Adaptive Failover for Real-time Middleware with Passive Replication

Jaiganesh Balasubramanian[†], Sumant Tambe[†], Chenyang Lu[‡], Aniruddha Gokhale[†],
Christopher Gill[‡], and Douglas C. Schmidt[†]
[†]Department of EECS, Vanderbilt University, Nashville, TN, USA
[‡]Department of CSE, Washington University, St. Louis, USA

Abstract

Supporting uninterrupted services for distributed soft real-time applications is hard in resource-constrained and dynamic environments, where processor or process failures and system workload changes are common. Fault-tolerant middleware for these applications must achieve high service availability and satisfactory response times for client applications. Although passive replication is a promising fault tolerance strategy for resource-constrained systems, conventional client failover approaches are non-adaptive and load-agnostic, which can cause system overloads and significantly increase response times after failure recovery.

This paper presents four contributions to the study of passive replication for distributed soft real-time applications. First, it describes how our Fault-tolerant Load-aware and Adaptive middlewaRe (FLARe) dynamically adjusts failover targets at runtime in response to system load fluctuations and resource availability. Second, it describes how FLARe's overload management strategy proactively enforces desired CPU utilization bounds by redirecting clients from overloaded processors. Third, it presents the design and implementation of FLARe's lightweight middleware architecture that manages failures and overloads transparently to clients. Finally, it presents experimental results on a distributed Linux testbed that demonstrate how FLARe adaptively maintains soft real-time performance for clients operating in the presence of failures and overloads with negligible runtime overhead.

1 Introduction

Distributed real-time middleware, such as Real-time CORBA [22] and Distributed Real-time Java [18], has been used to develop a range of distributed soft real-time applications, such as online stock trading systems and supervisory control and data acquisition (SCADA) systems. Such applications operate in dynamic environments where system loads and resource availabilities fluctuate significantly at runtime due to service request arrivals and processor failures. In such environments, it is important for applications to maintain both system availability and desired soft real-time performance. For example, in SCADA systems for power grid monitoring, remote terminal units must continue to process updates from sensors monitoring power grid failures, even when load fluctuations and failures occur.

ACTIVE and PASSIVE replication [17] are two common approaches for building fault-tolerant distributed applications. In ACTIVE replication [25], client requests are multicast and executed at all replicas. Failure recovery is fast because if any replicas fail, the remaining replicas can continue to provide the service to the clients. ACTIVE replication imposes high communication and processing overhead, however, which may not be viable in resource-constrained systems [8].

In PASSIVE replication [6] only one replica—called the primary—handles all client requests, and backup replicas do not incur runtime overhead, except for receiving state updates from the primary. If the primary fails, a failover is triggered and one of the backups becomes the new primary. Due to its low resource consumption, PASSIVE replication is appealing for soft real-time applications that cannot afford the cost of maintaining active replicas and need not assure hard real-time performance.

Although PASSIVE replication is desirable in resource-constrained systems, it is challenging to deliver soft real-time performance for applications based on PASSIVE replication. In particular, conventional client failover solutions [4, 23] in PASSIVE replication are non-adaptive and load-agnostic, which can cause post-recovery system overloads and significantly increase response times for clients. Moreover, the middleware must dynamically handle overload conditions caused by workload fluctuations and concurrent failures. Therefore, a lightweight middleware architecture is needed that can handle failures and overloads transparently from the applications.

To address this need, we have developed the *Fault-tolerant, Load-aware and Adaptive middlewaRe (FLARe)* which maintains service availability and soft real-time performance in dynamic environments. This paper evaluates

1080-1812/09 $25.00 © 2009 IEEE
DOI 10.1109/RTAS.2009.36

the following contributions to developing distributed soft real-time applications:

- A **Load-aware Adaptive Failover (LAAF) strategy**, which dynamically adjusts failover targets in response to load fluctuations and processor/process failures based on current CPU utilization.
- A **Resource Overload Management rEdirector (ROME) strategy**, which dynamically enforces schedulable utilization bounds by proactively redirecting clients from overloaded processors.
- A **lightweight adaptive middleware architecture**, which handles failures and overloads transparently from applications.

FLARe has been implemented atop the TAO Real-time CORBA middleware (`www.dre.vanderbilt.edu/TAO`) and evaluated empirically in the ISISlab testbed (`www.dre.vanderbilt.edu/ISISlab`). The experimental results reported in this paper demonstrate how FLARe can dynamically maintain both system availability and desired soft real-time performance for clients, while incurring negligible run-time overhead.

The remainder of this paper is organized as follows: Section 2 describes the system and fault models that form the basis for our work on FLARe; Section 3 describes the structure and functionality of FLARe; Section 4 empirically evaluates FLARe in the context of distributed soft real-time applications with dynamic application arrivals and failures; Section 5 compares FLARe with related research; and Section 6 presents concluding remarks.

2 System and Fault Models

FLARe supports distributed systems where application servers provide multiple long-running services on a cluster of computing nodes. The services in a system are invoked by clients periodically via remote operation requests. Further, these types of systems experience *dynamic* workloads when clients start and stop services at runtime. Clients demand both soft real-time performance as well as system availability despite workload fluctuations and processor and process failures.

