Load-aware Adaptive Failover for Middleware Systems with Passive Replication

Jaiganesh Balasubramanian[1], Sumant Tambe[1], Chenyang Lu[2], Aniruddha Gokhale[1], Christopher Gill[2], and Douglas C. Schmidt[1]

[1] Department of EECS, Vanderbilt University, Nashville, TN 37203, USA
[2] Department of CSE, Washington University in St. Louis, MO 63130, USA

Abstract. Supporting uninterrupted services for performance-sensitive distributed applications operating in resource-constrained environments is hard. It is even harder when the operating environment is dynamic and processor or process failures and system workload changes are common. Fault-tolerant middleware for these applications must assure high service availability and satisfactory response times for clients. Although passive replication is a promising fault tolerance strategy for resource-constrained systems, conventional passive replication solutions are non-adaptive and load-agnostic, which can cause post-recovery system overloads and significantly increase response times. This paper presents *Fault-tolerant Load-aware and Adaptive middlewaRe (FLARe)*, which enhances conventional passive replication schemes in three ways. First, its client failover strategy is load-aware, *i.e.*, failover targets are selected at run-time based on current CPU utilizations to maintain satisfactory response times and alleviate CPU overload after failure recovery, and adaptive, *i.e.*, failover targets are proactively adjusted in response to failures, system load fluctuations, and resource availability. Second, its client redirection strategy handles resource overloads that stem from multiple failures and workload fluctuations. Third, FLARe enables effective dissemination of failover decisions and manages CPU utilizations transparently to clients. Empirical evaluations on a distributed testbed demonstrate how FLARe efficiently uses available system resources and maintains satisfactory response times for clients when recovering from failures.

1 Introduction

Emerging trends and challenges. Middleware, such as CORBA, J2EE, and .NET, are widely used to develop performance-sensitive distributed applications, ranging from enterprise computing applications such as online stock trading systems to soft real-time systems such as sensor data acquisition applications in supervisory control and data acquisition (SCADA) systems. Such applications operate in resource-constrained environments where system loads and resource availabilities fluctuate because of dynamic changes; new applications are added, existing applications are stopped, and processors and/or processes fail. Even when operating in such unpredictable environments, it is important to maintain both system availability and satisfactory response times for clients. For example, in a online stock trading system, stock prices from an external database

must be processed and timely updates must be posted even when dynamic load fluctuations and failures occur.

ACTIVE and PASSIVE replication [9] are two common approaches for building fault-tolerant distributed applications that provide high availability and satisfactory response times for performance-sensitive distributed applications operating in dynamic environments. In ACTIVE replication [21], 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, however, imposes high communication and processing overheads, which may not be viable in resource-constrained environments.

In PASSIVE replication [3] only one replica—called the primary—handles all client requests, and backup replicas do not incur runtime overhead, except (in stateful applications) 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 runtime overhead, PASSIVE replication is appealing for applications that cannot afford the cost of maintaining active replicas.

Although PASSIVE replication is desirable in a resource-constrained environment, it is particularly challenging to support performance-sensitive distributed applications based on PASSIVE replication. Specifically, conventional client failover solutions in PASSIVE replication are non-adaptive and load-agnostic, which can cause post-recovery system overloads and significantly increase response times for clients. Furthermore, the middleware system must dynamically handle overload conditions caused by workload fluctuations and multiple failures. Finally, we need a lightweight middleware architecture that can handle failures transparently from the applications.

Solution approach → Fault-tolerant, Load-aware and Adaptive Middleware. To address the unresolved challenges with prior work, we developed the *Fault-tolerant, Load-aware and Adaptive middlewaRe (FLARe)* which maintains service availability and satisfactory response times in dynamic environments. Specifically, this paper makes the following key contributions to developing fault-tolerant middleware systems:

• **Load-aware Adaptive Failover (LAAF) strategy,** which is an adaptive and load-aware client failover strategy that uses up-to-date CPU utilization estimates to determine client failover targets that can maintain satisfactory response times in response to dynamic workload fluctuations and processor or process failures.

• **Resource Overload Management rEdirector (ROME) strategy,** which proactively redirects clients to resolve system overloads caused by simultaneous processor failures and workload fluctuations.

• **Lightweight adaptive middleware,** which is a lightweight middleware architecture that uses interceptors [2,24] to provide transparent fault-tolerance for distributed applications using PASSIVE replication. A key feature of the FLARe architecture is its support for *push*- and *pull*-based strategies to update the client-side middleware with failover and redirection targets so clients can be transparently redirected during failures and overloads.

FLARe has been implemented within the TAO Real-time CORBA middleware (`www.dre.vanderbilt.edu/TAO`). This paper reports the results of experiments that systematically evaluate FLARe on the ISISlab testbed (`www.dre.vanderbilt.edu/ISISlab`).

The remainder of this paper is organized as follows: Section 2 describes the system and fault model; Section 3 describes the design and implementation of FLARe focusing on the LAAF and ROME strategies; Section 4 provides an extensive experimental evaluation of our system; Section 5 compares FLARe with related research; and Section 6 provides concluding remarks.

