Load Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers

Roy Friedman*
Department of Computer Science
Cornell University
Ithaca, NY 14853
roy@cs.cornell.edu

Daniel Mosse†
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA
mosse@cs.pitt.edu

December 8, 1996

Abstract

Clusters of workstations, connected by a fast network, are emerging as a viable architecture for building high-throughput fault-tolerant servers. This type of architecture is more scaleable and more cost-effective than a tightly coupled multiprocessor and may achieve as good a throughput. Two of the most important issues that a designer of such clustered servers must consider in order for the system to meet its fault-tolerance and throughput goals are the load balancing scheme and the fault-tolerance scheme that the system will use.

This paper explores several combinations of such fault tolerance and load-balancing schemes, and compare their impact on the maximum throughout achievable by the system, and on its survivability. In particular, we show that a fault-tolerance scheme may have an effect on the throughput of the system, while a load-balancing scheme may affect the ability of the system to override failures. We study the scaleability of the different schemes under different loads and failure conditions.

The validation of our schemes is done using data taken from emulations of an intelligent networking coprocessor of a telephone switch, which follows, for example, the SS7 signaling protocol.

*Supported by ARPA/ONR grant N00014-92-J-1866
†Supported by NSF under RIA grant CCR-9308886, and under contract DABT63-96-C-0044.

1 Introduction

Recent de-regulation and increased competition in various industries has increased the awareness of service providers of the need to provide highly reliable services to their customers. At the same time, in order to satisfy the ever growing demand by consumers, such services must be flexible and costumizeable at low cost. In particular, to keep costs down, servers upon which providers rely on should be able to handle very high rates of requests, and be able to scale up beyond their initial installations.

Intelligent Networking (IN) coprocessors (or SCP in the SS7 terminology) are an important example of such servers. Modern telecommunication networks split the task of routing a call between the switch and an IN coprocessors: a switch in these networks handles fairly simple predefined set of tasks, e.g., routing a regular call. When it encounters more complex tasks, e.g., figuring out where a 1-800 call should be routed to, it hands the call to the IN coprocessors, and waits for instructions from it. This allows to add services, modify existing services, and customize services relatively cheaply, since such changes only need to occur in the IN coprocessors software, while the switch itself, which is implemented in hardware, is left untouched.

In order to provide the required levels of reliability and availability, many service providers build their servers on top of fault-tolerant computers, such as those built by BIIN [14], Tandem [3], and Stratus. These typically function as self-checking and replicated hardware, which execute instructions in lock-step, thus reducing the possible throughput of the system by at least 50%.

Replicated servers present a low-cost alternative to fault-tolerant computers for applications which require high throughput and high availability. These servers are loosely coupled and can incorporate different types of redundancy (hardware redundancy is a must if the system is to tolerate permanent faults). Therefore, replicated servers can decrease the overhead of providing redundancy in order to tolerate different types of faults if compared to straight hardware redundancy (active replication). This can be done by exploiting the fault model and applying the correct type of solution for a specific problem.

Replicated servers also simplify the task of doing on-the-fly software and hardware upgrades, an extremely attractive feature in order to enhance the competitiveness, maintainability and availability of the system. It is possible to bring down one node at a time while the other nodes automatically handle the extra load in a loosely-coupled fashion. Each node taken off line is then loaded with the new software, and re-incorporated into the running system. This process is repeated until all nodes are updated. Assuming that the upgraded software is backward compatible, this can be done without any interruption to service.

Finally, as indicated in [2, 7], most software faults are transient and correlated. Many times, their occurrence depends heavily on the exact order in which events occur. In particular, the demand for rapid enhancement and modifications of services increases the probability of having such software bugs in servers. Tightly coupled fault-tolerant computers, in which all components see the same set of events in exactly the same order, are vulnerable to total failures caused by such transient and correlated faults. Distributed solutions, in which different nodes see slightly different sets of events and/or in a different order, are less likely to suffer total failures due to such faults.

In order for distributed replicated servers to take a role as alternatives to fault-tolerant computers, they must be able to respond within the required deadlines under sustained high rate of requests, and to mask a failure of at least one component of the system. An important factor in achieving high-throughput is maintaining the load of the various compute nodes as balanced as possible, especially if the service time for different requests has large variance. Furthermore, the fault tolerant scheme employed by the server not only affects the failure scenarios that can be masked by the system, but also affects the achievable throughput. This is because the overhead imposed by the fault tolerant scheme in order to mask failures could be significant.

