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

Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale and Douglas C. Schmidt
Vanderbilt University, Nashville, TN
{jai,sutambe,gokhale,schmidt}@dre.vanderbilt.edu

Chenyang Lu and Christopher Gill
Washington University, St. Louis, MO
{lu,cdgill}@cse.wustl.edu

Abstract

An important class of distributed real-time and embedded (DRE) applications consists predominantly of periodic soft real-time tasks. Timeliness and reliability are both essential requirements for the correct operation of these applications. Conventional solutions to these challenges tend to use non-adaptive and load-agnostic fault tolerance solutions within a real-time system, which often end up making ad hoc failover decisions that can further overload already strained resources. Potential adverse consequences of these ad hoc actions include excessive delays for real-time tasks and cascades of resource failures.

This paper presents FLARe, a middleware that provides a lightweight fault tolerance solution for DRE systems. FLARe uses adaptive and load-aware mechanisms to prevent system overload after failures. We describe the design of FLARe and evaluate its performance on a representative Linux testbed. Our empirical results indicate the effectiveness of our proactive load-aware failover strategy, which significantly reduces client response times and system utilization after failures compared to conventional strategies.

1. Introduction

Many distributed real-time and embedded (DRE) systems, such as shipboard computing systems [24] and and intelligence, surveillance, and reconnaissance systems [25], consist predominantly of soft real-time tasks that must continue to provide real-time quality of service (QoS) even when hardware and software faults occur. For example, the behavior of the object tracking subsystem of a shipboard computing environment is influenced by external sensor readings. The object tracking system should continue to be available and responsive even if processes or processors fail. Likewise, it should continue to provide timely response even when system workload varies significantly at runtime, *e.g.*, due to faults, dynamic task arrival, or intrusion detection.

The Object Management Group (OMG) has worked to address the needs of DRE applications by developing middleware specifications that address one QoS dimension at a time, *e.g.*, Real-Time CORBA (RT-CORBA) [20], which provides capabilities to ensure predictable end-to-end behavior for remote object method invocations, and Fault-Tolerant CORBA (FT-CORBA) [19], which defines services and strategies to enhance the dependability of CORBA applications. It is not possible, however, for CORBA applications to obtain both real-time and fault-tolerance capabilities simply by adopting both standards. Existing mechanisms [6, 21] provide load-agnostic and non-adaptive recovery solutions, which can cause unacceptable delays for real-time tasks and produce further resource failures due to overloads.

What DRE systems need, therefore, are middleware capabilities that integrate real-time and fault tolerance by design, are lightweight, and are adaptive and load-aware so that these solutions can maintain soft real-time performance in the face of failures. This paper describes FLARe, which is middleware that provides an adaptive, lightweight fault tolerance solution for RT-CORBA [20]. The novelty of our approach stems from its proactive ability to provide timely failover and maintain soft real-time performance after processor failures. Specifically, this paper makes three contributions to state-of-the-art middleware for DRE systems:

- *Proactive, timely client redirection* – client-side middleware is updated proactively with suitable failover targets so it can make local request redirection decisions for clients when a server fails.
- *Adaptive, load-aware server failover* – failover targets are chosen based on up-to-date utilization estimates, which maintains timely response to client's requests and avoids system overload after a failover.
- *Lightweight middleware architecture* – standard CORBA interceptors [29] provide client-side enhancements, such as redirection for fault tolerance, extending the basic behavior of RT-CORBA object request brokers (ORBs) transparently.

FLARe has been implemented within the TAO [23] RT-CORBA middleware. We evaluate FLARe on a distributed testbed 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 rebalance system resource utilization after processor failures.

This paper is organized as follows: Section 2 describes our fault model and the lightweight architecture of FLARe; Section 3 evaluates the performance of FLARe in the presence of failures and contrasts its performance with alternatives; Section 4 compares FLARe with related work; and Section 5 presents concluding remarks and summarizes lessons learned.

2. Design of FLARe

This section describes the design of the FLARe middleware. We describe its replication style, real-time system model, and fault model, as well as its software design.

2.1 Replication Style

Conventional fault tolerance solutions replicate servers that may fail independently, giving clients a robust service that appears as though it was provided by a single server. Two common approaches for maintaining replicas are ACTIVE and PASSIVE replication. In ACTIVE replication, a collection of servers maintains identical state, and all client requests are executed atomically in all of the replicas using a group communication service [1]. If a server fails other servers continue to execute the protocol, and clients are not affected by the failures of individual replicas.

