over messages. For each of these schemes, we describe how the Proactive Fault-Tolerance Manager transfers clients connected to a faulty replica over to a non-faulty replica, along with the associated trade-offs. Each scheme assumes that the application uses CORBA's persistent object key policies to uniquely identify CORBA objects in the system. Persistent keys transcend the life-time of a server-instance and allow us to forward requests easily between server replicas in a group. Persistent object keys also eliminate any non-determinism due to differences in object keys across different replicas.

4.1. GIOP LOCATION_FORWARD Messages

CORBA's GIOP specification [11] defines a LOCATION_FORWARD `Reply` message that a server can use to redirect its current clients to an alternative server location. 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 Spread group communication system, to the MEAD Fault-Tolerance Managers collocated with the server replicas. Thus, each MEAD Fault-Tolerance Manager hosting a server replica is populated with the references of all of the other replicas of the 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


Client
Server Replica1
Server Replica2
1 Client sends request
2 Server sends GIOP LOCATION_FWD
3 Client ORB retransmits request
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
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

(a) GIOP LOCATION_FORWARD **(b) GIOP NEEDS_ADDR** **(c) MEAD Message**

Figure 2. Sequence diagrams of proactive recovery schemes.

retransmits the client request to the new replica without notifying the client application (see Figure 2a). The main advantage of this technique is that it does not require 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 an IOR entry is required for every object instantiated within the server; in addition, the client ORB has to resend the request, leading to increased bandwidth. This scheme also incurs a high overhead because we need to parse incoming GIOP `Request` messages to extract the `request_id` field so that we can generate corresponding LOCATION_FORWARD `Reply` messages that contain the correct `request_id` and object key. One of the optimizations that we add to this scheme is the use of a 16-bit hash of the object key that facilitates the easy look-up of the IORs, as opposed to a byte-by-byte comparison of the object key (which was typically 52 bytes in our test application).

4.2. NEEDS_ADDRESSING_MODE Messages

The GIOP NEEDS_ADDRESSING_MODE `Reply` message [11] 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 MEAD Fault-Tolerance Manager at the server replicas (using the underlying group communication system) to obtain the address of the next available replica. The first server replica listed in Spread's group-membership list responds to the client's request (see Figure 2b).

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 COMM_FAILURE 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 is not always the case.

4.3. MEAD Proactive 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 Spread 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 Spread), the first replica listed in the Spread group-membership message sends a message that synchronizes the listing of active servers 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 2c). We accomplish this by piggybacking regular GIOP `Reply` messages onto the MEAD proactive failover 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()` 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 [15]). This scheme reduces the average failover 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.

5. Empirical Evaluation

We ran our initial experiments on five Emulab [17] nodes with the following specifications: 850MHz processor, 512MB RAM, and the REDHAT Linux 9 operating system. For our test application, we used a simple CORBA client implemented over the TAO ORB (ACE and TAO version 5.4) that requested the time-of-day at 1ms intervals from one of three warm-passively replicated CORBA servers managed by the MEAD Recovery Manager. 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 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.

We compared our proactive schemes against two traditional reactive recovery schemes. In the first scheme, the client waited until it detected a server failure before contacting the CORBA Naming Service for the address of the next available server replica. In our second scheme, the client first contacted the CORBA Naming Service and obtained the addresses of the three server replicas, and stored them in a collocated cache. When the client detected the failure of a server replica, it moved on to the next entry in the cache, and only contacted the CORBA Naming Service once it exhausted all of the entries in the cache.

5.1. Fault-Injection Strategy

We injected a memory-leak fault by declaring a 32KB buffer of memory within the Interceptor, and then slowly exhausting the buffer according to a Weibull probability distribution (commonly used in software reliability and fault-prediction). The memory leak at a server replica was activated when the server received its first client request. At every subsequent 150ms intervals after the onset of the fault, we exhausted chunks of memory according to a Weibull distribution with a scale parameter of 64, and a shape parameter of 2.0. This setting led to approximately one server failure for every 250 client invocations. We used this approach to inject a memory leak as opposed to limiting the maximum heap size or the resident set size (rss) in Linux because Linux uses optimistic memory allocation whereby memory is allocated without first checking if the total required amount of memory is available at the time of the request. This means that an application might exhaust its heap size without raising a segmentation violation. The buffer-based approach provided us with a deterministic fault model which we could then use to test the effectiveness of our proactive recovery schemes, in a reproducible manner.

5.2. Results

For both the reactive and proactive schemes, we measured the following parameters (see Table 1):

- Percentage increase in client-server round-trip times over the reactive schemes;
- Percentage of failures exposed to the client application per server-side failure;
- Time needed by the client to failover to a new server replica.

In our proactive schemes, we also measured the effectiveness of failing over clients at different thresholds.