The end-to-end delay of a remote operation request comprises delays on the server, the client, and the network. FLARe is designed to bound server latencies, which often dominate in distributed real-time systems (*e.g.*, SCADA systems) equipped with high-speed networks. To meet desired server latencies FLARe allows users to specify a utilization bound for each CPU on the servers. The utilization bound can be set to below the schedulable utilization bound of the real-time scheduling policy (*e.g.*, rate monotonic) supported by the middleware scheduling service. At run time FLARe maintains desired server latencies by dynamically enforcing the utilization bounds on the servers [1].

Processors and processes may experience fail-stop [25] failures and concurrent failures in multiple processors or processes can occur. To provide lightweight fault-tolerance, FLARe employs PASSIVE replication [7], where services are replicated and deployed across multiple processors. We assume that networks provide bounded communication latencies and do not fail or partition. This assumption is reasonable for many soft real-time systems, such as SCADA systems, where nodes are connected by highly redundant high-speed networks. Relaxing this assumption through integration of our middleware with network-level fault tolerance and QoS management techniques [1] is an area of future work.

3 Design and Implementation of FLARe

This section describes the design and implementation of FLARe. The key design goals of FLARe are to (1) mask clients from processor and process failures through transparent client failover, (2) alleviate post recovery overload through load-aware failover target selection, and (3) maintain desired soft real-time performance by dynamically enforcing suitable CPU utilization bounds on the servers through overload management.

3.1 FLARe Middleware Architecture

FLARe's architecture, shown in Figure 1, has four main components: the *middleware replication manager*, the *client failover manager* for each client process, the *monitor* on each processor hosting servers, and the *state transfer agent* on each process hosting servers. FLARe achieves fault-tolerance through PASSIVE replication of CORBA objects, where the primary and backup replicas are deployed across different processors in the distributed system.

Middleware replication manager. FLARe's *middleware replication manager* (MRM) allows server objects to provide information about (1) the processors and processes in which their primaries and backups are hosted, (2) the CPU utilization that they will require to serve client requests should they become primary, and (3) their interoperable object reference (IOR) so that clients can invoke remote operations on them when the server objects are added to the system. To manage the primary and backup replicas—and to make adaptive failover target decisions—FLARe's MRM uses a *monitor* on each processor to track failures and CPU utilizations of all processors hosting the primary and backup replicas of each server object.

As highlighted by label A in Figure 1, FLARe's MRM employs a *Load-Aware and Adaptive Failover* (LAAF) tar-

[1] FLARe is targeted at *soft* real-time applications and does not provide hard guarantees on meeting every deadline

get selection algorithm (described in Section 3.2) to prepare an rank-ordered list of failover targets for each *primary* object in the system. The rank list includes multiple failover targets in order to handle multiple failures of the same server object. In some situations the current *primary* replica can become overloaded, *e.g.*, due to sudden workload fluctuations and multiple failures. FLARe's MRM employs the *Resource Overload Management rEdirector* (ROME) algorithm (described in Section 3.3) to redirect clients from overload processors to maintain the desired soft real-time performance. The LAAF and ROME strategies are detailed in Section 3.2 and Section 3.3, respectively. Finally, MRM could be co-located with server objects (*i.e.,* Host 1 or Host 2 in Figure 1) as the computation load of the LAAF and ROME algorithms implemented in MRM is relatively low compared to that of the server objects.

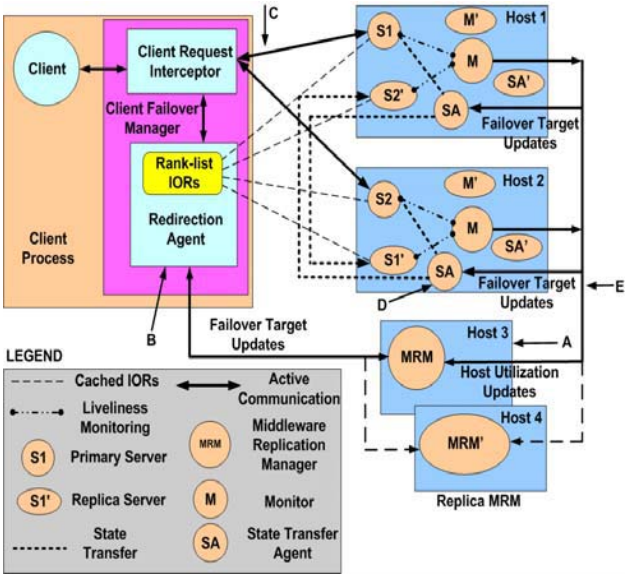


Figure 1: **The FLARe Middleware Architecture**

Monitors. The liveliness of the processes hosting the server objects and CPU utilization of the hosts is probed by monitors co-located with the server objects. Failures of processes, if any, are communicated instantaneously to the MRM whereas the CPU utilization is communicated at a configurable sampling rate. We do not, however, require fine-grained time synchronization since the sampling period is typically longer than the task periods. For instance, the task periods in the experiments described in Section 4 vary from one second to one-tenth of a second whereas the monitor sampling period is greater than one second.

Client failover manager. As highlighted by label B in Figure 1, FLARe's *client failover manager* contains a *redirection agent* that is updated with failover and redirection targets so clients can recover transparently from failures and overloads, respectively. To handle failures, as highlighted by label C in Figure 1, FLARe's *client request interceptor* catches failure exceptions and instead of propagating the exception to the client application, the client request interceptor redirects the client invocation to the appropriate failover target provided by the redirection agent.

