FLARe: a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems

Jaiganesh Balasubramanian[†], Chenyang Lu[‡], Aniruddha Gokhale[†], Christopher Gill[‡], and Douglas C. Schmidt[†]
[†]Department of Electrical Engineering and Computer Science, Vanderbilt University, USA
[‡]Department of Computer Science and Engineering, Washington University in St. Louis, USA

Abstract

A key challenge for middleware that supports both real-time and fault-tolerance properties is to maintain both system availability and timeliness, even in the presence of processor and/or process failures and fluctuations in system load. This paper presents FLARe, a new fault-tolerant, lightweight, adaptive, real-time middleware. FLARe provides a novel fail-over strategy that is (1) load-aware, i.e., *selects fail-over targets based on current CPU utilizations to prevent post recovery overload and maintain real-time performance, (2)* proactive, i.e., *provides clients with failover targets before a failure occurs to enable faster, localized, and predictable failure recovery; and (3)* adaptive, i.e., *dynamically adjusts the failover targets in response to failures and load fluctuations. Empirical results on a Linux cluster demonstrate that FLARe's adaptive approach outperforms static fail-over approaches by adapting efficiently and effectively to failures and load changes.*

1 Introduction

Emerging trends and challenges. Many distributed real-time and embedded (DRE) systems, such as command/-control and industrial process automation systems, consist of soft real-time applications that must maintain desired performance even when hardware and software failures occur. For example, the target tracking subsystem [3] of an avionics mission computer should continue to process external sensor readings and provide timely responses even if processors or processes fail.

ACTIVE and PASSIVE replication are two approaches for building fault-tolerant distributed systems [9, 12, 8, 22]. In ACTIVE replication, client requests are multicast and executed at all replicas to maintain strong consistency and provide fast failure recovery. ACTIVE replication, however, can incur excessive overhead for DRE systems composed of (1) stateless applications that do not require strong state consistency across multiple replicas and (2) soft real-time applications that can tolerate occasional deadline misses.

For such DRE systems, PASSIVE replication may be preferred, where only one replica—called the primary—handles all client requests, and backup replicas receive state updates (in the case of stateful applications) from the primary. If the primary fails, a failover occurs and one of the backups becomes the new primary. Prior work has focused on providing fault-tolerance for real-time applications using PASSIVE replication schemes. For example, MEAD [12] can determine the possibility of a primary failure and reduce fault recovery times proactively by redirecting clients to non-faulty replicas prior to the failure. ARMADA [1] uses a real-time primary-backup replication scheme [22] to provide fault-tolerance capabilities and improve response times for real-time applications that can tolerate minor state inconsistencies between the primary and the replicas.

Despite promising prior work on passive replication for DRE systems, the following challenges remain open:

- **Where to failover when a failure is detected**. Prior work [11, 17] on failover target selection algorithms has focused on where to redirect clients after the failure of a primary replica, but because it does not consider the effect of processor load on response times it cannot ensure real-time performance for clients.
- **How to proactively update the clients with the failover target decisions**. Prior work [2, 5] on fault-tolerant middleware has relied on portable interceptors [21] and redirection agents to redirect clients to alternate servers following a primary failure. These *reactive* approaches add significant and variable fail-over latency, and instead the clients should failover to an appropriate target within a short and predictably bounded interval.
- **How to manage overloads after a failover**. Prior work [14, 1] has focused on managing overloads that may result from reconfigurations and failover. Whereas these approaches focus on degrading application quality of service to handle overloads, when possible the middleware should handle overloads adaptively and transparently, and avoid degrading quality of service unnecessarily.

In summary, the effective use of PASSIVE replication in DRE systems requires (1) timely failover to an appropriate failover target and (2) ensuring that system utilization at all the processors remain below its schedulable utilization bound after failover, so that all real-time applications meet their deadlines [10]. To use PASSIVE replication for applications in DRE systems, therefore, middleware is needed that can provide adaptive (1) fault-tolerance by dynamically deciding and updating clients on appropriate failover targets to ensure predictable and timely failure recovery, and (2) resource management by preventing a class of overloads to ensure real-time performance after a failure recovery.

Solution approach → Resource-aware adaptive fault tolerance. To address unresolved challenges with prior work, we developed *F*ault-tolerant *L*ightweight *A*daptive *Re*al-time (FLARe), which is middleware for DRE systems that provides timely failover and real-time performance after processor failures via the following techniques:

- *Adaptive, load-aware server failover*, where up-to-date utilization estimates are used to choose failover targets that maintain timely response to client's requests and avoid system overload after a failover.
- *Proactive, timely client redirection*, where client-side middleware is updated proactively with suitable failover targets so it can make local request redirection decisions for clients when a server fails.
- *Lightweight middleware architecture* based on interceptors [21] that provide client-side enhancements, such as proactive redirection for fault tolerance, that extend the middleware transparently to applications.
- *Lightweight overload management* that performs client-redirection using a load redistribution algorithm that handles a class of after-recovery overloads in the case of multiple (possibly simultaneous) failures.

This paper describes the design of FLARe and evaluates it empirically on the ISISlab testbed (`www.dre.vanderbilt.edu/ISISlab`) consisting of 14 nodes running Fedora Core 4 Linux in the real-time scheduling class. Our results demonstrate the effectiveness of FLARe's proactive and load-aware failover strategy to minimize client response times and to re-balance system resource utilization after processor failures.

2 System and Fault Model

FLARe supports DRE systems, where clients periodically invoke real-time services provided by servers through remote operation requests. The current implementation of FLARe assumes the services are stateless. Many real-time services (*e.g.*, sensor data acquisition and processing) are inherently data-driven; *i.e.*, their execution is driven by the current sensor readings and hence are stateless in nature.

For example, in the target tracking system of a shipboard computing application, the coordinate calculator that calculates real-world coordinates of surveillance images can be designed to process each image independently, to avoid needing to maintain state between its separate invocations. Such systems need to provide real-time performance to clients, however, even in the presence of failures and load fluctuations. The goal of FLARe is thus to manage soft real-time performance of such systems.