2 System and Fault Models

System model. FLARe supports performance-sensitive applications that require satisfactory response times and overload protection despite workload fluctuations and processor failures[3]. Our failover strategy assumes services are stateless, which is characteristic of many three-tier architecture applications, such as online stock quoters or sensor data acquisition applications. In three-tier architectures, clients residing in the first-tier communicate with middle-tier application servers via remote operation requests. The middle-tier application servers execute the application business logic (*e.g.*, displaying stock prices or processing sensor data) while the database server tier maintains the state required by the applications (*e.g.*, price of a stock to display or sensory data). This paper presents our solution in the context of TAO CORBA Object Request Broker (ORB) since it is optimized for performance, though our techniques can generalize to other middleware, such as J2EE and .NET.

FLARe is designed to handle *dynamic* workloads in which applications may arrive and depart anytime. For example, in an online stock quoter, a new service might be introduced when a client wants to monitor a new stock index, or in a sensor acquisition system, new sensors start sensing and sending information. Similarly, existing services might be dynamically stopped, new clients could arrive, and failures and subsequent recoveries could cause system reconfigurations. Services typically run for a long time after being deployed and clients invoke remote invocations periodically (*e.g.*, to update the price of a stock index or periodic sensing and dissemination of data). Due to these dynamic fluctuations, an offline fault-tolerance scheme that determines a static list of failover targets for a client will not account for dynamic load conditions. FLARe solves this problem using adaptive and load-aware fault-tolerance.

Fault model and replication scheme. We assume that processors and processes are fail-stop [21]. Multiple faults in multiple processors can occur, and faults in distinct processors are assumed to be independent. Considering unpredictable behavior of processes or processors after a fault is beyond the scope of this paper. We also assume that networks provide bounded communication latencies and do not fail or partition. This assumption is reasonable for many

[3] Due to its low and predictable failover latency, ACTIVE replication [11,19] is preferred for hard real-time systems, where meeting every deadline—irrespective of failures—is important. Hence, our work does not focus on hard real-time systems.

performance-sensitive systems, such as the online stock quoters or SCADA systems, where nodes are connected by highly redundant high-speed networks and network QoS can be provided by mechanisms such as Differentiated Services (DiffServ). Relaxing this assumption through integration of our middleware with network level fault tolerance techniques is an area of future work. FLARe employs PASSIVE replication [3] to provide fault-tolerance.

3 Design and Implementation of FLARe

This section describes the design and implementation of the FLARe middleware. The key design goals of FLARe include (1) masking clients from processor and process failures through transparent client failover, (2) maintaining satisfactory response times and alleviating post recovery overload through load-aware failover target selection, and (3) adapting to load fluctuations caused by dynamic workload and processor failures through overload management techniques.

3.1 Overview of FLARe Architecture

FLARe's architecture shown in Figure 1 has three main components: the *Middleware Replication Manager*, the *Client Failover Manager* for each client, and the *Monitor* on each processor hosting servers. FLARe achieves fault-tolerance

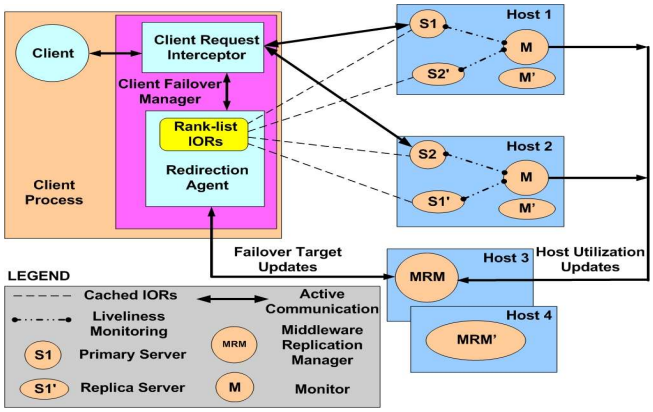


Fig. 1: **FLARe Middleware Architecture**

through PASSIVE replication of CORBA objects, where the primary and backup replicas are deployed across different processors in the distributed system.

FLARe's *middleware replication manager* provides interfaces that server objects use to provide information about (1) the processors and the specific process where their primaries and backups are hosted, (2) the CPU utilization that they will contribute to when they become a primary replica to serve client requests, 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 their backup replicas–and to make adaptive failover target decisions—FLARe's middleware replication manager uses a *monitor* on each processor to track failures and CPU utilization of all the processors hosting the primary and backup replicas of each server object.

FLARe's middleware replication manager employs the *Load-Aware and Adaptive Failover* (LAAF) target selection algorithm to prepare a list of failover targets for each *primary* object operating in the system. Multiple failover targets are prepared to handle multiple failures of the same *type* of server object. Section 3.2 describes and analyzes this algorithm in detail.

There are situations when the selected failover targets are not appropriate for clients due to sudden workload fluctuations and multiple failures. FLARe's middleware replication manager employs the *Resource Overload Management rEdirector* (ROME) algorithm to detect these overloads and system resource imbalances, determine alternate replica targets, and redirect clients transparently to those targets. Section 3.3 describes and analyzes this algorithm in detail.

FLARe's *client failover manager* comprises a *redirection agent*, which is updated with the failover and redirection targets so that clients can transparently recover from failures and overloads, respectively. To handle failures, FLARE's *client request interceptor* catches failure exceptions and modifies the exception handling behavior. Instead of propagating the exception to the client application, the client request interceptor redirects the client invocation to an appropriate failover target, provided by the redirection agent. To handle overloads, the client request interceptor alters the execution of an invocation by changing targets at the start of an invocation.