State transfer agent. As highlighted by label D in Figure 1, FLARe's *state transfer agent* allows server objects to inform it about changes to application states. The *state transfer agent* is updated with per-server-object failover targets by FLARe's MRM. When a *primary* replica in a process informs it about application state change, the *state transfer agent* utilizes interfaces provided by the server object to obtain the new state. The *state transfer agent* synchronizes the state of the *backup* replicas with the new state, by making remote invocations on the *backup* replicas using the provided failover target references as highlighted by label E in Figure 1.

FLARe schedules state update propagations from the primary replica to the backup replicas using remote operation requests, from the state transfer agent on the primary replica to one of the backup replicas. The period of the state update task is equal to the period of the primary task. In the current implementation, each state update task is scheduled on the processor hosting the backup replicas at the priority determined by the rate-monotonic scheduling algorithm.

To support distributed soft real-time applications in FLARe, the *primary* replica updates the states of its *backup* replicas *after* it sends its response to the client. This design choice significantly reduces the response times for clients, but supports only "best effort" guarantees for state synchronization. Replica consistency may be lost if the *primary* replica crashes after it responds to the client, but before it propagates its state update to the *backup* replicas. This design tradeoff is desirable in many distributed soft real-time applications where state can be reconstructed using subsequent (*e.g.*, sensor) data updates at the cost of transient degradation of services.

3.2 Load-aware and Adaptive Failover

As described in Section 3.1, FLARe's MRM collects periodic measurement updates from the monitors about CPU utilizations and liveness of processors/processes. FLARe provides a *load-aware, adaptive failover (LAAF)* target selection algorithm that uses these measurements to select per-object failover targets. LAAF uses the following inputs: (1) the list of processors and the list of processes in each processor, (2) the list of primary object replicas operating in each process, (3) the list of backup replicas for each primary object replica and the processors hosting those replicas, and (4) the current CPU utilizations of all processors in the system. This algorithm is executed whenever there is a change in the CPU utilization by a *threshold* (*e.g.*, $\pm$ 10%) in any of the processors in the system since FLARe must

react to such dynamic changes.

The output of LAAF is a ranked list of failover targets for each primary object replica in the system. To deal with concurrent failures, FLARe maintains an ordered list of failover targets, instead of only the first one. When both the primary replica and some of its backup replicas fail concurrently, the client can failover to the first backup replica in the list that is still alive. LAAF estimates the post-failover CPU utilizations of processors hosting backup replicas for a primary object, assuming the primary object fails. The backup replicas are then ordered based on the estimated CPU utilizations of the processors hosting them, and the backup replica whose host has the lowest estimated CPU utilization is the first failover target of the replica. To balance the load after a processor failure, LAAF redirects the clients of different primary objects located on the same processor to replicas on different processors. Finally, the references (IORs) to those replicas are collected in a list and provided to the redirection agents for use during failure recovery. To reduce the failover delay, MRM *proactively* updates a client whenever its failover target list changes.

Algorithm 1 LAAF Target Selection Algorithm

```
1:  P_i : Set of processes on processor i
2:  O_j : Set of primary replica objects in process j
3:  R_k : list of processors hosting backup replicas for a primary object k
4:  cu_i : current utilization of processor i
5:  eu_i : expected utilization of processor i after failovers
6:  l_k : CPU utilization attributed to primary object k
7:  for every processor i do
8:      eu_i = cu_i // reset expected utilization
9:      for every process j in P_i do
10:         for every primary object k in O_j do
11:             sort R_k in increasing order of expected CPU utilization
12:             eu_x += l_k, where processor x is the head of the sorted list R_k
13:         end for
14:     end for
15: end for
```

Algorithm 1 depicts the steps in the LAAF target selection algorithm. For every processor in the system (line 7), LAAF iterates through all hosted processes (line 9), and the primary replicas that are hosted in those processes (line 10). For every primary replica, the algorithm determines the processors hosting its backup replicas and the least loaded of those processors (line 11). The algorithm then adds the load of the primary object replica (known to FLARe's MRM because of the registration process as explained in Section 3.1) to the load of least loaded processor and defines that as the *expected utilization* of that processor (line 12) were such a failover to occur.

The algorithm repeats the process described above for every other primary replica object hosted in the same process (Lines 10–12). The least loaded failover processor is determined by considering the expected utilizations of the processors (line 11). This decision allows the algorithm to consider the failover of co-located primary replica objects within a processor while determining the failover targets of other primary replica objects hosted in the same processor. The failover target selection algorithm therefore makes decisions not only based on the dynamic load conditions in the system (which are determined by the monitors), but also based on load additions that may be caused by failovers of co-located primary objects. The failover targets are then used for redirecting a client if any failure occurs before the next time LAAF is run.

LAAF is optimized for multiple process failures or single processor failures. It may result in suboptimal failover targets, however, when multiple processors fail concurrently. In this case, clients of objects located on different failed processors may failover to a same processor, thereby overloading it. Similarly, LAAF may also result in suboptimal failover targets when process/processor failures and workload fluctuation occur concurrently, *i.e.*, before FLARe's MRM receives the updated CPU utilization from the monitors. To handle such overload situations FLARe employs the ROME algorithm (described next in Section 3.3) to redirect clients of overloaded processors, proactively to less loaded processors.