The clients and servers (*e.g.*, the image forwarding base station and coordinate calculator services) are implemented Real-time CORBA service objects using the TAO [16] Object Request Broker (ORB). The services on each node are scheduled using Rate Monotonic Scheduling [10] (FLARe's architecture can also support other middleware, such as Distributed Real-time Java, and other scheduling policies, such as Maximum Urgency First (MUF) [18]). A processor hosts multiple processes and each process hosts multiple objects or services.

The processors and the processes hosted by the processors are assumed to be *fail-stop* [15], where (1) each processor or a process halts in response to a failure rather than produce erroneous results and (2) a processor's or process's halted state can be detected by a failure detector. These types of faults may occur due to aging or acute damage. Considering unpredictable behavior of processes or processors is beyond the scope of this paper.

We assume that networks provide bounded communication latencies and do not fail. This assumption is reasonable for many DRE systems, such as avionics mission computing and shipboard computing environments, where nodes are connected by highly redundant high-speed networks. Relaxing this assumption through integration of our middleware with network level fault tolerance techniques is another area of future work. The services (*e.g.*, the coordinate calculator service) are replicated using the PASSIVE replication scheme and deployed across multiple nodes distributed across a local area network (LAN). Each node can host multiple *primary* services or multiple *backup* services; the *primary* services are accessed by several clients.

3 Design of FLARe

This section describes the design of the FLARe middleware. Our research objective is to mask failures from client applications, provide fast failure recovery, and maintain desired real-time performance after recovery. FLARe achieves transparent and fast failover through an adaptive redirection agent that intercepts a client-side failure exception and redirects the client to an appropriate failover target, which was determined before the failure occurred based on measured CPU utilization. FLARe maintains desired real-time performance by enforcing the *schedulable utilization bound* of the processors when selecting the failover target.[1]

3.1 Overview of FLARe Architecture

The key components of FLARe are shown in Figure 1 and described below in the context of the resource-aware fault-tolerance challenges they address.

Middleware replication manager. FLARe's middleware

[1]According to well-established real-time scheduling theories, all periodic services executing on a processor will meet their deadlines if their total CPU utilization remain below the schedulable utilization bound for the real-time scheduling policy. For example, the schedulable utilization bound of the well known Rate Monotonic scheduling policy is $n(2^{1/n} - 1)$, where n is the number of periodic services on the processor [10].

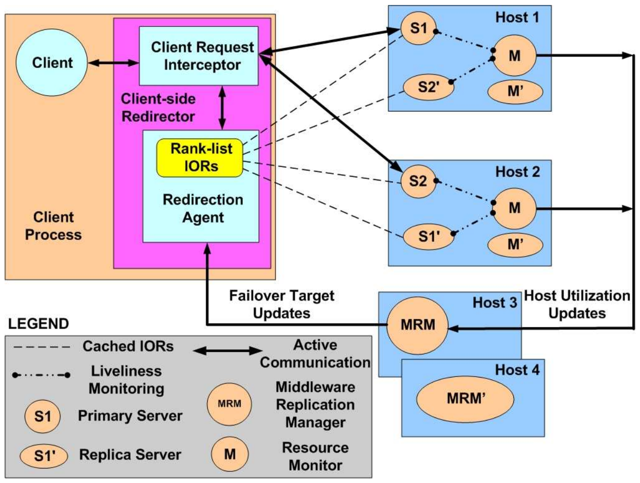


Figure 1: FLARe Middleware Architecture

replication manager provides interfaces for registering and managing information about the server objects and their backup replicas. For each server object, FLARe's middleware replication manager keeps track of the processors hosting replicas, and collects information about their CPU utilization and failures. The middleware replication manager determines a list of failover targets ordered by the predicted CPU utilization of the processors if a failover occurs (the processor with the lowest predicted CPU utilization is the first in the list). As the system operating conditions (e.g., processor loads) change, the middleware replication manager also may update the failover targets. The middleware replication manager sends the updated list of failover targets to the redirection agents through remote operation requests whenever the list is changed. Section 3.2 describes our adaptive failover target selection algorithm in detail.

Client request interceptor. Client request interceptors [21] transparently modify the behavior of the CORBA object method calls invoked by an application. A client-side COMM_FAILURE exception is raised after a connection timeout because of a server failure. The client request interceptor then modifies the exception handling behavior that is triggered: instead of propagating that exception to the client application, the client request interceptor transparently redirects the client invocation to an appropriate failover target object provided by the redirection agent. Section 3.3 describes how FLARe's client request interceptor works with the middleware replication manager to redirect clients to appropriate failover targets.

Redirection agent. Client request interceptors are not themselves CORBA objects, and thus cannot be invoked through remote interfaces. To allow FLARe's middleware replication manager to send the list of failover targets to the client request interceptor, a redirection agent runs in a separate thread as a separate CORBA object within the client process, and the middleware replication manager updates the redirection agent with the current list of failover targets. Upon catching an exception, the client request interceptor contacts the redirection agent to obtain the failover object address, and redirects the client to that server object. Section 3.3 describes how clients are proactively updated as needed with new list of failover target decisions so that failovers can be timely, and predictable.

Resource monitor. FLARe runs a resource monitor on each processor to track its CPU utilization and the liveness of processes hosted by the processor. FLARe's replication manager makes failover target selection and load redistribution decisions based on the updated CPU utilization and liveness information provided by the resource monitors. Sections 3.4 and 3.5 describe how FLARe manages system overloads with only small client perturbance and maintains desired real-time performance after recovery.