3.2 Load-aware and Adaptive Failover Target Selection

As described in Section 3.1, FLARe's replication manager collects updates from monitors about the CPU utilizations and liveness of processors/processes. Algorithm 1 depicts FLARe's *load-aware, adaptive failover (LAAF)* target selection algorithm that uses the measurements to select per-object failover targets.

The LAAF algorithm 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 utilization of all processors in the system. This algorithm is run whenever there is a change in the CPU utilization by a *threshold* (*e.g.*, $\pm$ 10) in any of the processors in the system, as FLARe needs to react to such dynamic environment changes.

The output of the LAAF target selection algorithm is a ranked list of failover targets for each primary object replica in the system. FLARe maintains an ordered list of failover targets instead of only the first one to deal with concurrent failures. 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.

The LAAF target selection algorithm 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, morever, the LAAF target selection algorithm redirects the clients of different primary objects located

on a 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 a failure recovery process.

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
```

The LAAF algorithm works as follows: For every processor in the system (line 7), the algorithm iterates through all hosted processes (line 9), and the primary replicas that are hosted in those processes (line 10). For every such 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 the middleware replication manager 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.

When the algorithm repeats the process described above for every other primary replica object hosted in the same process (Lines 10–12), the least loaded fail over processor is determined while taking into consideration 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 failover target 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 the client failovers of co-located primary objects (which are estimated by the algorithm). The computed failover target decisions are then used for redirecting a client if any failure occurs before the next time the LAAF target selection algorithm is run.

The LAAF algorithm is optimized for process failures on same processor 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, the LAAF algorithm may also result in suboptimal failover targets when process/processor failures and workload fluctuation occur concurrently, *i.e.*, before FLARe's middleware replication manager receives the updated CPU utilization from the monitors. To handle such overload situations FLARe employs the ROME strategy (described in Section 3.3) to proactively redirect clients of overloaded processors to less loaded processors.

3.3 Resource Overload Management and Redirection

FLARe's middleware replication manager employs the *Resource Overload Management and rEdirection (ROME)* algorithm to handle overloads and load imbalance in the system. FLARe allows users to specify a per-processor *overload threshold*. A processor whose CPU utilization exceeds the overload threshold is considered overloaded.

It is important to resolve processor overload as CPU saturation may cause system failure due to kernel starvation [14]. For soft real-time applications users may specify the overload threshold based on the suitable schedulable utilization bounds [23] to achieve satisfactory response times. Similarly, FLARe also allows users to specify a per-object *migration threshold* to migrate primary objects from current heavily loaded (but not overloaded) processor to the least loaded processor hosting a replica of that object. Balancing processor CPU utilization helps reduce the response times and avoid overload on a subset of processors in the system. Through the ROME algorithm, FLARe effectively handles CPU overload and load imbalance as special cases of failures for performance-sensitive applications.

In the case of failures, the clients are redirected to appropriate failover targets based on decisions made by the LAAF algorithm, 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 *loads* of objects as opposed to *objects*. Our approach is thus more efficient and less time consuming than physically moving the object itself to a lightly loaded processor. Moreover, our approach leverages existing replicas and effectively utilizes them for maintaining satisfactory response times for clients. We now describe how the ROME algorithm handles CPU overload and load imbalance, respectively.

Handling overloads. When the CPU utilization at any of the processor crosses the overload threshold, the middleware replication triggers the ROME algorithm to react to the overloads. FLARe determines the primary objects whose clients need to be redirected and their target hosts using ROME's overload management algorithm. Given an overloaded processor, *i.e.*, whose CPU utilization exceeds the overload threshold, the algorithm (shown in Algorithm 2) considers the primary objects on the processor in the 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, 12, 13, 14, and 15). The attempt fails if the least-loaded processor of the backup replicas would exceed the

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 threshold 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 processor i
21:      end if
22:   end for
23: end for
```

overload threshold if the migration occurs. The algorithm attempts migrations until (1) the processor is no longer overloaded or (2) all primary objects in the overloaded processor have been considered for migration.

Similar to the failover target selection algorithm, the ROME algorithm also uses the *expected CPU utilization* to spread the load of multiple objects on an overloaded processor to different hosts. The expected CPU utilization accounts for the load change due to the migration decisions on other objects on the same processor. After 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 minimal perturbations.

FLARe also adopts a similar approach to handle load imbalance among processors. Its middleware replication manager triggers redirection, when it detects that the difference in load between the processor hosting the current primary object, and a processor hosting one of the backup replicas is above a *migration threshold*. The middleware replication manager monitors the difference between the CPU utilization of the processor hosting the primary replica and the least loaded processor hosting the backup replica, and migrates the clients of the primary replica to the selected backup replica, if the difference is above the migration threshold. To prevent a large number of objects from migrating to the same

processor, FLARe's middleware replication manager uses the *expected utilization* of processors when making decisions based on the migration threshold.

3.4 FLARe Middleware Implementation

We now describe how different components of the FLARe middleware are implemented atop the TAO Real-time CORBA middleware.

Monitoring CPU utilization and processor failures. On Linux, FLARe's *monitor* process uses the `/proc/stat` file to estimate the CPU utilization (the fraction of time when the CPU is not idle) in each sampling period. The monitors can be configured to periodically sample and report the CPU utilization of a processor or can be queried *on-demand* by the middleware replication manager, particularly in the cases where there are overloads in the system.