Recovery Strategy	Increase *in RTT (%)*	*Client Failures (%)*	*Failover Time*	
			(ms)	*change (%)*
Reactive Without Cache	baseline	100%	10.177	baseline
Reactive With Cache	0%	146%	10.461	+2.8%
NEEDS_ ADDRESSING Mode	8%	25%	9.396	-7.7%
LOCATION_ FORWARD	90%	0%	8.803	-13.5%
MEAD Message	3%	0%	2.661	-73.9%

Table 1. Overhead and fail-over times.

5.2.1. Number of client-side failures: In the reactive scheme that did not cache server references, 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 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 reactive scheme that used cached server references 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 TRANSIENT exceptions that occurred when the client accessed a stale replica reference within the cache. Stale cache references occur when we refreshed the cache before a faulty replica has had a chance to restart and register itself with the CORBA Naming Service, thereby leaving its old invalid reference in the cache. This problem can be avoided by staggering the cache-refresh process over time, instead of refreshing all 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 from the Spread group at the point when the previous replica died, but before the group-membership message indicating the replica's crash had been received by all the replicas in the group. At this point, there is no agreed-upon primary replica to service the client request; therefore, the blocking `read` at the client timed out and the client catches a COMM_FAILURE exception.

For the proactive schemes in which there was enough advance 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!

5.2.2. Overhead: We measured the overhead in terms of the percentage increase in client-server round-trip times (RTT). We defined round-trip time as the amount of time that elapsed from the time the client application sent the request to the time it received a reply from the server. The overhead in the reactive schemes, which averaged 0.75ms, served as our baseline reference.

The scheme which used GIOP 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 and fabricate the appropriate GIOP messages needed to forward requests to the next available replica. The NEEDS_ADDRESSING_MODE scheme's overhead was only 8% higher than the baseline since 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 dependability framework, in the event of a failure, typically range between 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 supports more objects, the overhead of the GIOP LOCATION_FORWARD scheme will increase significantly above the rest since it maintains an IOR entry for each object instantiated.

5.2.3. Average 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 3). 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 1.8ms to register at the client. The client then incurs a spike of about 8.4ms to resolve the next server replica's reference (resulting in an overall failover time of 10.2ms).

In the case where we cache server references, we experience about one TRANSIENT exception for every two COMM_FAILURE exceptions. The COMM_FAILURE exception takes about 1.1ms, and the time needed to fail-over to the next cached replica reference and receive a normal response is 7.9ms. However, when the client accesses a stale cache reference, the client experiences a TRANSIENT failure in addition to the COMM_FAILURE exception. The TRANSIENT failure takes about 2.4ms since it includes the time to redirect entries to the new replica, as well as the time to process the the actual TRANSIENT exception. Finally, the client also experiences a spike of about 9.7ms, which is the time taken to resolve all three replica references and receive a correct response. The average fail-over time for this scheme is about 10.5ms, *i.e.*, ((1.1+7.9)*2/3 + (1.1+2.4+9.7)/3) ms. (See Figure 3).

For the proactive scheme that used LOCATION_FORWARD messages, the average fail-over time was 8.8ms (13.5% below the reactive scheme with no cache) because when the client ORB receives the LOCATION_FORWARD, it has to resend the request to the

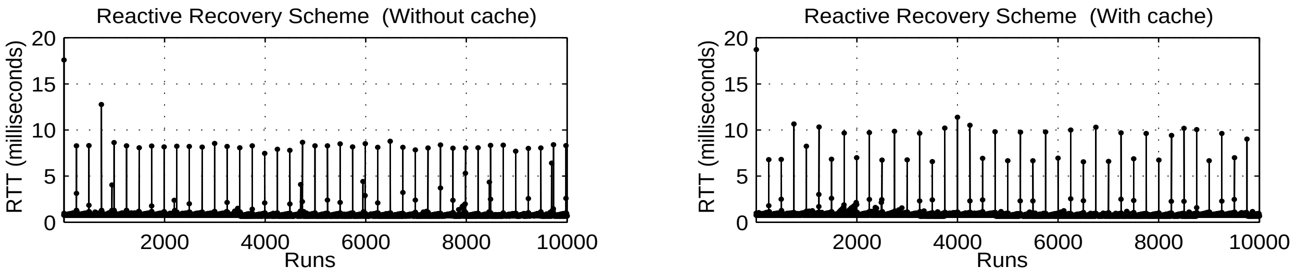


Figure 3. Reactive recovery schemes.

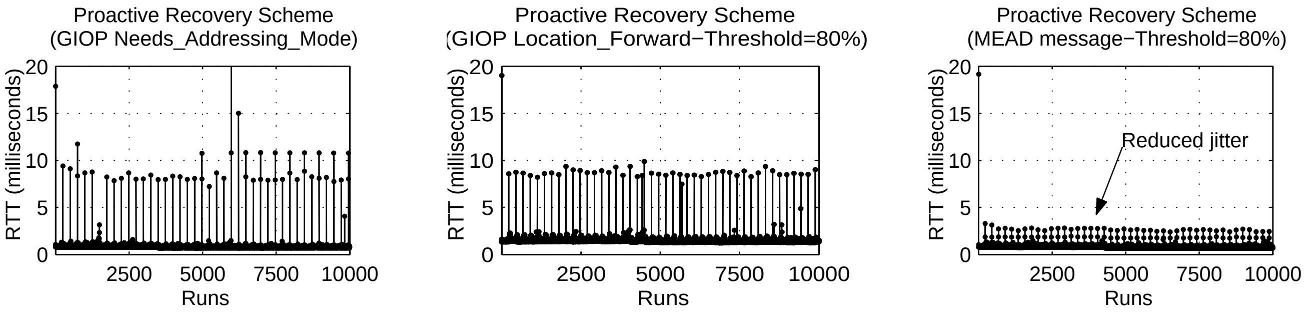


Figure 4. Proactive recovery schemes.