3.2 Failover Target Selection

As described in Section 3.1, FLARe's middleware replication manager collects updates from the resource monitors about the CPU utilizations and liveness of processors. We now describe how using these measurements, FLARe's middleware replication manager selects per-object failover targets using adaptive failover target selection Algorithm 1. The algorithm takes the list of processors, the set of objects

Algorithm 1 Rank Per-object Failover Targets

```
1: $o_i$ : number of objects in processor $i$
2: $R_j$ : list of processors hosting object $j$'s replicas
3: $cu_i$ : current utilization of processor $i$
4: $eu_i$ : expected utilization of processor $i$ after failovers
5: $l_j$ : CPU utilization of object j
6: for every processor $i$ do
7:     $eu_i = cu_i$
8:     for every object $j$ in $o_i$ do
9:         sort $R_j$ in increasing order of expected CPU utilization
10:        $eu_i$ += $l_j$, where processor $i$ is the head of the sorted list $R_j$
11:    end for
12: end for
```

and the current CPU utilization in each of those processors as input, and determines a ranked list of failover targets for each of those objects. The algorithm recognizes that multiple and simultaneous failures of the same type of service (*e.g.*, failure of *primary* and *backup* replicas of the same service) needs to be handled, and hence determines an ordered *list* as opposed to a single failover target.

Since the middleware replication manager manages information about the objects and their replica hosts, intuitively one can order the potential failover targets of an object in the increasing order of the current CPU utilization of their host processors. However, the algorithm also recognizes that in the event of a processor failure, invocations on all the objects hosted on the processor will need to failover to other hosts and multiple objects could have failover targets hosted in the same host.

To avoid overloading a processor with multiple failovers, the algorithm spreads the failover targets of objects located on the same processor onto multiple hosts. The middleware replication manager also sorts the failover targets of an object based on the *expected CPU utilization* instead of the current CPU utilization of their host processors. The *expected CPU utilization* accounts for the potential load increase due to failover decisions of other objects in the same processor (lines 9 and 10 of Algorithm 1).

3.3 Client Redirection

FLARe provides fast failover with predictable latencies by proactively sending updated failover targets from the middleware replication manager to a *client-side redirector* that handles failures transparently to the client objects. The client-side redirector comprises a *client request interceptor* for each client object and a *redirection agent* in each client process. Whenever the updated list of failover targets changes FLARe's replication manager *proactively* sends the list the redirection agent. In FLARe, the client request interceptor transparently handles CORBA COMM_FAILURE exceptions that are raised in response to server or service failures.[2]

After catching a failure exception, rather than propagating that exception to the client application, the client request interceptor contacts the redirection agent to obtain the failover object address, and redirects the client to that server object. In the case of simultaneous failures of the *primary* as well as *backup* replicas of the same service, the references held by the redirection agent could be stale. However, since the redirection agent maintains a *list* of failover targets, the client will be redirected to a server object in a timely fashion unless all the replicas of a service failed.

3.4 Resource Monitoring

FLARe runs a *resource monitor* on each processor to track the CPU utilization and liveness of the processes hosted by the processor. On Linux platforms, the resource monitor uses the `/proc/stat` file to estimate the CPU utilization in each sampling period. The `/proc/stat` file records the number of "jiffies" (a default duration of 10ms in Linux) when the CPU is in user, nice, system and idle modes. At the end of each sampling period, the resource monitor reads the counters and estimates the CPU utilization as the fraction of time when the CPU is not idle.

To detect the failure of a process quickly, each application process on a processor opens up a passive POSIX local socket (also known as a UNIX domain socket), and registers the port number with the resource monitor. The resource monitor connects to and performs a blocking read on the socket. If an application process crashes, the socket and the opened port will be invalidated. The resource monitor then receives an invalid read error on the socket, which indicates the failure of the process.

The resource monitor periodically updates FLARe's middleware replication manager with the processor utilization information. To improve the FLARe middleware's responsiveness to sudden workload changes and failures, the resource monitor also generates event-driven updates to the middleware replication manager, when utilization levels increase beyond a certain threshold or when a process fails. This design allows the middleware replication manager to recompute the failover information for the affected server objects, in response to dynamic changes in system workload and failures. FLARe's middleware replication manager also proactively notifies the redirection agents of any such changes so client requests will be redirected to appropriate failover objects in an adaptive manner.

3.5 Overload Management

Section 3.2 described a failover target selection algorithm that determines appropriate failover targets for server objects. While the failover target selection algorithm described in Section 3.2 helps avoid CPU overload by selecting the host with the lowest CPU utilization, failover may still cause the failover target processor to exceed its schedulable utilization bound. FLARe's middleware replication manager employs an algorithm to manage the overload caused by failover. Prior work [7] on overload management has considered moving objects to any of the less loaded processors. Consequently, clients are redirected to the object's new location and helps reduce CPU utilization in overloaded processors. This approach, however, increases overload management latency as an object's replica might not be operating in some of the chosen processors and may require bootstrapping. The goal of our overload management algorithm is to manage overloads in a more efficient manner.

We solve this problem by deactivating *primary* objects on an overloaded processor and activating their *backup* replicas on a lightly loaded processor. Clients of the *primary* replicas are automatically redirected to the chosen *backup* replicas. We refer to this load redistribution mechanism *lightweight migration*, as we migrate *load* of as opposed to *objects*. Hence our approach is much more efficient than physically moving the object itself to a lightly loaded processor.

FLARe determines the *primary* objects that need to be migrated and their target hosts using the overload management algorithm. Given an overloaded processor, *i.e.*, whose

[2] CORBA relies on the underlying network transport protocol's (*e.g.*, TCP) connection timeout mechanisms to detect server failures. Since TAO supports client-server communications using many different protocols, our failure detection mechanism can be significantly improved with protocols like SCTP [19].

CPU utilization exceeds its schedulable utilization bound, the algorithm considers the *primary* objects on the processor in the decreasing order of CPU utilization, and attempts to migrate the load generated by those objects to the least-loaded processor hosting their *backup* replicas. The algorithm continues to attempt migrations until (1) the overloaded processor is no longer overloaded (the algorithm solves the overload) or (2) all the *primary* objects in the overloaded processor have been considered for migration.

Similar to the failover target selection algorithm, FLARe's overload management algorithm also uses the *expected CPU utilization* to spread the load of multiple objects on an overloaded processor to different hosts. The expected CPU utilization accounts for the load change due to the migration decisions on other objects on the same processor. After the new reconfigurations are identified, redirection agents are updated to redirect existing clients from the current *primary* service to the newly activated *primary* service at the start of the next remote invocation. Clients, and thereby their loads, are thus redirected to new targets with minimal disturbance.