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 respond to the middleware replication manager within a timeout period, the middleware replication manager suspects that the processor has crashed.

Middleware replication manager. The middleware replication manager is designed using the *Active Object* pattern [22] to decouple the reporting of a load change or a failure from the process. This decoupling allows several monitors to register with the middleware replication manager while allowing synchronized access to its internal data structures. Moreover, it is strategized with the LAAF and ROME algorithms. The middleware replication manager is replicated using SEMI_ACTIVE replication [8] (provided by 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* [24] and a redirection agent, which together coordinate to handle failures transparently from client application logic. Whenever a primary fails, the interceptor catches the CORBA COMM_FAILURE exception. At this point in time, 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.

A rank list is necessary to determine the failover target. The redirection agent is therefore a CORBA object that runs in a separate thread from the interceptor thread. Since portable interceptors are not remotely invocable objects, it was not feasible for an external entity (such as a middleware replication manager) to send the rank list information to the interceptor.

The rank list can be propagate to the redirection agent via one of two approaches supported by FLARe. The first approach is called *proactive* propagation, where each redirection agent registers with a middleware replication manager that is responsible for *pushing* the newly computed ranked lists to the

redirection agents. An advantage of this scheme is that the redirection agent need not be concerned with the failure and recovery semantics of the replication manager. The second approach is called *reactive* propagation. In this approach the interceptor consults the redirection agent for a ranked list when a failure occurs. The redirection agent in turn *pulls* a ranked list from the middleware replication manager.

Client redirection during overloads. To redirect clients during overloads, FLARe requires that each server object know whether they are a primary or a backup. Each server object implements a *not_primary(bool)* operation. During overloads, the middleware replication manager uses this operation to inform the server object if their *primary* status has changed. The primary replica starts rejecting client invocations after completing currently pending invocation if there is one by throwing a LOCATION_FORWARD exception back to the clients. FLARe's client request interceptor catches the exception and queries the redirection agent for the next target. The redirection agent redirects clients to chosen backups during overloads using the same failure management framework described above.

4 Empirical Evaluation of FLARe

The experiments were conducted at ISISlab (`www.dre.vanderbilt.edu/ISISlab`) on a testbed of 14 blades. Each blade has two 2.8 GHz CPUs, 1GB memory, a 40 GB disk, and runs the Fedora Core 4 Linux distribution. Our experiments used one CPU per blade and the blades were connected via a CISCO 3750G switch into a 1 Gbps LAN. 12 of the 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 2 blades. Monitors, along with their replicas, are deployed on each of the server hosting processors, and client failover managers (including the client request interceptor and the redirection agent) are hosted in each of the processors hosting the clients.

To emulate dynamic soft real-time applications, such as distributed sensor data acquisition systems, 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 presented in this paper, client applications invoked operations on server objects using one of the following rates: 10 HZ, 5 HZ, 2 HZ, or 1 HZ.

We measured the CPU utilization and 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. The clients also experience additional delays for fault-detection and failover after failures. 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.

4.1 Evaluating LAAF

This experiment evaluates how FLARe's LAAF strategies select failover targets based on current CPU utilizations, maintains satisfactory response times for clients, and alleviates processor overloads. We compared FLARe's proactive and reactive load-aware client failover strategy (Section 3.4) 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. In contrast, using the LAAF algorithm, FLARe's proactive and reactive load-aware client failover strategies recompute failover targets whenever the CPU utilizations in the system change.

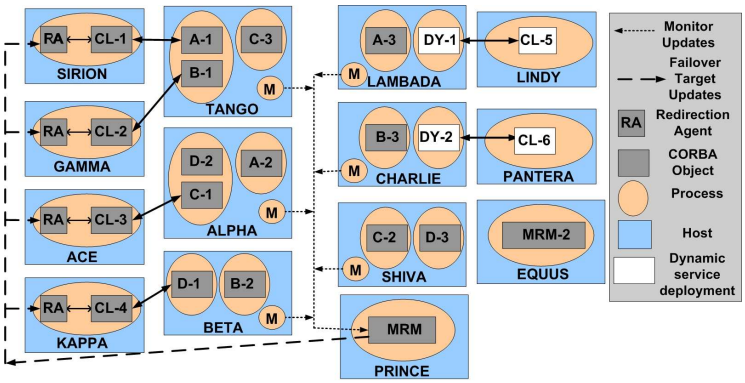


Fig. 2: **Load-aware Failover Experiment Setup**

Experiment setup. Figure 2 and Table 1 illustrate our experimental setup to evaluate LAAF. To evaluate FLARe in the presence of resource contention created by *dynamic workload changes*, such as dynamic service deployments, at 50 seconds after the experiment was started, we introduced dynamic deployment of two server objects DY-1 and DY-2, and their client objects, CL-5, and CL-6, respectively. 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 chosen backup server objects.

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; if B-1 fails, contact B-3 followed by B-2. We would like to note that at *deployment time*, this is an *optimal* failover strategy for the clients. With our LAAF-driven proactive and reactive load-aware failover strategies, those failover decisions are updated dynamically when and if failures occur, as the processors' utilization levels and sets of live processes change.