3.3 Resource Overload Management and Redirection

FLARe's MRM employs the *Resource Overload Management and rEdirection (ROME)* algorithm to enforce desired CPU utilization and service delay bounds. FLARe allows users to specify a per-processor *utilization bound* based on the schedulable utilization bound of the real-time scheduling policy (*e.g.*, rate monotonic) supported by the middleware scheduling service. A processor whose CPU utilization exceeds the *utilization bound* is considered overloaded.

In the case of failures, the clients are redirected to appropriate failover targets based on decisions made by LAAF, as described in Section 3.2. In the case of overloads, clients of the current primary replicas are redirected automatically to the chosen new backup replicas. We refer to this load redistribution mechanism as *lightweight migration* since we migrate the *loads* (through client redirection) of objects as opposed to the less efficient alternative of migrating the *objects* themselves.

Algorithm 2 depicts the steps ROME uses to handle CPU overload and load imbalance, respectively.

Algorithm 2 Determine Load-redistributing Targets

```
1:  O_i : list of primary objects in an overloaded 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 migrations
5:  l_j : CPU utilization of primary object j
6:  t_i : upper bound for processor i's CPU utilization
7:  eu_i = cu_i, for every processor i
8:  for every overloaded processor i do
9:     sort O_i in decreasing order of their CPU utilizations
10:    for every object j in the sorted list O_i do
11:       min : processor i in R_j with lowest CPU utilization
12:       if (l_j + eu_min) < t_min then
13:          migrate the load of object j to j's replica in min
14:          eu_min += l_j
15:          eu_i -= l_j
16:       end if
17:       if eu_i < t_i then
18:          processor i is no longer overloaded; stop
19:       else
20:          migrate another primary object j in the proces-
             sor i
21:       end if
22:    end for
23: end for
```

Handling overloads. When the CPU utilization at any of the processor crosses the *utilization bound*, FLARe's MRM triggers ROME to react to the overloads. FLARe determines the primary objects whose clients need to be redirected, and their target hosts, using ROME. Given an overloaded processor (*i.e.*, whose CPU utilization exceeds the *utilization bound*), ROME considers the primary objects on the processor in decreasing order of CPU utilization (line 9), and attempts to migrate the load generated by those objects to the least-loaded processor hosting their backup replicas (lines 11 through 15). The attempt fails if the least-loaded processor of the backup replicas would exceed the *utilization bound* if the migration occurs. ROME attempts migrations until (1) the processor is no longer overloaded or (2) all clients of primary objects in the overloaded processor have been considered for redirection.

Similar to LAAF, ROME 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 redirection decisions affecting the overloaded processor. After new reconfigurations are identified, redirection agents are updated to redirect existing clients from the current primary replica to the selected backup replica at the start of the next remote invocation. Clients are thus redirected to new targets with less perturbations.

3.4 Implementation of FLARe

FLARe has been implemented atop the TAO Real-time CORBA middleware. It is implemented in ∼9,000 lines of C++ source code (excluding the code in TAO). Below we highlight several key aspects of the FLARe implementation (a more detailed description of FLARe appears in [2]).

Monitoring CPU utilization and processor failures. On Linux, FLARe's *monitor* process uses the `/proc/stat` file to estimate the CPU utilization (*i.e.*, the fraction of time when the CPU is not idle) in each sampling period. We chose to measure the CPU utilization online, rather than relying on the estimated CPU utilization provided by users to account for estimation errors and for other activities in the middleware and OS kernel.

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 monitor. The monitor connects to the socket and performs a blocking read. If an application process crashes, the socket and the opened port will be invalidated, in which case the monitor receives an invalid read error on the socket that indicates the process crash. Fault tolerance of the monitor processes is also achieved through passive replication. If the *primary* monitor replica fails to send updated information or to respond to FLARe's *middleware replication manager* (described below) within a timeout period, FLARe suspects that the processor has crashed.

Middleware replication manager. FLARe's *middleware replication manager* is designed using the Active Object pattern [26] to decouple the reporting of a load change or a failure from the process. This decoupling allows several monitors to register with FLARe's *middleware replication manager* while allowing synchronized access to its internal data structures. Moreover, FLARe can be configured with the LAAF and ROME algorithms via the Strategy pattern [11]. FLARe's *middleware replication manager* is replicated using SEMI_ACTIVE replication [14] (provided by the TAO middleware), with regular state updates to the backup replicas.

Client failover manager. As shown in Figure 1, the client's failover manager comprises a CORBA portable interceptor-based *client request interceptor* [27] and a redirection agent, which together coordinate to handle failures in a manner transparent to the client application logic. Whenever a primary fails, the interceptor catches the CORBA COMM_FAILURE exception. Since portable interceptors are not remotely invocable objects, it was not feasible for an external entity (such as a MRM) to send the rank list information to the interceptor, which is necessary to determine the next failover target. The redirection agent is therefore a CORBA object that runs in a separate thread from the interceptor thread. The interceptor consults the

redirection agent for the failover target from the rank list it maintains. The interceptor will then reissue the request to the new target. The rank list is propagated to the redirection agent *proactively* by FLARe's MRM whenever the failover target list changes.