3.6 Implementation of FLARe

FLARe has been implemented on the TAO Real-time CORBA middleware [16]. It is implemented in 3400 lines of C++ source code (excluding the code in TAO). The current implementation of FLARe is based on Linux 2.6. It can easily be ported to other platforms due to the portability of TAO and the underlying ACE library [16]. FLARe uses semi-active replication [13] to provide fault-tolerance capabilities to its middleware replication manager as well as to the per-processor resource monitor. Since FLARe's middleware replication manager and its replicas are located on a set of dedicated processors they will not experience overloads after failures. FLARe is available in open-source format from `www.dre.vanderbilt.edu`.

4 Empirical Evaluation

This section evaluates FLARe's ability to handle various common fault and overload scenarios that can arise during DRE system operation.

4.1 Experiment Configurations

The experiments were conducted at ISISlab (`www.dre.vanderbilt.edu/ISISlab`) on a testbed of 14 blades. Each blade has two 2.8 GHz CPUs, 1GB memory, a 40 GB disk, and runs the Fedora Core 4 Linux distribution. Our experiments used one CPU per blade and the blades were connected via a CISCO 3750G switch over a 1 Gbps LAN.

As shown in Figure 2, 12 blades at ISISlab ran Real-time CORBA client/server applications developed using FLARe. FLARe's middleware replication manager and its backup replicas ran in the other two blades.

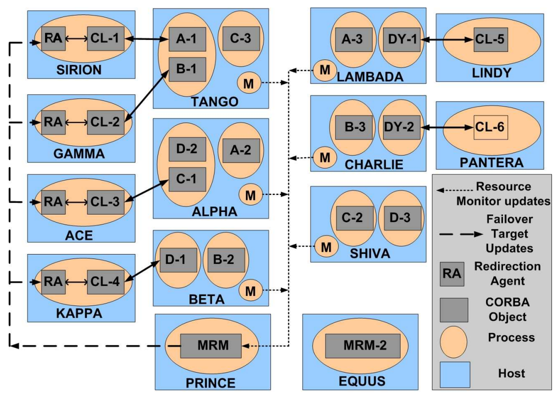


Figure 2: Experiment Setup

The clients in these experiments used threads running in the Linux real-time scheduling class to invoke operations on server objects at periodic intervals. For the experiments conducted for this paper, client applications invoked operations on server objects using one of the following rates: 10 HZ, 5 HZ, 2 HZ, or 1 HZ. As shown in Figure 2, four clients (CL-1, CL-2, CL-3, and CL-4) invoke operations on four different types of server objects (A-1, B-1, C-1, and D-1). To evaluate FLARe in the presence of resource contention created by external disturbances, such as dynamic task arrivals, we introduced dynamic requests using two additional clients, CL-5, and CL-6, to invoke operations on two server objects, DY-1 and DY-2, respectively.

The server objects also have backups deployed on other processors. For example, A-2, and A-3 are replicas of the server object A-1 deployed on processors ALPHA and LAMBADA, respectively. Since the clients invoke operations at four different rates, the higher each server object's invocation rate, the higher the priority at which it was run, per RMS.

We compared FLARe's proactive load-aware client failover strategy (Section 3) with the following two client failover strategies:

- *Static* client failover strategy, where the client is initialized with a *static* list of IORs, which are not updated based on the replicas' readiness or effectiveness to handle client invocations after a failover.
- *Reactive load-aware* client failover strategy, where the client-side middleware invokes a remote operation on the middleware replication manager *after* each failure to obtain the suitable failover target address. The replication manager uses the replica selection algorithm described in Section 3. The reactive load-aware strategy is thus an *on-demand* alternative to FLARe's *proactive* target update feature, which we evaluate for purposes of comparison.

As is described in Section 3, the strategy adopted by FLARe is both proactive *and* load-aware, where the middleware replication manager proactively pushes failover target updates to clients.

4.2 Load-aware Failover Decisions

Rationale. When process or processor failures occur in a system, FLARe fails over the clients' server object references to backup replicas hosted in other available processes and/or processors. This experiment evaluates how end-to-end response times and processor utilizations are affected due to failover decisions made by the different failover strategies.

Methodology. The reactive load-aware failover strategy is similar to our proactive load-aware failover strategy, except that in the case of the reactive load-aware strategy there is an additional delay for a remote call to the middleware replication manager to locate the failover server object's address. The failover object that is chosen is the same as the one chosen by FLARe since the supplier of that information (the middleware replication manager) is the same in both the strategies. This experiment therefore compares the proactive load-aware strategy and the static strategy to evaluate the effects of load-awareness.

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
CL-1	A-1	10	40%
CL-2	B-1	5	30%
CL-3	C-1	2	20%
CL-4	D-1	1	10%
CL-5	DY-1	5	50%
CL-6	DY-2	10	50%

Table 1: Experiment setup

Experiment setup. As shown in Figure 2, in this experiment four different clients, CL-1, CL-2, CL-3, and CL-4, invoke operations on server objects with configurations described in Table 1. This table also describes the configurations for the dynamic clients CL-5 and CL-6. The experiment ran for 300 seconds, and as described above all the clients made their respective invocations on different server objects unless a failure happened to cause clients to continue their invocations on common backup server objects.

Failure scenario. To evaluate the performance of the different failover strategies, we emulated a failure 150 seconds after the experiment started. We used a simple fault injection mechanism, where when clients CL-1 or CL-2 make invocations on server objects A-1 or B-1 respectively, the server object calls the *exit (1)* command, crashing the process hosting server objects A-1 and B-1 on processor TANGO. The clients receive COMM_FAILURE exceptions, and then make continued invocations on replicas chosen by the failover strategy.

Failover strategy configurations. The static failover strategy makes failover decisions at deployment time, as follows: if A-1 fails, contact A-3 followed by A-2; and if B-1 fails, contact B-3 followed by B-2. With FLARe's proactive load-aware failover strategy, those failover decisions are updated dynamically when/if failures occur, as processor utilization levels and sets of live processes change.