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%

(a)

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
Dynamic Loads			
CL-5	DY-1	5	50%
CL-6	DY-2	10	50%

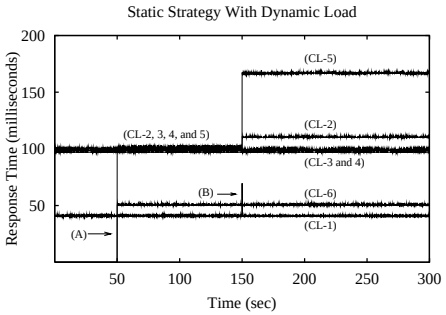
(b)

Table 1: **Experiment setup for LAAF**

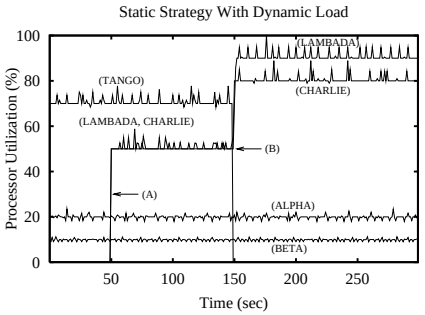
Failure scenario. To evaluate the performance of the different failover strategies, we emulated a failure 150 seconds after the experiment started. We used a fault injection mechanism, where when clients CL-1 or CL-2 make invocations on server objects A-1 or B-1 respectively, the server object calls 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.

Analysis of results. Figure 3a shows the end-to-end response times perceived by all the clients, and Figure 3b shows the CPU utilizations at all processors, both when clients used the static client failover strategy. At 50 seconds, servers DY-1 and DY-2 were dynamically deployed to the system on the processors LAMBADA and CHARLIE. As shown and highlighted by label A in Figure 3b, at 50 seconds, the CPU utilizations at processors LAMBADA and CHARLIE increases from 0% to 50%. As shown and highlighted by the label B in Figure 3a, at 150 seconds when A-1 fails (along with the process in which it is hosted, thereby failing B-1 as well) at processor TANGO, client CL-1 experiences an increase of 17.2 milliseconds in its end-to-end response time, which is the combined FAULT_DETECTION_DELAY and FAILOVER_DELAY. After the failure at 150 seconds, 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 indicated by the label B in Figure 3b, after the failover at 150 seconds, the CPU utilizations at processors LAMBADA and CHARLIE increase to 90% and 80% respectively. As a result of the overload, the response times of clients CL-5 and CL-2 increased (significantly for CL-5) after 150 seconds (as shown in Figure 3a), because their servers had the lowest priorities on LAMBADA and CHARLIE respectively.

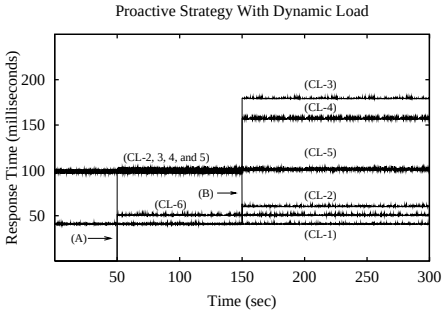
Figure 3c shows the response times perceived by all the clients and Figure 3d shows the CPU utilizations at all the processors when clients were configured to use the proactive client failover strategy. The middleware replication manager triggers the LAAF algorithm to recompute the failover targets for all the server objects in response to change in CPU utilizations in the system. At 150 seconds, client failovers cause CL-1 to invoke remote operations on A-2 and CL-2 to invoke remote operations on B-2. This is because, the LAAF algorithm changed the failover targets from A-3 to A-2 and from B-3 to B-2 because of the load changes in the processors LAMBADA and CHARLIE respectively. As highlighted by the label B in Figure 3d, none of the processor utilizations are greater than 60% after the failover of clients CL-1 and CL-2. This shows that the LAAF algorithm



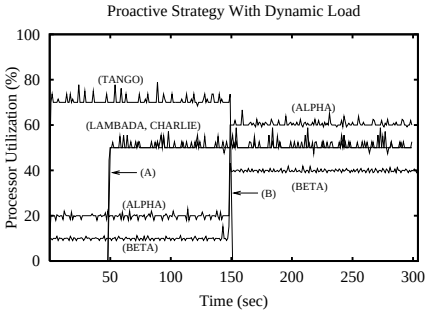
(a) Response times with static strategy



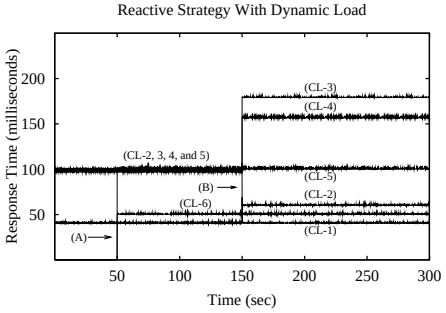
(b) Utilization with static strategy



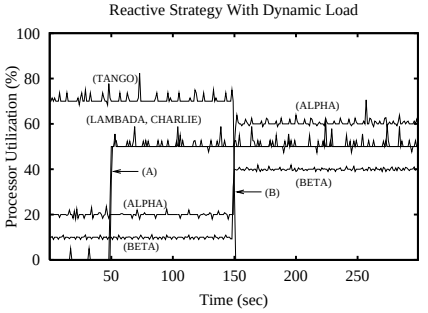
(c) Response times with proactive strategy



(d) Utilization with proactive strategy



(e) Response times with reactive strategy



(f) Utilization with reactive strategy

Fig. 3: **End-to-end response times and utilizations with different failover strategies**

makes failover target decisions in a load-aware manner and alleviates processor overloads. Consequently, as highlighted by label B in Figure 3c, the response times of CL-2 and CL-5 did not increase after the failover of clients CL-1 and CL-2, which was the case when the clients used the static client failover strategy.