4 Empirical Evaluation of FLARe

We empirically evaluated FLARe 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 into a 1 Gbps LAN. 12 of the blades ran Real-time CORBA applications on FLARe. FLARe's MRM and its backup replicas ran in the other 2 blades. To emulate distributed soft real-time applications, the clients in these experiments used threads running in the Linux real-time scheduling class to invoke operations on server objects at periodic intervals. All operations and state updates on the servers were executed according to the rate monotonic scheduling policy supported by the TAO scheduling service.

4.1 Evaluating LAAF

The first experiment was designed to evaluate FLARe's LAAF algorithm (described in Section 3.2) and compare it with the optimal *static* client failover strategy. In the static client failover strategy, the client middleware is initialized with a *static* list of IORs of the backup replicas, ranked based on the CPU utilization of their processors at *deployment time*. The list is not updated at run-time based on the current CPU utilizations in the system (the failover targets are optimal at deployment time, but any *static* failover target can become suboptimal at run-time in face of dynamic workloads). In contrast, LAAF dynamically recomputes failover targets whenever there is a change in the CPU utilization by a *threshold* (*e.g.*, $\pm$ 10%) in any of the processors in the system.

Experiment setup. Figure 2 and Table 1 illustrate our experimental setup. The experiment ran for 300 seconds. To evaluate FLARe in the presence of *dynamic workload changes*, at 50 seconds after the experiment was started, we introduced dynamic invocations on two server objects DY-1 and DY-2, using client objects, CL-5, and CL-6, respectively. The *static* failover strategy selects failover targets that are optimal at deployment time, as follows: if A-1 fails, contact A-3 followed by A-2; if B-1 fails, contact B-3 followed by B-2.

We emulated a process failure 150 seconds after the experiment started. We used a fault injection mechanism, where when clients CL-1 or CL-2 make invocations on server objects A-1 or B-1, respectively, the server objects 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 failover to replicas chosen by the failover strategy.

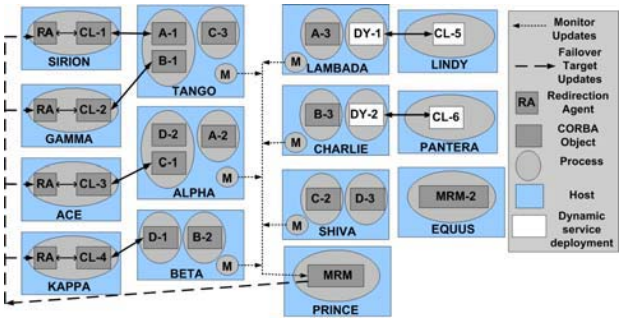


Figure 2: **Load-aware Failover Experiment Setup**

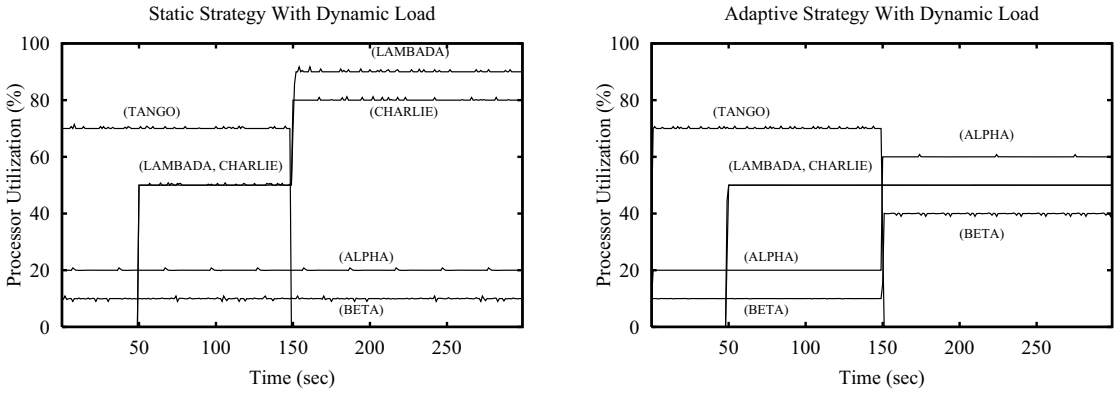
Analysis of results. Figure 3a shows the CPU utilizations at all the processors, when clients used the static client failover strategy. At 50 seconds, servers DY-1 and DY-2 were invoked by clients CL-5 and CL-6 causing the CPU utilizations at processors LAMBADA and CHARLIE to increase from 0% to 50%.

At 150 seconds when process hosting both A-1 and B-1 fails on the processor TANGO, clients CL-1 and CL-2 failover to the statically configured replicas A-3 at processor LAMBADA and B-3 at processor CHARLIE respectively. As a result, the CPU utilizations at processors LAMBADA and CHARLIE increase to 90% and 80% respectively. Note that 90% CPU utilization is highly undesirable in middleware systems because it is close to saturating the CPU which may result in kernel starvation and system crash [21]. The high CPU utilizations on processors CHARLIE and LAMBADA occur, because the *static* client failover strategy did not account for *dynamic* system loads while determining client failover targets.