Metrics. We measured the per-invocation roundtrip response time a client experienced both in the presence and absence of failures. We also measured the following fault-recovery metrics: (1) FAULT_DETECTION_DELAY is the time taken for the client to receive a COMM_FAILURE exception after the server object failure, and (2) FAILOVER_DELAY is the time taken for the client to find the next replica address to contact after the COMM_FAILURE exception is received in the case of a failure. We also measured processor utilizations throughout the experiment.

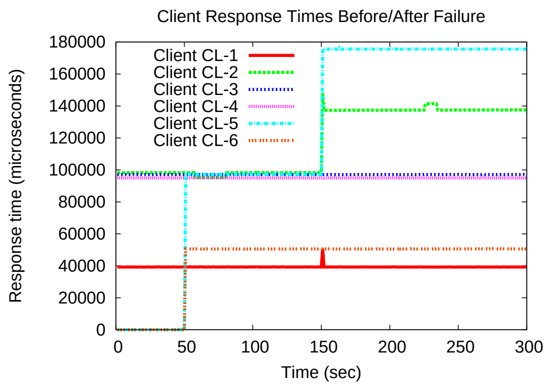
Analysis of results. Figure 3a shows the end-to-end response times perceived by clients CL-1 and CL-2 when they are configured to use the static strategy. At 150 seconds, when A-1 and B-1 fail, CL-1 and CL-2 receive a COMM_FAILURE exception and make a failover to the statically-chosen failover targets A-3 and B-3 respectively. As shown in Figure 3a, at 150 seconds, the end-to-end response time perceived by CL-1 increases by 10.2 milliseconds, which is the combined FAULT_DETECTION_DELAY and FAILOVER_DELAY.

After failing over to target B-3, the end-to-end response time perceived by CL-2 increases by ∼40% and the processor utilization at CHARLIE increases from 50% to 80% because B-3 shares the processor resources of CHARLIE with DY-2, which is accessed by CL-6. As described in Table 1, CL-6 has a higher invocation rate than CL-2, so their respective servers (DY-2 and B-3) operate at different priorities (DY-2 has a higher priority). Since CL-2 is the lower priority client, and the processor utilization is high after the failover, its end-to-end response times increases.

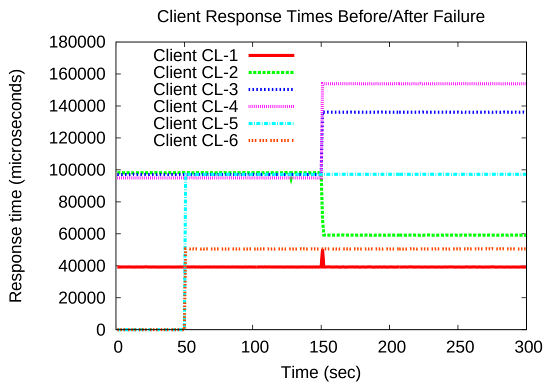
On the other hand, after the failover CL-1 invokes remote operations on A-3, which is hosted along with DY-1 at LAMBADA. As shown in Figure 3c, the utilization of LAMBADA grew from 50% to 90%. This utilization increase affects CL-5, which accesses DY-1. This is because CL-1 accesses A-3 at higher priority because of its higher invocation rate. Consequently, the end-to-end response times perceived by CL-5 increases as shown in Figure 3a.

In summary, because of load-agnostic failovers : (1) failing over clients can affect the performance of the processor's previously-active clients (*e.g.*, the case of CL-1 affecting CL-5), and (2) already active clients in the processor can affect the failing over clients (*e.g.*, the case of CL-6 affecting CL-2).

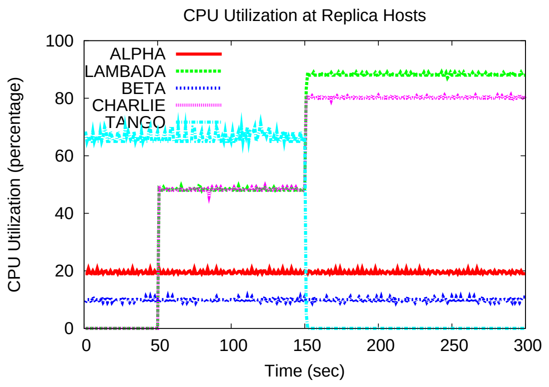
We repeated the same experiment with FLARe's proactive load-aware failover strategy. Figure 3b shows the end-to-end response times perceived by clients CL-1, CL-2, CL-5, and CL-6 and figure 3d shows the utilizations of all their server's respective processors. After system bootstrapping, the middleware replication manager monitors the CPU uti-



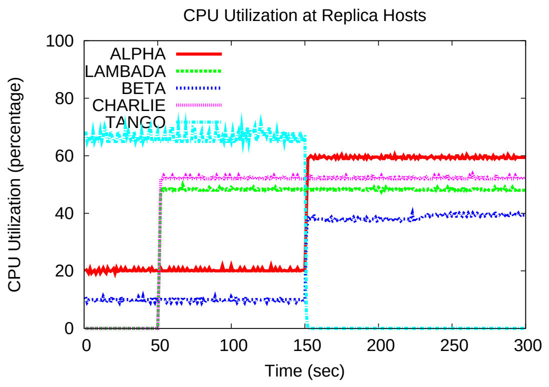
(a) End-to-end response times with static strategy



(b) End-to-end response times with proactive load-aware strategy



(c) Utilization with static strategy



(d) Utilization with proactive load-aware strategy

Figure 3: End-to-end response times and utilizations with different failover strategies

lizations at the hosts of the *backup* replicas of services A-1 and B-1, which are accessed by clients CL-1 and CL-2.