Although ACTIVE replication has been used for some hard real-time systems [17] it is expensive for systems that do not require strong state consistency or hard real-time guarantees due to its high resource usage. For such systems, PASSIVE replication may be preferred, where one replica—called the primary—handles all client requests, and backup replicas receive state updates from the primary. If the primary dies, a backup is elected to become the new primary.

Electing the new primary takes time, however, which could affect client failover latency. After failover, client-perceived response times depend on the load on the processor hosting the new primary. Timely failover and choosing the right failover target to maintain timely responses to clients are thus crucial to use PASSIVE replication schemes effectively for DRE systems. Our FLARe middleware is designed to provide these capabilities.

2.2. FLARe's Real-time System Model

FLARe supports DRE systems consisting predominantly of soft periodic real-time tasks, such as those found in shipboard computing and intelligence, surveillance, and reconnaissance systems.[1] For the purposes of this paper, we assume the soft periodic tasks are deployed on a RT-CORBA [20] infrastructure, such as that provided by the TAO middleware [23]. These tasks are scheduled on different nodes of the DRE system, with tasks on each node scheduled using Rate Monotonic Scheduling (RMS) [18].

The client-side request rate defines the priority at which the task will be executed at the server. We therefore use RT-CORBA's `CLIENT_PROPAGATED` priority model. Since a single server process may handle multiple clients with different priorities, we also use the RT-CORBA *thread pool with lanes* feature. While our current implementation employs RMS, our middleware architecture can easily incorporate other scheduling policies, such as Maximum Urgency First [26] (MUF).

2.3. Fault Model

This paper focuses on a fail-stop model of failures, where processes or processors can fail and the remainder of the system can continue executing. We assume that processor faults are *hard* faults, *i.e.,* when a processor has a fault it stops permanently. These types of faults may occur due to aging or acute damage, though in domains like shipboard computing acute damage is the main concern since hardware components are periodically replaced through routine maintenance. Considering unpredictable behavior of processes or processors is beyond the scope of our research.

We assume that networks provide bounded communication latencies and do not fail. This assumption is reasonable for certain DRE systems, such as shipboard computing, 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 an area of future work. Similar to the COLD_PASSIVE replication style used in the FT-CORBA [19] specification, our current middleware implementation assumes that objects are stateless, although the techniques presented here may be integrated with SEMI_ACTIVE replication [11] to support stateful objects.

2.4. Fault-Tolerant Real-time CORBA Middleware

The goal of FLARe is to provide middleware mechanisms that enable timely failover after a process or processor failure, for real-time applications that use PASSIVE replication. We begin by discussing the limitations of the existing FT-CORBA [19] standard for DRE systems.

2.4.1 Limitations of FT-CORBA

To support PASSIVE replication, the FT-CORBA [19] specification collects CORBA objects into *replication groups*. Replica addresses are grouped by a standard mechanism

[1] In such systems we assume that each node has a single processor and hard real-time tasks are provisioned separately, with dedicated hardware and software, and as such are outside the scope of this discussion.

called an *interoperable object group reference* (IOGR), which has a linked list of CORBA *interoperable object references* (IORs), with each member of the list pointing to a server replica IOR. FT-CORBA clients invoke operations using IOGRs, as if they were making invocations using IORs.

If a server object fails in the IOGR model the client-side ORB catches the exception, and cycles through the IORs in the IOGR to handle the request at a different replica. This algorithm ensures faster client failover, and provides clients with a transparent abstraction as though the service was provided by a single server. If none of the IORs in the IOGR list is valid (*e.g.*, if no replicas are live), an exception is propagated to the client application so it can start a recovery process to find a new set of server object addresses.

Although the IOGR provides a standardized, transparent mechanism for client-side failover if a server replica fails, it has the following shortcomings:

- *No seamless integration with RT-CORBA* – Not all RT-CORBA ORBs support the FT-CORBA IOGR feature. Even if it is supported, there are no guidelines for how those features should work with RT-CORBA features like banded connections.
- *Fixed and load-unaware replica selection* – FT-CORBA's mechanism of selecting the next IOR from a linked list provides faster failover. The default FT-CORBA replica selection policy, however, does not consider each server's resource utilization, which may affect client response times after failover. A replica that may be a suitable failover at deployment time may not be so at runtime due to dynamic task arrivals and changing system utilization levels.