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

Table 1: **Experiment setup for LAAF**

In contrast, FLARe's MRM triggers LAAF to recompute the failover targets in response to load changes. At 50 seconds, LAAF changed the failover target of the primary replica A-1 from A-3 to A-2, in response to the load



(a) Utilization with static strategy

(b) Utilization with adaptive strategy

Figure 3: CPU utilizations with static and adaptive failover strategies

increase on processor LAMBADA (host of A-3). Similarly, LAAF also changed the failover target of B-1 from B-3 to B-2 in response to the load increase on processor CHARLIE (host of B-3). At 150 seconds, clients CL-1 and CL-2 failover to backup replicas A-2 and B-2 respectively. As shown in Figure 3b, none of the processor utilizations is greater than 60% after the failover of clients CL-1 and CL-2. This result shows that LAAF effectively alleviates processor overloads after failure recovery, due to its adaptive and load-aware failover strategy.

4.2 Evaluating ROME

We designed two more experiments to evaluate the ROME algorithm described in Section 3.3. We stress-tested ROME under overloads caused by dynamic workload changes and multiple failures.

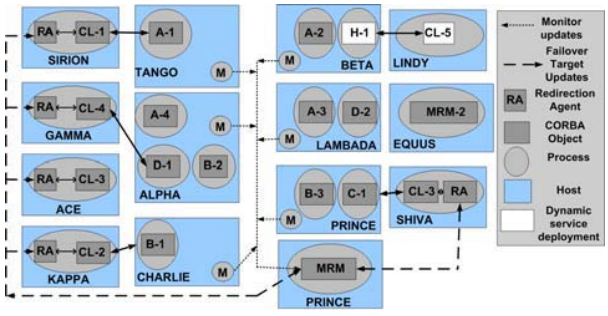


Figure 4: **Overload Redirection Experiment Setup**

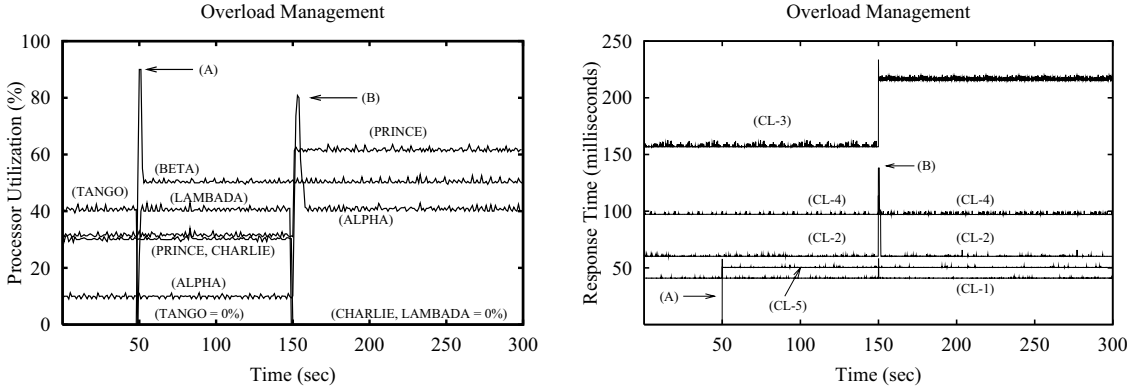
Experiment setup. Figure 4 and Table 2 show the experimental setup. The utilization bound on every processor was set to 70%, which is below the schedulable utilization bound (based on the number of tasks) for the rate monotonic policy supported by the middleware scheduling service. The required server delay for each task equalled its invocation period.

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
Static Loads			
CL-1	A-1	10	40%
CL-2	B-1	5	30%
CL-3	C-1	2	30%
CL-4	D-1	1	10%
Dynamic Loads			
CL-5	H-1	10	50%

Table 2: **Experiment setup for ROME**

Concurrent Workload Change and Process Failure. We emulated a failure 50 seconds after the experiment started. We used a fault injection mechanism, where when client CL-1 makes invocations on server object A-1, the server object calls the *exit (1)* command, crashing the process hosting server object A-1 on the processor TANGO. The client CL-1 receives a COMM_FAILURE exception due to the failure of A-1, and then consults its rank list to make a failover decision, which is A-2. At the same time, a client CL-5 starts making invocations on a new service H-1.

As a result of the concurrent failure and workload change, the load on the processor BETA rises to 90% (highlighted by point A in the Figure 5a), which exceeds the specified utilization bound (70%) and consequently triggers ROME. ROME then performs a lightweight migration of the clients of A-2 and redirects all of its clients to A-3, which is hosted in the least loaded of all the processors hosting a replica of A-1. Within 1 second, the utilization of processor BETA decreases to 50%, while the utilization of processor LAMBADA increases to 40% due to A-3 becom-



(a) Utilizations with overload management

(b) Client response times with overload management

Figure 5: Evaluation of ROME

ing the new primary replica.

At this stage, the CPU utilizations of all processors are below 70%. We also plot the measured end-to-end response times perceived by the clients in Figure 5b. After ROME redirected the client's requests, the end-to-end response times of all the clients drop below the required server delays, indicating that every server object achieved its required server delay (which is a part of the corresponding end-to-end response times). This result demonstrated that ROME can handle overload effectively and efficiently.