At 0 seconds, CL-3 and CL-4 make invocations on servers C-1 and D-1 respectively. As described in Table 1, the utilizations at their respective processors ALPHA and BETA increase by 20% and 10% respectively. At 50 seconds, the utilizations of LAMBADA and CHARLIE increases by 50% because of the activation of the servers DY-1 and DY-2, respectively. Since the utilizations of these processors are higher than the utilizations of ALPHA and BETA, the middleware replication manager chooses the failover targets for A-1 and B-1 as A-2 (hosted on ALPHA) and B-2 (hosted on BETA), respectively.

At 150 seconds, A-1 and B-1 fail. As shown in Figure 3b, the end-to-end response times perceived by clients CL-1 and CL-2 increases by 10.2 milliseconds at 150 seconds because of the FAILOVER_DELAY. The end-to-end response time perceived by CL-2 decreases by about 40% after the failover to B-2 on host BETA, which also hosts D-1 at a lower priority than B-2. The sharp decrease in the end-to-end response time perceived by CL-2 is caused by the low processor utilization of BETA, which does not increase by more than 40% throughout the experiment. Moreover, B-2 serves requests at the highest priority on BETA. Using the proactive load-aware strategy, the end-to-end response times perceived by CL-2 after the failover are 3 times less than those perceived by CL-2 using the static strategy. This result demonstrates the significant positive impact of proactive load-aware failover on the real-time performance of DRE systems.

Moreover, the end-to-end response times perceived by CL-5 and CL-6 did not change after the failover of clients CL-1 and CL-2. The behavior is unchanged because the replica selection algorithm did not choose LAMBADA and CHARLIE as the failover target processors, once DY-1 and DY-2 were activated in those processors. This result demonstrates that FLARe proactively updates failover targets when system workload changes dynamically.

Figure 3d shows the utilizations of processors ALPHA and BETA where the failover targets were hosted. After failover the utilizations of these processors are similar to the utilizations of processors LAMBADA and CHARLIE, which host the other replicas of the failed objects A-1 and B-1. This result contrasts the processor utilizations with the static strategy shown in Figure 3c, where the utilizations of the processors hosting the failover targets are 4 times and 8 times higher than the utilizations of the processors hosting the other replicas of A-1 and B-1.

By keeping the utilizations balanced, FLARe's proactive load-aware strategy not only provides timely responses to the failing over clients, but also did not affect the already-active servers (which is significantly better performance than the static strategy). For example, as shown in Figure 3b, the end-to-end response times of CL-3 and CL-4 increases by only 35% and 50% after the failover of CL-1 and CL-2. This result is much better when compared to the 90% increase in end-to-end response times for CL-5 in the static strategy.

4.3 Overload Management

We now evaluate how FLARe manages a class of overloads where multiple *primary* replica failures cause an overload. In this experiment as shown in the Figure 4, four different clients, CL-1, CL-2, CL-3, and CL-4, invoke operations on server objects with configurations described in Table 2. The experiment ran for 300 seconds and similar to the experiment described in Section 4.2, at 150 seconds A-1 and B-1 fail on processors TANGO and ALPHA respectively. With our proactive load-aware failover strategy, the failover decisions are made at runtime; if A-1 fails, contact A-2 followed by A-3 (C-1 is collocated with A-3 and A-2 is deployed in an idle processor; A-2 is the least-loaded failover target). Similarly, if B-1 fails, contact B-2, and then followed by B-3.

Client	Server	Rate (Hz)	CPU Util. %
CL-1	A-1	10	40%
CL-2	B-1	5	50%
CL-3	C-1	2	20%
CL-4	D-1	1	10%

Table 2: Overload Experiment Setup

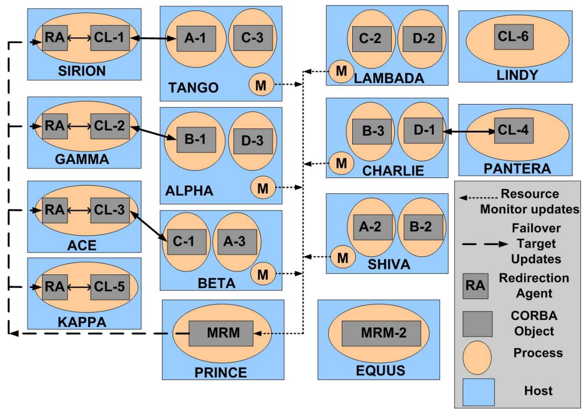


Figure 4: Overload Experiment Setup

Analysis of results. Figure 5a shows the end-to-end response times perceived by clients CL-1 and CL-2 before and after the *primary* replica failure. When FLARe's failover target selection algorithm makes failover target decisions for the *primary* replica B-1, it does not consider that B-1 could fail simultaneously with the *primary* replica A-1. In this experiment, since the *primary* replicas A-1 and B-1 fail together at 150 seconds, both clients failover to the same processor. As shown in Figure 5a, at 150 seconds, the end-to-end response times of client CL-2 increases by ∼40% due to the overloads caused by the simultaneous failure of the *primary* replica A-1. The end-to-end response times of client CL-1 is not affected, however, since it still invokes remote operations on the higher priority server (A-2) after the failover to processor SHIVA.

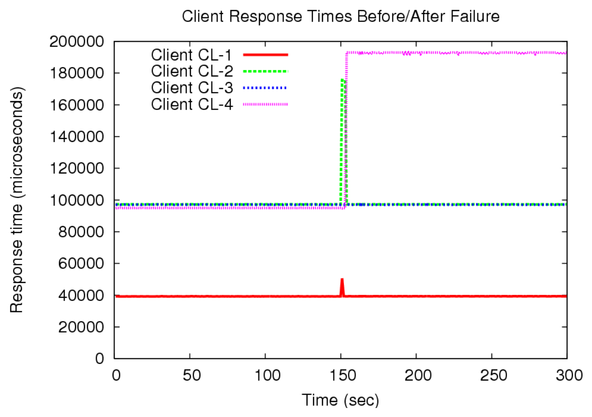
The increase in loads in the processor SHIVA (see Figure 5b) is immediately reported via event-driven updates to the middleware replication manager. The middleware replication manager invoked the overload management algorithm, which immediately detected that the load generated by the replica B-2 can be redistributed to the replica B-3 in the processor CHARLIE. The redirection agents were informed of the decision to redirect clients to the replica B-3 within 500 milliseconds. The next time, when the client CL-2 invoked a remote operation, the redirection agent automatically redirected the clients to the replica B-3, and the end-to-end response times of the client decreased around 153 seconds (within 3 seconds after the overload) as shown in Figure 5a. Thus FLARe can handle overloads in a faster manner and also manage the utilization levels of all the processors as shown in Figure 5b to maintain soft real-time performance for clients.