Figure 3e shows the response times perceived by all the clients and Figure 3f shows the CPU utilizations at all the processors when clients were configured to use the reactive client failover strategy. From the figures, it is clear that the

performance of the clients and the processor utilizations are similar to those received by the clients when they were configured to use the proactive client failover strategy. This is not surprising as the proactive and reactive strategies both use the same LAAF algorithm to decide failover targets. However, the reactive client failover strategy incurs higher failover delay as the redirection agent communicates with the middleware replication manager to receive the failover target list after a failure exception is intercepted (see Section 4.3 for a detailed study on failover delays)

4.2 Evaluating ROME

This experiment evaluates the effectiveness of ROME in handling different overload scenarios. We stress-test ROME under overload and load imbalance caused by dynamic workload changes and multiple failures. In our experiment, we set 70% as the maximum load allowed on any node beyond which we treat it as an overload condition. We set the *migration threshold* of each object to be 120% of its utilization.

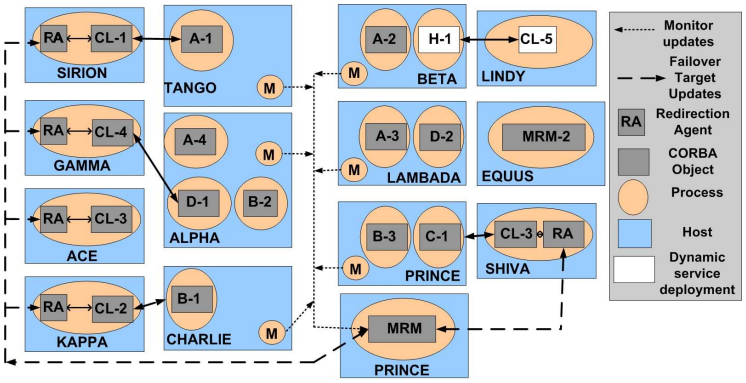


Fig. 4: **Overload Redirection Experiment Setup**

Experiment setup. Figure 4 and Table 2 illustrate the experimental setup for ROME. For the service objects that are started at deployment time, the LAAF algorithm can compute the rank lists for all the clients using the CPU utilizations known prior to the dynamic deployment of service H-1. The failover target lists are as follows: (1) for A-1, it is $\langle$A-2, A-3, A-4$\rangle$ (2) for B-1, it is $\langle$B-2, B-3$\rangle$ (3) for C-1, no replicas of C-1 are deployed in the system (4) for D-1, it is $\langle$D-2$\rangle$ (5) for the dynamically deployed service H-1, no replicas of H-1 are deployed in the system The experiment ran for 300 seconds with different overload and dynamic reconfigurations in the system over the span of the experiment.

1. Overload due to dynamic deployment of services. We emulated a failure of process A-1 on processor TANGO 50 seconds after the experiment started. 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 happens to be A-2. At the same time a new service H-1 is deployed dynamically on processor BETA and a client CL-5 starts making requests on this new service.

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	30%

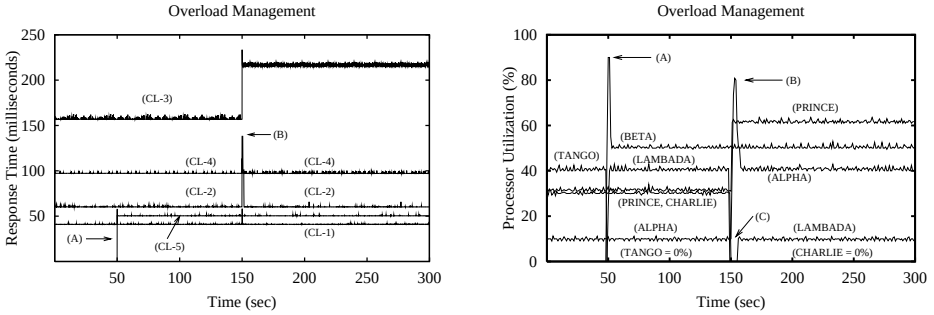
(a)

Client Object	Server Object	Invocation Rate (Hz)	Server Object Utilization
CL-4	D-1	1	10%
Dynamic Loads			
CL-5	H-1	10	50%

(b)

Table 2: **Experiment setup for ROME**

As a result of the concurrent failure and workload change, the load on the processor BETA rises to 90% (shown by point A in the Figure 5b) and the ROME algorithm is triggered. The ROME algorithm 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 the *type* A. Notice how around 51 seconds, the utilization on processor BETA goes down to 50% since the clients of A-2 were redirected while the utilization of processor LAMBADA increases to 40% due to A-3 becoming the primary. At this stage, the CPU utilizations of all processors were below 70%, while all clients receive satisfactory response times. This result demonstrated that ROME can handle overload effectively and efficiently.