These shortcomings of FT-CORBA for DRE systems motivate the lightweight FLARe solution discussed next in Section 2.4.2, which uses recent processor utilizations to select each failover target. Since FLARe maintains updated processor utilization information at runtime, it can also provide an adaptive mechanism to refresh failover targets proactively in the client-side ORB.

2.4.2 Overcoming FT-CORBA's Limitations with FLARe

We now describe how FLARe is designed to overcome the limitations of FT-CORBA described in Section 2.4.1.

FLARe's middleware architecture. Figure 1 depicts the architecture of the FLARe middleware. The major components of FLARe are described below:

Middleware replication manager. FLARe's middleware replication manager provides interfaces for registering and managing information about the server objects and their backup replicas. It also determines the most appropriate server object to which to fail over, when an invocation fails. For each server object, FLARe's middleware replication manager keeps track of the processors hosting their replicas. From each processor, the middleware replication manager collects CPU utilization information, and orders the failover targets according to the utilizations of the processors hosting the replicas.

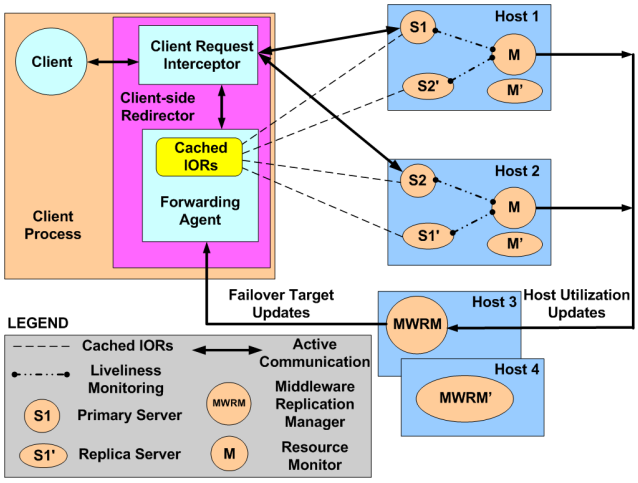


Figure 1: FLARe Middleware Architecture

Client request interceptors. A client-side process contains an application that invokes requests on servers using an RT-CORBA ORB. Clients use the simpler IOR mechanism instead of the IOGR mechanism, so that client applications can make use of RT-CORBA features in a seamless manner. Collocated with the client process is a client request interceptor [29], which is integrated with the ORB at system initialization time.

Client request interceptors 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 a different appropriate server object as part of the client failover process. For any server object managed by FLARe, the middleware replication management maintains information about which server object to use during failover.

Forwarding 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 object failover information to the client request interceptor, a forwarding agent runs in a separate thread within the client process, and the middleware replication manager updates the forwarding agent with the failover information. Upon catching an exception, the client request interceptor contacts the forwarding agent to obtain the failover object address, and redirects the client to that server object.

Resource monitor. FLARe runs a resource monitor on each processor to track the CPU utilization and liveliness of the processes hosted by the processor. On a Linux platform, 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 1 minus the percentage of jiffies spent in idle mode in the last sampling period.

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, it also generates event-driven updates to the middleware replication manager, when utilization levels increase beyond a certain threshold or when a processes 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 forwarding agents of any such change, so that client requests will continue to be redirected to appropriate failover objects whenever failures occur.

FLARe's Replica selection algorithm. FLARe's middleware replication manager uses Algorithm 1 to select a per-object failover target based on the utilization levels of the processors in a DRE system. This algorithm can be run repeatedly over the remaining potential failover targets to provide a total ordering of the failover target list.

FLARe's replica selection algorithm chooses the processor with the lowest utilization from among all processors hosting an object's replicas as its failover target. The expected utilization variable is used to account for the failover decision of other objects located on the same processor. By selecting the processor with the lowest expected utilization, our replication selection algorithm distributes the failover targets of objects on a single processor to multiple processors.