4.4 Proactive Failover Decisions

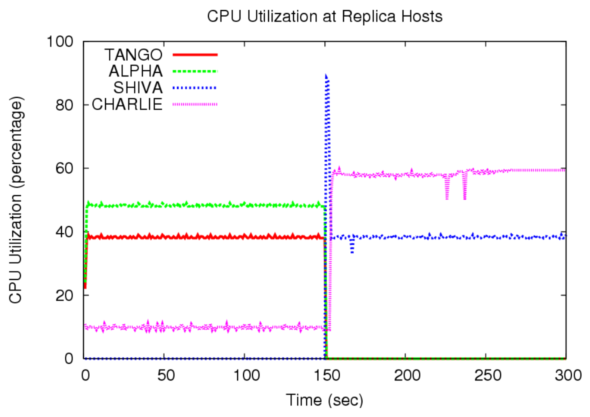
When compared with the proactive load-aware and static failover strategies, the reactive load-aware strategy incurs more time to failover to the next server object. This increase stems from the remote invocation of FLARe's middleware replication manager after receiving the COMM_FAILURE exception from a server object failure. To evaluate the delay empirically, we ran an experiment with client CL-1 invoking operations on server object A-1. No other processes operate in the processor hosting A-1, so that the response time will equal the execution time of the server.

We ran the experiment for 10,000 iterations. A fault is injected to kill the server while executing the 5001st request. The clients then failover to backup server objects A-2 and A-3, which execute the remaining 5,000 requests (including the one experiencing the failure).

Figure 6a shows the different response times perceived by client C-1 in the presence of server object failures. The failover delays for the static and proactive load-aware strategies are similar because both strategies know the failover decision *a priori* and just use the next available address. In the reactive load-aware strategy, however, the decision is not known *a priori*, so FLARe's middleware replication manager is contacted to get the next address to try. This remote invocation increases the response time of the failover request further. When combined with the results shown in Section 4.2, the results in Figure 6a clearly show that the proactive load-aware strategy is better than either the reac-

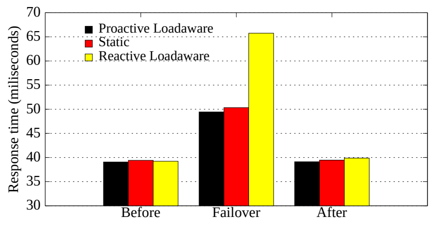


(a) End-to-end response times under overloads

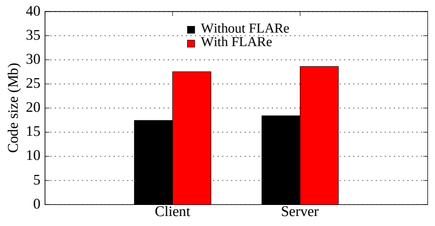


(b) CPU utilizations under overloads

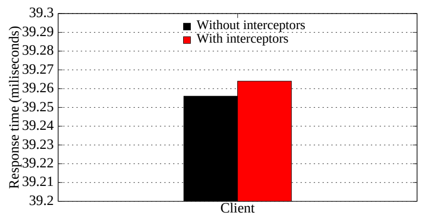
Figure 5: FLARe's overload management performance



(a) Failover delay



(b) Code size overhead



(c) Runtime overhead

Figure 6: Failover delay and overhead measurements

tive load-aware or static failover strategy, and thus is more suitable for use in DRE systems.

4.5 Overhead Measurements

FLARe provides fault tolerance capabilities to DRE systems using a lightweight middleware architecture, as described in Section 3. A DRE system spends the bulk of the time performing its application logic, and comparatively less time detecting and recovering from failures. It is therefore worthwhile to determine what time/space overhead is added by the FLARe middleware to the normal functioning of applications in DRE systems.

Memory footprint and run-time invocation overhead are important time/space metrics for DRE systems since they affect the ability of applications to run in resource-constrained environments. The following capabilities of FLARe affect the memory footprint and runtime performance of applications in DRE systems: forwarding agents are added to handle proactive updates from the middleware replication manager; client request interceptors are added to catch COMM_FAILURE exceptions and transparently redirect requests to suitable failover targets; and resource monitors are added to track host utilizations and the liveness of processes.

Measuring FLARe's memory footprint overhead. To evaluate the effect of FLARe on the memory footprint of a DRE system, we designed a baseline single-threaded server and client application process using TAO's RT-CORBA implementation. The server process activates a single object, and the client process invokes an operation on that object. We measured the memory footprint in one of the blades and then compared the baseline version to a version linked with the FLARe middleware.

Figure 6b shows the memory footprint of the client and server applications with and without FLARe. The figure shows that in the chosen platform, FLARe increases the memory footprint of the client and the server application by 10.2 MB, which stems largely from the memory footprint added by the threads that run the forwarding agents and resource monitors.

On the Linux platform we used for our experiments, the default minimum stack size of a thread is 10,240 Kbytes, which is governed by the constant PTHREAD_STACK_MIN. Every new thread created by an application will thus incur a corresponding increase in its memory footprint. The default value of the stacksize is clearly excessive for the forwarding agent's functionality. For applications with more stringent footprint requirements, it may require recompiling the OS kernel with a much smaller value of the default thread stack size. This result indicates that the footprint overhead is due primarily to the thread stack size, rather than FLARe's infrastructure elements, such as the forwarding agent and resource monitor.