Concurrent Failures. We then stress-tested ROME further with concurrent failures. Since the CPU utilizations in the system have changed dynamically, FLARe's MRM also employs LAAF to redetermine the failover targets for all the primary objects in the system. The recomputed failover targets are as follows: (1) for A-1, it is $\langle$A-4, A-2$\rangle$ (2) for B-1, it is $\langle$B-2, B-3$\rangle$, and (3) for D-1, it is $\langle$D-2$\rangle$

We emulated a failure 150 seconds after the experiment started. We used a fault injection mechanism, where when clients CL-1 and CL-2 make invocations on server objects A-3 and B-1, respectively, the server objects call the *exit (1)* command, crashing the process hosting server objects A-3 on processor LAMBADA and B-1 on processor CHARLIE. The clients receive COMM_FAILURE exceptions, and then fail over to replicas chosen by the failover strategy. Using the failover targets computed by LAAF, client CL-1 fails over to A-4 while client CL-2 fails over to B-2, both of which end up starting on the same processor ALPHA, which is already hosting a primary D-1.

As a result, the CPU utilization of the processor ALPHA jumps to 80% (as highlighted by point B in Figure 5a), while the clients CL-1, CL-2, and CL-4 see an increase in response times (as shown in Figure 5b). FLARe's MRM triggers ROME once again to resolve the overload, starting with the most heavily loaded service, A-4, but clients of A-4 cannot be moved, as that would again overload the processor BETA. Hence, ROME redirects all clients of B-2 (which is the next most heavily loaded object) to its replica B-3 on processor PRINCE. As a result, the CPU utilizations of all the processors settle below 70% as shown by point (B in Figure 5a), while the end-to-end response times (and hence the server delays) drop below the required server delays.

This experiment demonstrates that ROME can effectively enforce the specified utilization bound and server delays by dynamically handling overloads caused by concurrent failures and workload changes.

4.3 Failover Delay

To empirically evaluate the failover delays under the *static* and the *adaptive* failover strategies, we ran an experiment with client CL-1 invoking 10,000 requests on server object A-1. No other processes operated in the processor hosting A-1, so that the response time will equal the execution time of the server. A fault was injected to kill the server while executing the $5,001^{st}$ request. The clients then failover to backup server objects A-2, which execute the remaining 5,000 requests (including the one experiencing the failure).

The left side of Figure 6 shows the different response times perceived by client C-1 in the presence of server object failures. The failover delays for the *static* and *adaptive* failover strategies are similar because under the static strategy the client knows the failover decision *a priori*, while under the LAAF strategy, FLARe's MRM proactively sends the updated failover targets to the client so they are also readily available when a failover occurs. Our results indicate that FLARe's proactive failover strategy achieves fast failover with a failover delay comparable to the static strategy.

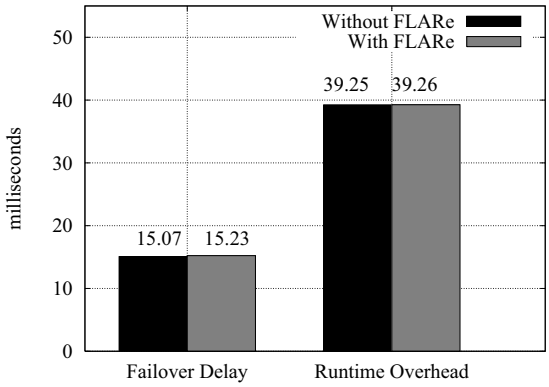


Figure 6: **Failover delay and run-time overhead**

4.4 Overhead under Fault-Free Conditions

FLARe uses a CORBA client request interceptor to catch COMM_FAILURE exceptions and transparently redirect clients to suitable failover targets. To evaluate the runtime overhead of these per-request interceptions during normal failure free conditions, we ran a simple experiment with client CL-1 making invocations on server object A-1 with and without client request interceptors.

We ran this experiment for 50,000 iterations and measured the average response time perceived by CL-1. The right side of Figure 6 shows that the average response time perceived by CL-1 increased by only 8 microseconds when using the client request interceptor. This result shows that interceptors add negligible overhead to the normal operations of an application.

5 Related Work

Fault-tolerance in non-real-time middleware. Prior research has focused on designing fault-tolerant middleware systems using CORBA [10, 3, 4]. A survey of different architectures, approaches, and strategies using CORBA-based fault-tolerance capabilities is presented in [23]. Research has also focused on non-CORBA based fault-tolerant middleware. For example, IFLOW [8] uses fault-prediction techniques to increase or decrease the frequency of backup replica state synchronizations to optimize state transfer during failure recovery. These prior middleware platforms, however, were not designed for real-time applications. In contrast, FLARe can maintain desired soft real-time performance in face of dynamic workload and failures.

Fault-tolerance in real-time systems based on active replication. Prior research has focused on developing middleware systems that provide fault-tolerance for real-time systems using ACTIVE replication. AQUA [20] dynamically adapts the number of replicas receiving a client request in an ACTIVE replication scheme so that slower replicas do not affect the response times received by clients. Eternal [19] dynamically changes the locations of active replicas by migrating soft real-time objects from heavily loaded processors to lightly loaded processors, thereby providing better response times for clients. In the past decade, research has also focused on task partitioning algorithms [16, 9, 13] that allocate tasks and their ACTIVE replicas on appropriate processors at deployment time while satisfying timing and dependability constraints. We recognize that hard real-time systems require predictable performance despite the occurence of failures, and hence require ACTIVE replication. In contrast, FLARe focuses on PASSIVE replication, which is more suitable for resource-constrained distributed soft real-time applications due to its low resource usage.