FLARe's FT capabilities. FLARe handles different types of failures as follows:

Algorithm 1 Determine per-object failover targets

```
1:  N = number of processors
2:  for i = 1 to N do
3:      reset expected utilization of all the processors to the
        current utilization
4:      P = number of processes in this processor i
5:      for j = 1 to P do
6:          O = number of objects running in this process j
7:          for k = 1 to O do
8:              find all the processors of the object k's replicas
9:              find the processor MIN with the minimum ex-
                pected utilization
10:             failover target for object k is the object running
                in MIN
11:             expected utilization of processor MIN += object
                k's load
12:         end for
13:     end for
14: end for
```

- *Failure of a server object.* At system initialization time, the forwarding agent in each client process registers with FLARe's middleware replication manager. This manager acknowledges the registration by sending the failover information about the server objects managed by FLARe. Through periodic utilization monitoring at each of the processors, FLARe's middleware replication manager updates the forwarding agent whenever the failover information for a server object changes, which allows clients to failover to the appropriate replica server objects.
- *Failure of a replica server object.* When a process crashes, any replica server objects in that process would fail as a result. Since those failed replica objects could be potential failover targets, upon the detection of a failed server replica, the resource monitor on that same processor notifies the replication manager immediately. The middleware replication manager recomputes the failover information for that server object and updates the forwarding agents of all client processes.
- *Failure of a resource monitor or of the middleware replication manager.* FLARe uses semi-active replication [11, 4] 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.

3. Empirical Evaluation of FLARe

This section describes experiments we conducted to evaluate FLARe's performance and compare its proac-

tive load-aware failover approach with alternative failover strategies.

3.1 Experiment Configurations

The experiments were conducted at ISISlab (`www.dre.vanderbilt.edu/ISISlab`) on a testbed of 15 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 to ensure that the RMS scheduling policy worked properly with our experiment configurations. The blades were connected via a CISCO 3750G switch over a 1 Gbps LAN. As shown in Figure 2, twelve of those blades ran RT-CORBA client/server applications developed using FLARe, which is based on TAO 1.5.8. FLARe's middleware replication manager and its backup replicas ran in the other three blades. The entire FLARe middleware (excluding the code in TAO) was implemented in ∼3,400 lines of C++.

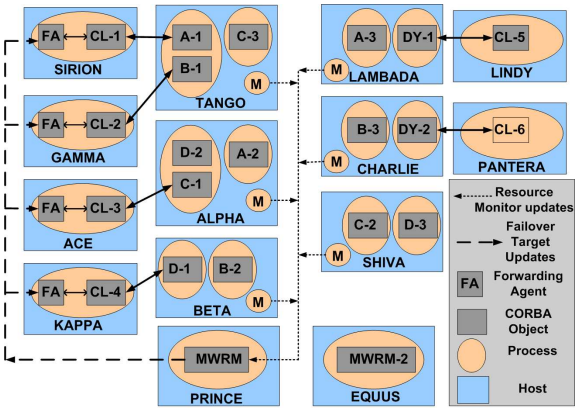


Figure 2: Experiment Setup

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 2.4.2) 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 2.4.2. 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 2.4.2, the strategy adopted by FLARe is both proactive *and* load-aware, where the middleware replication manager proactively pushes failover target updates to clients.

3.2 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 de-

scribed 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. With the static failover strategy, failover decisions are made 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 our proactive load-aware failover strategy, those failover decisions are updated dynamically when and if failures occur, as the processors' 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. The client-perceived end-to-end response time depends on the following factors: (1) CLIENT_REQUEST_DELAY, which is the time taken for the request to traverse the client ORB, the network, and the server ORB, (2) SERVER_DELAY, which is the response time of the server object, and (3) SERVER_REPLY_DELAY, which is the time taken for the reply to traverse the server ORB, the network, and the client ORB. FAULT_DETECTION_DELAY is the time taken for the client to receive a COMM_FAILURE exception after the server object failure. 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. Clients CL-1 and CL-2 make invocations on servers A-1 and B-1, respectively. The request rates of CL-1 and CL-2 are 10 HZ and 5 HZ, respectively. A-1 thus serves requests at higher priority than B-1. At 50 seconds, clients CL-5 and CL-6 start invoking operations on servers DY-1 and DY-2, respectively, which provide dynamic workload changes to the system.

At 150 seconds, when CL-1 makes an invocation on server A-1, our fault injection mechanism crashes the process hosting A-1. CL-1 receives a COMM_FAILURE exception after the connection timeout, and makes a failover to the target A-3, which is already pre-determined in the static strategy. For the invocation at 150 seconds, the end-to-end response time perceived by CL-1 increases by 10.2 milliseconds, as shown in Figure 3a. This increase occurs because the request experiences a FAULT_DETECTION_DELAY. The FAILOVER_DELAY is negligible because the failover target address is readily available at FLARe's client request interceptor. Similarly, Figure 3a shows that for the invocation at 150 seconds, the end-to-end response time perceived by CL-2 increases by 10.3 milliseconds as part of the failover.