next server replica. For the scheme using MEAD messages, the average fail-over time was about 2.7 ms (73.9% below the reactive scheme with no cache), since 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 average fail-over time is about 9.4ms (7.7% below the reactive scheme with no cache), which is the time taken to contact the Spread group, redirect the client connection and retransmit the request to the new server.

5.2.4. 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 set too low, the overhead in the system increases due to unnecessarily migrating clients. For example, the group communication bandwidth between the servers is about 6,000 bytes/sec at an 80% threshold, but this 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 rejuvenation thresholds and more bandwidth is used up in reaching group consensus (See Figure 5). The best performance is achieved by delaying proactive recovery so that the proactive dependability framework has just enough time to redirect clients away from the faulty server replica to a non-faulty server replica in the system.

5.2.5. 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-σ. These outliers occurred between 1-2.5% of the time. In the fault-free run, the highest spike we observed was 2.3ms. (These spikes might be due to file system journaling done by the operating system.) We also observed one large spike of about 30ms that occurred 0.01% of the time in the GIOP proactive recovery schemes. This spike occurred when we set the rejuvenation threshold below 80%. We suspect the spike happens when a client sends a request to a newly restarted server that is updating its group membership information. The highest spike we observed with the MEAD proactive messages was 6.9 ms at the 20% rejuvenation threshold.

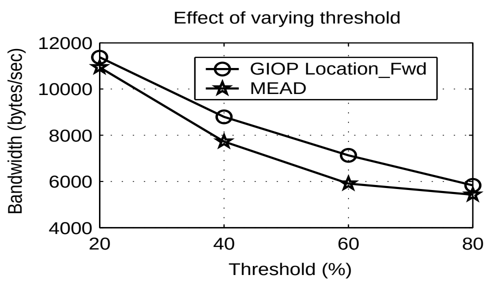


Figure 5. Varying thresholds.

6. Conclusion

In this paper, we describe the development of a transparent proactive 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 73.9% reduction in average fail-over times over a traditional reactive recovery scheme. We incur a reasonable overhead of about 3% over the normal client/server round-trip times. When we use GIOP messaging schemes, the fail-over time is about 13.5% lower, but we need to maintain additional state at the server and incur an round-trip time overhead of about 90%. If we attempt to suppress exceptions at the client whenever we have insufficient time to initiate proactive recovery, our fail-over time is about 7.7% lower than in the reactive case. However, we also observe a 25% client-side failure rate. Since it is not possible to predict every system failure, (or have enough time to recover from a failure, even if fault-prediction is possible), proactive recovery schemes should be used to complement, but not replace, reactive schemes.

We also show that if we trigger proactive recovery too early, 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.

As part of our future work, we plan to extend our proactive dependability framework to include more sophisticated failure prediction. We also plan to integrate adaptive thresholds into our framework rather than relying on preset thresholds supplied by the user.

References

[1] Y. Amir, C. Danilov, and J. Stanton. 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] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *Computer Performance and Dependability Symposium, IPDS '98*, pages 4–12, September 1998.

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

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

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

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

[7] T.-T. Y. Lin and D. P. Siewiorek. Error log analysis: Statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, October 1990.

[8] C. Marchetti, L. Verde, and R. Baldoni. CORBA request portable interceptors: A performance analysis. In *3rd International Symposium on Distributed Objects and Applications*, pages 208–217, September 2001.

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

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

[11] Object Management Group. The CORBA/IIOP Specifi cation Version 3.0.2. OMG Technical Committee Document formal/2002-12-02, December 2002.

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

[13] G. Rubino. Predicting dependability properties online. In *Symposium on Reliable Distributed Systems*, pages 22–24, October 1997.

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

[15] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available Internet services using connection migration. In *22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

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

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

[18] D. Wong and T. J. Lim. Soft handoffs in CDMA mobile systems. In *Personal Communications, IEEE [see also IEEE Wireless Communications], Vol.4, Iss.6*, pages 6–17, December 1997.

[19] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system fi eld failure data analysis. In *IEEE Pacific Rim International Symposium on Dependable Computing*, December 1999.

[20] B. Yujuan, S. Xiaobai, and K.S. Trivedi. Adaptive software rejuvenation: Degradation model and rejuvenation scheme. In *International Conference on Dependable Systems and Networks*, pages 241–248, June 2003.