(a) Client response times with overload management (b) Utilizations with overload management

Fig. 5: **Evaluation of ROME**

2. Overload due to multiple failures. We let the experiment continue and stress-tested ROME further with concurrent failures. Since the CPU utilizations in the system have changed dynamically, the middleware replication also employs the LAAF algorithm 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 ⟨A-4, A-2⟩ (2) for B-1, it is ⟨B-2, B-3⟩, and (3) for D-1, it is ⟨D-2⟩

At time t = 150 seconds, the processes on processors LAMBADA and CHARLIE fail. Clients CL-1 and CL-2 must now failover. Using the rank lists, 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. Moreover, another primary D-1 is already running on ALPHA taking its utilization to 80%. This is shown by point B in Figure 5b. Similarly the clients CL-1, CL-2 and CL-4 see an increase in response times. The

ROME algorithm is triggered once again to resolve the overload. It starts with the heaviest service, which is 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 heaviest 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 5b.

3. Load imbalance. After redirecting clients of B-2 to B-3, the middleware replication manager notes that the *migration threshold* for object D-1 is reached. This is because, D-1 is operating in processor ALPHA whose CPU utilization is 50 %, while D-2 is operating in processor LAMBADA whose CPU utilization is 0. The difference in CPU utilization between the processors hosting the primary replica D-1 and backup replica D-2 is greater than 120% of the primary object D-1's load (which is 10%). ROME detects this imbalance and redirects clients of D-1 to D-2 to take advantage of a less utilized processor. As a result, the difference in CPU utilization between the processors hosting the *new* primary replica D-2 and backup replica D-1 has reduced from 50% to -30%. Note that ROME will not thrash because the processor hosting the current primary replica now has lower utilization than that hosting its backup replica. ROME therefore effectively alleviates the load imbalance in the system through lightweight migration.

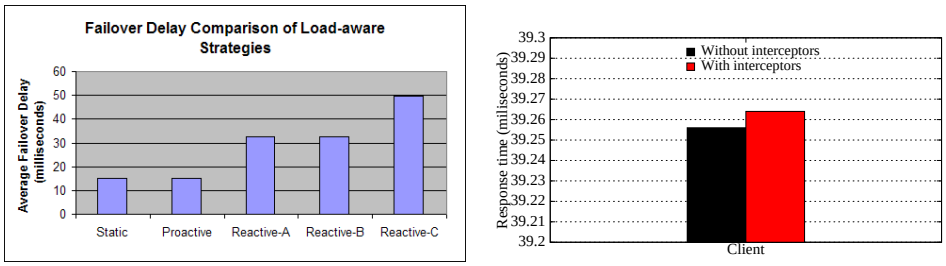
4.3 Failover Delay

The reactive load-aware strategy incurs higher failover delays than the proactive load-aware and static failover strategies. The higher failover delay stems from the need for the client redirection agent to perform a remote invocation of FLARe's middleware replication manager to get the failover target list after receiving the COMM_FAILURE exception due to a server object failure. To evaluate the failover delays under different failover strategies empirically, we ran a set of experiments 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 $5,001^{st}$ request. The clients then failover to backup server objects A-2, which execute the remaining 5,000 requests (including the one experiencing the failure).

Figure 6a shows the different response times perceived by client C-1 in the presence of server object failures. The failover delays for the static and proactive load-aware strategies are similar because both strategies know the failover decision *a priori* and just use the next available address. In the reactive load-aware strategy, however, the client redirection agent must invoke the middleware replication manager to acquire the next failover target, which result in a 16 ms of increase in failover delay (see REACTIVE-A in Figure 6a).

We repeated the experiment under two more scenarios: (1) *multiple clients*: one more client C-2 invokes remote operations on the server A-1, and when the server A-1 fails, redirection agents of both the clients C-1 and C-2 contacted the middleware replication manager for the failover targets, and (2) *middleware replication manager failure*: when the clients contacted the middleware replication manager, the replication manager failed. Note, neither scenario changes

the failover delays under the static strategy and proactive strategy because they do not require clients invoke the middleware replication manager. As shown by REACTIVE-B in the Figure 6a, the failover delay does not change significantly when multiple clients contact the middleware replication manager to obtain the failover targets. This is because the middleware replication manager does not compute the failover target list when the redirection agent initiates the communication; instead it provides the redirection agent with the list it already has in its cache.



(a) Failover delay (b) Runtime overhead

Fig. 6: **Failover delay and Runtime overhead of Failover Strategies**

Failure of the middleware replication manager increases the failover delay as shown by REACTIVE-C in the Figure 6a. This is because of the failover required for the redirection agent to be redirected to the backup replication manager to obtain the failover target lists. This delay will increase further with more failures of the replication manager. Overall, our results indicate that the proactive failover strategy achieves fast failover comparable to the static strategy. The reactive failover strategy incurs higher failover delay, especially when the failover coincide with the failure of the replication manager. System users therefore need to consider the tradeoff between the lower failover delay of the proactive strategy and the simplicity of the reactive strategy based on their application requirements.

4.4 Overhead under Fault-Free Conditions

FLARe uses a CORBA client request interceptor to (1) catch COMM_FAILURE exceptions and transparently redirect clients to suitable failover targets, and (2) catch LOCATION_FORWARD exceptions to transparently redirect clients during overloads. 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.