The end-to-end response time perceived by CL-2 increases by $\sim$40% after the failover to B-3 on host CHARLIE, which also hosts DY-2 at a higher priority than B-3. The sharp increase in the response time perceived by CL-2 is caused by the high processor utilization of CHARLIE, which increases from 50% to 80% as a result of the failover. This result demonstrates the negative impact of load-unaware failover on the real-time performance of applications.

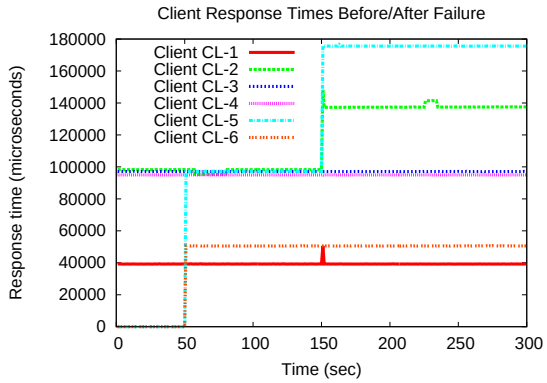
Moreover, the end-to-end response times perceived by CL-5 increases by about 90% after the failover of CL-1 to A-3 in LAMBADA, even though CL-5 did not perform a failover itself. As shown in Figure 3c, this increase occurs because the utilization of LAMBADA grew from 50% to 90% due to the activation of A-3 after the failover, resulting in a sharp increase in the response time of DY-1 hosted on LAMBADA. This result shows that load-unaware failovers can severely affect the real-time performance of already-active servers as well as the response times of their respective clients.[2]

Figure 3b shows the end-to-end response times perceived by clients CL-1, CL-2, CL-5, and CL-6 when they are configured to use the proactive load aware strategy. CL-1, CL-2, CL-5, and CL-6 invoke operations with the same configurations they had for the static strategy experiment. At 0 seconds, CL-3 and CL-4 make invocations on servers C-1 (hosted on ALPHA) and D-1 (hosted on BETA) respectively. The request rates of CL-3 and CL-4 are 2 HZ and 1 HZ, respectively. Figure 3d shows the utilizations of all the processors. At 50 seconds, the utilizations of LAMBADA and CHARLIE increased 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.

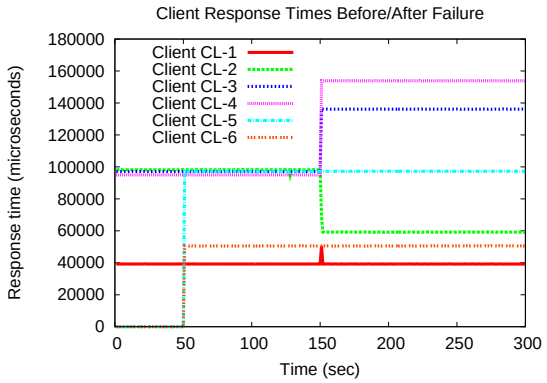
At 150 seconds, as described earlier, the fault injection mechanism crashes the process hosting A-1 and B-1[3]. The

[2] As shown in Figure 3a, the end-to-end response times perceived by clients CL-1 and CL-6 did not change after the failover because they are served by highest priority server objects.

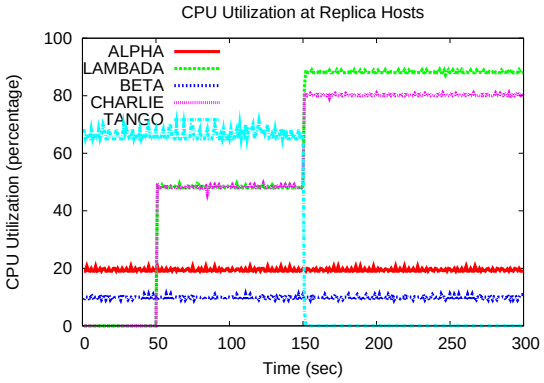
[3] The FAILOVER_DELAY for CL-1 and CL-2 is the same as in static



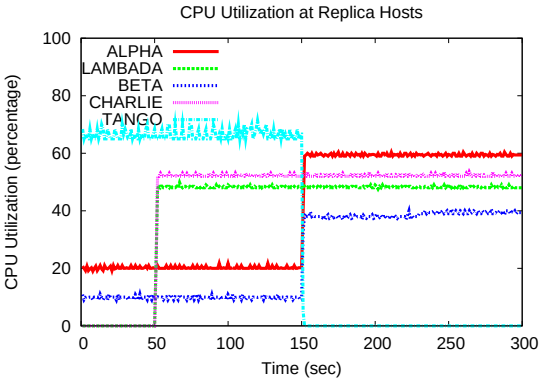
(a) End-to-end response times with static strategy



(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

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.