In this paper we investigate and evaluate several load balancing schemes and fault-tolerance schemes that can be used in a distributed fault-tolerant server, in terms of their effect on the degree of fault-tolerance and throughput. Our motivating application is a telecommunication switch, which receives a steady influx of requests, as described in [5]. These are requests for lookup and redirect telephone calls being established. However, the architecture that we use is applicable to many kinds of servers that require similar fault-tolerant and high-throughput characteristics, e.g., air-traffic control systems, submarine systems, and rail-road controllers.

An interesting result of our simulations is that load balancing schemes can also serve as natural failure detectors; whenever a node fails, since it stops responding to requests, other nodes perceive it as overloaded, and therefore refrain from sending more requests to that node. Similarly, if a node becomes very busy, either due to some transient failure or "bad luck" (the node was assigned much work), the rest of the system will refrain from submitting new requests to that node. This will give the temporarily overloaded node a chance to recover, and therefore reduce the probability of a node failure due to transient overloads. Finally, we have simulated two load balancing schemes that take into consideration the actual load on the compute nodes in the system. One of these schemes only approximated that load, and was therefore practical, while the other scheme has full and exact knowledge of the load, and was used as a best case (but impossible to implement). An encouraging result of our simulations was that a load balancing scheme which only approximated the load performed almost as well as the full knowledge scheme.

The rest of this paper is organized as follows: We compare this work with related work in Section 2. The general architecture of a replicated fault-tolerant server is described in section 3. The various load balancing schemes that we used are described in Section 4 and the simulation results in Section 5. Finally, we conclude with a discussion in Section 6.

2 Related Work

Permanent faults can only be tolerated by using hardware redundancy since if one component fails, another must replace it. The three main kinds of hardware redundancy include passive, active and hybrid [8]. In passive redundancy, the faults are masked in order to prevent errors. *Fault masking* is a technique in which multiple outputs are generated using hardware modules, and voting is conducted to hide faulty outputs. In active redundancy, faults are detected and tolerated by using redundancy already present in the system. Then the faulty hardware is removed from the system. Finally, hybrid redundancy uses a combination of these two approaches.

One of the popular techniques of tolerating permanent faults is the use of *triple modular redundancy* (TMR). In TMR, the hardware is triplicated and a majority voting is performed to determine the output of the system. Even if one of the hardware modules is faulty, the other two modules generate the correct output, and the system continues to generate the correct output. An extension of TMR is NMR in which N modules are used instead of three. Both TMR and NMR are passive redundancy techniques because they mask faults while trying to detect which processor is faulty.

Some active redundancy techniques use *spare* processors, *pair and a spare*, and *watchdog timers* [8]. If the results do not match, an error has occurred. Spare processors are used to replace faulty processors once a fault is detected. In pair and a spare, two hardware modules are used to generated outputs which are compared; if there is a result mismatch, spare processors are used to re-execute the task. A survey of various types of *watchdog processors* for concurrent error detection is presented in [11]. A watchdog processor monitors the behavior of a system to detect errors. It is small and simple coprocessor and requires less hardware as compared to replication. It can be used to detect both transient and permanent faults.

The general approach of scheduling two copies of each task in the system is called the *primary/backup* approach, and the two copies are called the *primary* and the *backup*. The *primary/backup* (PB) approach allows multiple copies of a task to be scheduled on different processors [15]. One or more of these copies can be run to make sure that the task completes before its deadline. In the PB approach, the backup task is activated only if a fault occurs while executing the primary task. Note that this scheme of adding time redundancy also allows different processors to execute the task as primary and backup.

Friedman and Birman presented a prototype implementation of a distributed fault-tolerant IN coprocessor in [5]. Their work however assumed that the service time of different requests is roughly the same, and have implemented only what we call in this work the BASIC scheme for fault-tolerant and the round-robin approach to load balancing. This work extends [5] in studying more schemes for fault-tolerance and more approaches for load-balancing. Also, a group communication system [16] was used in [5] to reconfigure the group after failures, while we ignore this overhead due to the limited functionality. Lastly, we are using a simulator, and they executed their software on a cluster of workstations.

Some works have incorporated the PB scheme into the scheduling phase (pessimistic approach to fault-tolerance guarantees). When a new task is scheduled, certain conditions are tested to ensure that faulty tasks can be re-executed to complete within their deadlines. When a fault is detected, a *recovery scheme* is needed to determine the steps to complete all tasks within their deadlines. The recovery scheme used in this work is simply the re-execution of faulty tasks.

As a special case of the PB approach, a fault tolerant scheduling algorithm for periodic tasks is proposed in [10] to handle transient faults in a uniprocessor system. One of the restrictions of this approach is that the period of any task should be a multiple of the period of its preceding tasks.

Son and Oh describe a PB scheduling strategy for periodic tasks on multiprocessor systems [12, 13]. In this strategy, a backup schedule is created for each task in the primary schedule. The tasks are then rotated such that the primary and backup schedules are on different processors and do not overlap. In [13] the number of processors required to provide a schedule to tolerate a single failure is double the number of the non-fault-tolerant schedule.