Measuring FLARe's runtime overhead during fault-free conditions. FLARe uses a client request interceptor to catch COMM_FAILURE exceptions and transparently redirect clients to suitable failover targets. CORBA interceptors check every invocation made by the client, when a request is sent to a server, as well as when the reply/exception is received from the server. To evaluate the runtime overhead of these per-request interceptions, we ran a simple experiment with client CL-1 making invocations on server object A-1 with and without client request interceptors. No other processes operated in the processor hosting A-1, so that the response time was equal to the execution time of the server.

We ran this experiment for 50,000 iterations, and measured the average end-to-end response time perceived by CL-1. Figure 6c shows that the average end-to-end response time perceived by CL-1 increased by only 8 microseconds when using the client request interceptor. This result shows that the interceptor adds negligible overhead to the normal operations of a real-time application. Moreover, it provides capabilities to add client redirection transparently *without* modifying TAO's RT-CORBA implementation.

5. Related Work

Our work on FLARe can be compared with related work along two dimensions:

1. Scheduling algorithms. Fundamental ideas and challenges in combining real-time and fault tolerance are described in [20], where imprecise computations are used to provide degraded QoS to applications operating in the presence of failures. [4] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. [6] proposes a fixed priority preemptive scheduling scheme to preallocate time intervals to both the primary and backup replicas of a task, and adaptively executes either the primary or a backup depending on failures and available time. FLARe differs from these approaches in providing fault tolerance capabilities to soft real-time applications. Rather than ensuring hard deadlines are met in the presence of failures, FLARe focuses on minimizing the impact of failure recovery on client response times and system resource utilization, and also provides timely client failover to appropriate failover targets.

2. Real-time fault-tolerant systems. Delta-4/XPA [13] provided real-time fault-tolerant solutions to distributed systems by using the semi-active replication model. MEAD [12] and its proactive recovery strategy for distributed CORBA applications can minimize the recovery time for DRE systems. The Time-triggered Message-triggered Objects (TMO) project [8] considers replication schemes such as the primary-shadow TMO replication (PSTR) scheme, for which recovery time bounds can be quantitatively established, and real-time fault tolerance guarantees can be provided to applications. FLARe builds upon and extends this prior work by focusing on maintaining soft real-time performance after failure recovery.

6 Concluding Remarks

The FLARe middleware described in this paper provides both timeliness and availability to distributed real-time and embedded (DRE) systems. FLARe focuses on passive replication to meet the needs of resource-constrained environments. FLARe overcomes limitations of passive replication for DRE systems by providing a load-aware, proactive and adaptive solution for clients (predictable and fast failover) and servers (overload management). Lessons learned in developing FLARe include:

• Common CORBA features, such as portable interceptors, and POSIX features, such as local sockets, can be leveraged to provide fault tolerance capabilities to soft real-time systems without modifying the implementation of standard-compliant Real-time CORBA ORBs.

• Our experimental results demonstrate the effectiveness and efficiency of FLARe's replica selection and overload management algorithms in the context of multiple different kinds of failure and overload scenarios. The configurability and flexibility offered by FLARe can be used to tune parameters, such as the replica selection and overload management intervals, and to trade off some extra overhead for even faster reaction.

In its current design, FLARe does not support alternative consistency models for stateful fault tolerance and does not handle network failures, which will be the focus of our future work..

References

[1] T. F. Abdelzaher, S. Dawson, W. chang Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. G. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron. ARMADA middleware and communication services. *Real-Time Systems*, 16(2-3):127–153, 1999.

[2] T. Bennani, L. Blain, L. Courtes, J.-C. Fabre, M.-O. Killijian, E. Marsden, and F. Taiani. Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization. In *DSN' 04*, pages 549–554, Florence, Italy, 2004.

[3] D. Corman. WSOA-Weapon Systems Open Architecture Demonstration-Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target (TCT) Prosecution. In *DASC'2001*, Oct. 2001.

[4] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *RTSS '97*, page 79, San Francisco, CA, USA, 1997.

[5] E. Hadad. *Architectures for Fault-Tolerant Middleware Services*. PhD thesis, Technion - Israel Institute of Technology, 2006.

[6] C.-C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Trans. on Comp.*, 52(3):362–372, 2003.

[7] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic migration algorithms for distributed object systems. In *ICDCS '01*, page 119, Phoenix, AZ, USA, 2001.

[8] K. H. K. Kim and C. Subbaraman. The pstr/sns scheme for real-time fault tolerance via active object replication and net-

work surveillance. *IEEE Trans. on Know. and Data Engg.*, 12(2), 2000.

[9] S. Krishnamurthy, W. Sanders, and M. Cukier. A Dynamic Replica Selection Algorithm for Tolerating Timing Faults. *DSN' 01*, pages 107–116, 2001.

[10] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *RTSS' 89*, pages 166–171, 1989.

[11] A. P. A. V. Moorsel. The 'qos query service' for improved quality-of-service decision making in corba. In *SRDS '99*, page 274, Lausanne, Switzerland, 1999.

[12] S. Pertet and P. Narasimhan. Proactive recovery in distributed corba applications. In *DSN '04*, page 357, Florence, Italy, 2004.

[13] D. Powell. Distributed fault tolerance: Lessons from delta-4. *IEEE Micro*, 14(1):36–47, 1994.

[14] P. Ramanathan. Overload management in real-time control applications using m,k (m,k)-firm guarantee. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):549–559, 1999.

[15] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[16] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

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

[18] D. B. Stewart and P. K. Khosla. Real-time Scheduling of Sensor-Based Control Systems. In W. Halang and K. Ramamritham, editors, *Real-time Programming*. Pergamon Press, Tarrytown, NY, 1992.

[19] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP) A Reference Guide*. Addison-Wesley, Boston, 2001.

[20] F. Wang, K. Ramamritham, and J. A. Stankovic. Determining redundancy levels for fault tolerant real-time systems. *IEEE Transactions on Computers*, 44(2):292–301, 1995.

[21] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran. Evaluating Meta-Programming Mechanisms for ORB Middleware. *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, 39(10):102–113, Oct. 2001.

[22] H. Zou and F. Jahanian. A real-time primary-backup replication service. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):533–548, 1999.