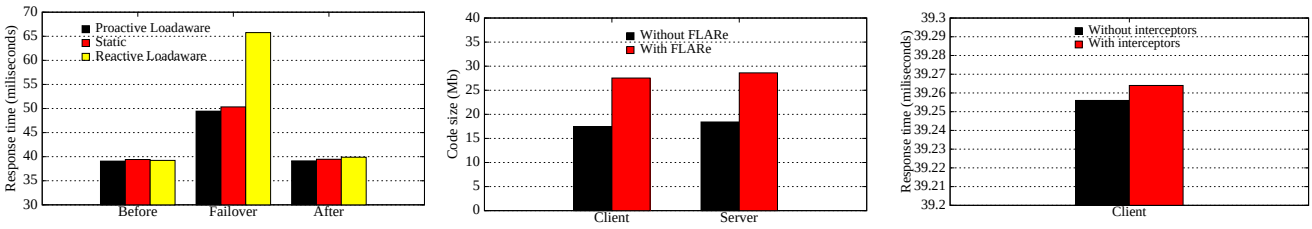
Figure 3d shows the utilizations of processors ALPHA and BETA, where the failover targets were hosted. This figure shows that after failover, the utilizations of these processors are similar to the utilizations of processors LAMBADA and CHARLIE, where the other replicas of the failed objects A-1 and B-1 are hosted. This result is in contrast to the processor utilizations with the static strategy as shown in Figure 3c, where the utilizations of the processors hosting the failover targets were 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, our proactive load-aware strategy not only provided timely responses to the failing over clients, but also did not affect the already-active servers. As shown in Figure 3b, the end-to-end response times of CL-3 and CL-4 increased by 35% and 50% after the failover. This result is much better than the 90% increase in end-to-end response times for CL-5 in the static strategy.

3.3 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

strategy. The end-to-end response time perceived by CL-1 does not change after failover because of being served by highest priority server object.



(a) Failover delay (b) Code size overhead (c) Runtime overhead

Figure 4: Failover delay and overhead measurements

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 5001^{st} 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 4a 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 3.2, the results in Figure 4a clearly show that the proactive load-aware strategy is better than either the reactive load-aware or static failover strategy, and thus is more suitable for use in DRE systems.

3.4 Overhead Measurements

FLARe provides fault tolerance capabilities to DRE systems using a lightweight middleware architecture, as described in Section 2.4.2. 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 4b 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 mea-

sured the average end-to-end response time perceived by CL-1. Figure 4c 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.

4. Related Work

Fundamental ideas and challenges in combining real-time and fault tolerance are described in [28], where the notion of imprecise computations have been used to provide degraded QoS to applications operating in the presence of failures. [12] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. [9] proposes a feasibility test to determine if a given task set is schedulable for fault-tolerant purposes using earliest deadline first (EDF) scheduling. [14] proposes a fixed priority-driven 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. [15] generates a FT schedule for tasks with precedence constraints and plans for sufficient slack time to handle recovery actions in case of failures. 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, therefore, 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.

Other research has focused on deployment-time allocation of resources to tasks operating in a multi-processor environment while considering fault tolerance. [10] focuses on choosing appropriate task implementations and degrees of replication for fault tolerance depending on system resource availability. [13] proposes a fully polynomial-time approximation algorithm to map tasks and their replicas to heterogeneous multiprocessors. [3] proposes a bi-criteria heuristic for scheduling operations in heterogeneous architectures while minimizing schedule length and maximizing reliability. [7] proposes a polynomial-time approximation scheme for replication of periodic hard real-time tasks in identical multiprocessor environments while minimizing system utilization. The FLARe middleware can be extended readily to support deployment-time allocation planning using such algorithms. Furthermore, as failures occur and tasks arrive dynamically at run-time, FLARe can also adapt by changing failover targets on the fly so that client response times are not overly affected by failures.