We ran this experiment for 50,000 iterations and measured the average response time perceived by CL-1. Figure 6b 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

CORBA-based fault-tolerant middleware systems. Prior research has focused on designing fault-tolerant middleware systems using CORBA. A survey of the different architectures, approaches, and strategies using which fault-tolerance capabilities can be provided to CORBA-based distributed applications is presented in [17]. [2] describes a CORBA portable interceptor-based fault-tolerant distributed system using passive replication and extends the interceptors to redirect clients with a static client failover strategy. MEAD [16], FTS [6] and IRL [1] use CORBA portable interceptors to provide fault-tolerance for CORBA-based systems using active replication. The key novel features of FLARe that distinguishes it from these systems are the adaptive load-aware failover target selection and overload management capabilities based on a passive replication approach.
Adaptive passive replication systems. Prior research has focused on adaptive passive replication to reduce delays incurred by conventional passive replication during fault detection, client failover, and fault recovery. For example, IFLOW [4] uses fault-prediction techniques to change the frequency of backup replica state synchronizations to minimize state synchronization during failure recovery. Similarly, MEAD [18] 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. Other research [12] uses simulation models to analyze multiple checkpointing intervals and their effects on fault recovery in fault-tolerant distributed systems. [7] focuses on an adaptive dependability approach by mediating interactions between middleware and applications to resolve constraint consistencies while improving availability of distributed systems. FLARe focuses on adaptive failover strategies to alleviate overload after recovering from a failure , which is particularly important to performance-sensitive applications and has not been addressed by previous research. The LAAF and ROME strategies of FLARe are therefore complimentary to the aforementioned adaptive fault tolerance techniques.
Load-aware adaptations of fault-tolerance configurations. Prior research has focused on run-time adaptations of fault-tolerance configurations [5]. For example, the DARX framework [15] provides fault-tolerance for multi-agent software platforms by focusing on dynamic adaptations of replication schemes as well as replication degree. AQUA [13] 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 [11] 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. Our work on FLARe differs from earlier work by focusing on dynamic adaptations of failover targets in a *passive* replication scheme, so that the middleware can provide both high availability and satisfactory performance for distributed applications operating in resource-constrained environments.

Quality of Service using generic interception frameworks. Other projects have focused on interceptions above the middleware layer to add quality of service (QoS) for applications. For example, QuO [25] weaves in QoS aspects into applications at compile time by wrapping application stubs and skeletons with specialized delegates that can be used for intercepting application requests and replies. The ACT project [20] provides response to unanticipated behavior in applications by weaving adaptive code into ORBs at runtime and provides fine-grained adaptations by intercepting application requests and replies. CQoS [10] provides platform-dependent interceptors based on stubs and skeletons, and QoS-specific service components, that work with the interceptors to add QoS like fault-tolerance to applications. Although, FLARe is based on CORBA portable interceptors, its LAAF and ROME strategies could be used in conjunction with any of the generic interceptors described in these research to provide fault-tolerance and performance management for distributed applications.

6 Concluding Remarks

This paper described the *Fault tolerant Load-aware and Adaptive middlewaRe (FLARe)* system specifically designed for performance-sensitive and dynamic distributed applications. FLARe achieves these objectives through three novel contributions. First, its *Load-aware and Adaptive Failover (LAAF)* algorithm determines the appropriate failover targets for different services by considering dynamic loads on different CPU resources of the system. Second, the *Resource Overload Management Redirector (ROME)* algorithm manages CPU resource overloads that may be caused due to multiple overloads by a lightweight migration scheme for heavily loaded services, and transparent redirection of clients. Finally, a lightweight middleware architecture provides a low overhead solution to realize FLARe's capabilities. In the future, we plan to extend the FLARe architecture to support stateful services by integrating state synchronization with a resource management framework.

References

1. R. Baldoni and C. Marchetti. Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper.*, 33(8):767–797, 2003.
2. 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)*.
3. 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.
4. 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.
5. T. Dumitras and P. Narasimhan. Fault-Tolerant Middleware and the Magical 1%. *In: Proc. of Middleware (2005)*, 2005.
6. R. Friedman and E. Hadad. Fts: A high-performance corba fault-tolerance service. In *Proc. of WORDS.(2002)*.

7. L. Froihofer, K. M. Goeschka, and J. Osrael. Middleware support for adaptive dependability. In *Middleware*, pages 308–327, 2007.
8. 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.
9. R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, Apr. 1997.
10. J. He, M. A. Hiltunen, M. Rajagopalan, and R. D. Schlichting. Providing qos customization in distributed object systems. In *Middleware*, pages 351–372, 2001.
11. 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.
12. P. Katsaros and C. Lazos. Optimal object state transfer - recovery policies for fault tolerant distributed systems. In *Proc. of DSN. (2004)*.
13. 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.
14. C. Lu, X. Wang, and C. Gill. Feedback Control Real-time Scheduling in ORB Middleware. In *Proc. of RTAS. (2003)*.
15. O. Marin, M. Bertier, and P. Sens. Darx: A framework for the fault-tolerant support of agent software. In *Proc. of ISSRE. (2003)*.
16. P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava. MEAD: Support for Real-time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.
17. 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.
18. S. Pertet and P. Narasimhan. Proactive recovery in distributed corba applications. In *Proc. of DSN. (2004)*.
19. Y. Ren, D. Bakken, T. Courtney, M. Cukier, D. Karr, P. Rubel, C. Sabnis, W. Sanders, R. Schantz, and M. Seri. AQuA: an adaptive architecture that provides dependable distributed objects. *Computers, IEEE Transactions on*, 52(1):31–50, 2003.
20. S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. In *Proc. of ICDCS. (2004)*.
21. 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.
22. 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.
23. L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, 2004.
24. 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.
25. J. A. Zinky, D. E. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.