Fault-tolerance in real-time systems based on passive replication. MEAD [24] reduces fault detection and client failover time by determining the possibility of a primary replica failure using simple failure prediction mechanisms and redirects clients to alternate servers before failures occur. [28] presents a real-time primary backup replication scheme that uses scheduling algorithms such as rate monotonic scheduling algorithm for providing temporal consistency guarantees for operations as well as update transmissions. The key contributions of FLARe are its adaptive failover target selection and overload management approach for handling dynamic soft real-time applications.

Prior research has also focused on deployment-time scheduling and task partitioning algorithms that deploy tasks and their PASSIVE replicas in their appropriate processors. [5] analyzes first-fit assignments for periodic real-time tasks scheduled using rate monotonic priority assignments with both passive and active instances. To provide fault-tolerance for aperiodic tasks in multiprocessor systems, [12] introduces backup overbooking techniques that allocate multiple passive replicas to the same processor assuming that only some passive replicas must be activated at the same time. Likewise, [15] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability and supports both PASSIVE and ACTIVE replication. FLARe can benefit from such work for deploying long-running periodic tasks and their PASSIVE replicas at their most appropriate processors. The novelty of FLARe lies its capability to adapt to dynamic workloads through load-aware failover and overload management.

6 Concluding Remarks

This paper presents the Fault-tolerant Load-aware and Adaptive middlewaRe (FLARe) for distributed soft real-time applications. FLARe features (1) the Load-aware and Adaptive Failover (LAAF) strategy that adapts failover targets based on system load; (2) the Resource Overload Man-

agement Redirector (ROME) strategy that dynamically enforces CPU utilization bounds to maintain desired server delays in face of concurrent failures and load changes; and (3) an efficient fault-tolerant middleware architecture that supports transparent failover to passive replicas. FLARe has been implemented on top of the TAO RT-CORBA middleware as open-source software. Empirical evaluation on a distributed testbed demonstrates FLARe's capability to maintain system availability and soft real-time performance in the face of dynamic workload and failures while introducing only negligible run-time overhead.

Acknowledgments: This work has been supported in part by NSF CAREER Award CNS-0448554.

References

[1] J. Balasubramanian, S. Tambe, B. Dasarathy, S. Gadgil, F. Porter, A. Gokhale, and D. C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08*, pages 113–122, 2008.

[2] J. Balasubramanian, S. Tambe, A. Gokhale, C. Lu, C. Gill, and D. C. Schmidt. FLARe: a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems. Technical Report ISIS-08-812, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, October 2008.

[3] R. Baldoni and C. Marchetti. Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper.*, 33(8):767–797, 2003.

[4] T. Bennani, L. Blain, L. Courtes, J. C. F. M. O. Killijian, E. Marsden, and F. Taiani. Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization. In *Proc. of DSN. (2004)*.

[5] A. A. Bertossi, L. V. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, pages 934–945, 1999.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-backup Approach. In *Distributed systems (2nd Ed.)*, pages 199–216. 1993.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-backup Approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[8] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom. Utility-Driven Proactive Management of Availability in Enterprise-Scale Information Flows. In *Proceedings of ACM/Usenix/IFIP Middleware*, pages 382–403, 2006.

[9] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng. Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems. In *RTAS '07*, pages 249–258, 2007.

[10] R. Friedman and E. Hadad. Fts: A high-performance corba fault-tolerance service. In *Proc. of WORDS.(2002)*.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[12] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(3):272–284, 1997.

[13] S. Ghosh, R. R. Rajkumar, J. Hansen, and J. Lehoczky. Scalable resource allocation for multi-processor qos optimization. In *ICDCS '03*, page 174, Providence, RI, USA, 2003.

[14] A. S. Gokhale, B. Natarajan, D. C. Schmidt, and J. K. Cross. Towards real-time fault-tolerant corba middleware. *Cluster Computing*, 7(4):331–346, 2004.

[15] 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.

[16] S. Gopalakrishnan and M. Caccamo. Task partitioning with replication upon heterogeneous multiprocessor systems. In *RTAS '06*, pages 199–207, 2006.

[17] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, Apr. 1997.

[18] E. D. Jensen. Distributed Real-time Specification for Java. `java.sun.com/aboutJava/communityprocess/jsr/jsr\_050\_drt.html`, 2000.

[19] V. Kalogeraki, P. M. Melliar-Smith, L. E. Moser, and Y. Drougas. Resource Management Using Multiple Feedback Loops in Soft Real-time Distributed Systems. *Journal of Systems and Software*, 2007.

[20] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An Adaptive Quality of Service Aware Middleware for Replicated Services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, 2003.

[21] C. Lu, X. Wang, and C. Gill. Feedback Control Real-time Scheduling in ORB Middleware. In *Proc. of RTAS. (2003)*.

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

[23] Pascal Felber and Priya Narasimhan. Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. *Computers, IEEE Transactions on*, 54(5):497–511, May 2004.

[24] S. Pertet and P. Narasimhan. Proactive recovery in distributed corba applications. In *DSN '04*, pages 357–366, 2004.

[25] 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.

[26] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[27] 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.

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