Delta-4/XPA [22] was an early effort to provide real-time fault-tolerant solutions to distributed systems by using the semi-active replication model, where all the replicas are active, but only one replica sends output responses. ARMADA [2] defines a set of communication and middleware services that support fault tolerance and end-to-end guarantees for real-time distributed applications. MEAD [21] 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 [16] 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's research contributions are similar to these projects in providing modular middleware services to add fault tolerance capabilities to object-based systems. FLARe also enhances traditional fault tolerance techniques with utilization monitoring techniques, however, so as to minimize the effect of recovery on client response times, and to manage system resources efficiently.

5. Concluding Remarks

This paper describes the design and performance of FLARe, which is a lightweight middleware that enhances RT-CORBA to provide adaptive and load-aware fault tolerance solutions for DRE systems. The 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 DRE systems without modifying the implementation of standard-compliant RT-CORBA ORBs.
- FLARe's proactive load-aware failover strategy can support transparent and timely failure handling for DRE applications by selecting failover targets on processors with the least load, thereby minimizing the impact of failures, such as unpredictable system utilization and increased client-perceived end-to-end response times.
- FLARe is currently designed for environments where the networks provide bounded communication latencies and have no single point of failure. Certain DRE systems, however, may run in environments where networks fail, causing severe resource contention in the remaining links. Our future work therefore focuses on integrating FLARe with network QoS mechanisms like DiffServ [5] and Bandwidth Brokers [8], so that critical communications can use network QoS mechanisms to meet critical QoS requirements. We are also investigating advanced error detection capabilities by integrating SCTP [27] with FLARe.
- Supporting stateful applications in DRE systems not only requires timely failover, but client consistency re-

quirements, such as weak or strong consistency models. FLARe is currently designed for stateless applications, so our future work will enhance the replica selection algorithm to consider consistency levels of the replicas while choosing failover targets. We are also enhancing FLARe to support replication requirements for different consistency levels.

FLARe is available in open-source format from `www.dre.vanderbilt.edu/~jai/FLARe/`.

References

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. Rtcast: lightweight multicast for real-time process groups. *IEEE RTAS*, 00:250, 1996.

[2] 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.

[3] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *DSN '04*.

[4] P. Barrett, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *Proceedings of the 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pages 481–488, June 1990.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *Internet Society, Network Working Group RFC 2475*, pages 1–36, Dec. 1998.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[7] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng. Real-time task replication for fault tolerance in identical multiprocessor systems. *IEEE RTAS*, 0:249–258, 2007.

[8] B. Dasarathy, S. Gadgil, R. Vaidhyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot. Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework. In *IEEE RTAS*, 2005.

[9] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. In *RTSS '95*.

[10] S. Ghosh, R. Rajkumar, J. Hansen, and J. Lehoczky. Scalable resource allocation for multi-processor qos optimization. In *ICDCS '03*.

[11] A. S. Gokhale, B. Natarajan, J. K. Cross, D. C. Schmidt, C. Andrews, S. J. Fernandez, and C. D. Gill. Towards Real-time Support in Fault-tolerant CORBA. In *Proceedings of the Workshop on Dependable Middleware-Based Systems*, Washington, DC, June 2002.

[12] 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*.

[13] S. Gopalakrishnan and M. Caccamo. Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems. In *RTAS 2006*.

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

[15] N. Kandasamy, J. P. Hayes, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Transactions on Computers*, 52(2):113–125, 2003.

[16] K. H. K. Kim and C. Subbaraman. The pstr/sns scheme for real-time fault tolerance via active object replication and network surveillance. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):145–159, 2000.

[17] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 09(1):25–40, 1989.

[18] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-time Systems Symposium (RTSS 1989)*, pages 166–171. IEEE Computer Society Press, 1989.

[19] Object Management Group. *Fault Tolerant CORBA, Chapter 23, CORBA v3.0.3*, OMG Document formal/04-03-10 edition, Mar. 2004.

[20] Object Management Group. *Real-time CORBA Specification v1.2 (static)*, OMG Document formal/05-01-04 edition, Nov. 2005.

[21] S. Pertet and P. Narasimhan. Proactive Recovery in Distributed CORBA Applications. In *DSN 2004*.

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

[23] 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.

[24] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk - The Journal of Defense Software Engineering*, Nov. 2001.

[25] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan. Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[26] 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.

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

[28] 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.

[29] 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.