Two other works have studied fault-tolerant PB scheduling [1, 9]. In [1], there is a description of a primary/standby approach, where the standby has execution time smaller than the primary tasks (as in [10]). Both primary and standby start execution simultaneously and if a fault affects the primary, the standby will send its results. On the other hand, [9] presents theoretical results assuming that an optimal schedule exists and enhancing that schedule with the addition of standby tasks. Not all schedules permit such additions.

In [6], a general approach was presented in which application's software modules (tasks) are replicated across processors in a way that guarantees that all the tasks are executed within their deadlines despite some fixed, user-specified number of faults. The novel ideas in that approach are the combined use of *weighed backup overloading* and *backup de-allocation*. Also presented was a Markov model to analyze the schedulability when tasks have unit execution time and, for more general tasks, an evaluation of the scheme by a suite of simulated tests. Our scheme is based on the primary/backup technique and proves to be best suited for low failure rates and for applications that have large laxity.

3 General Architecture and Fault-Tolerance Schemes

A key requirement in every fault-tolerant system is that a failure of a single component should not disrupt normal service to the outside world. The architecture used in [5] for implementing a distributed fault-tolerant IN coprocessor is based on this concept. In this work, we modify that architecture, as illustrated in Figure 1.

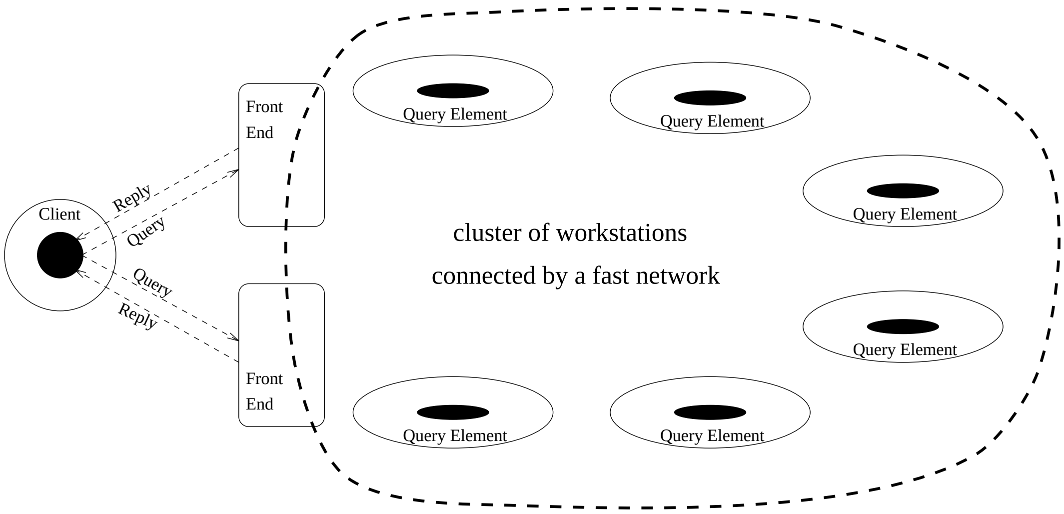


Figure 1: Architecture of a scalable server

We assume that the replicated server has a number of nodes (two in this paper) that serve as *front ends* (FEs); the FEs receive requests from clients of the system, relay these requests to the computing nodes, called *query elements* (QEs), get the reply from the QEs, and then forward the reply to the client that initiated the request. In order to guarantee no single point of failure after the receipt of a request, we assume that (a) requests are sent by the clients to both FEs, (b) QEs send their replies to both FEs, and (c) each reply is forwarded by both FEs to the client. We do not consider client failures in this paper.

Below we outline five possible schemes for governing the transfer of requests from FEs to QEs and for collecting replies of QEs by FEs; all these schemes are able to tolerate a failure of at least a single node, and some of them can deal with slightly more complex failure scenarios. In all these schemes we assume that all requests carry a unique identifier, the request number issued by the client, which is, for simplicity of the discussion, an integer. Hereafter, an *even request* is a request whose identifier is an even number, and an *odd request* is a request whose identifier is an odd number. Denote the two FEs by FE_0 and FE_1. Our software architecture is such that we assign FE_0 to be the *primary* FE for all even requests, and the *backup* FE for all odd requests, while FE_1 is assigned to be the primary FE for all odd requests, and the backup FE for all even requests. The meaning of being a primary FE (FE_P) or a backup FE (FE_B) is explained shortly, and depends on the specific fault-tolerance scheme. Formally, FE_i is the primary FE for request j such that $j \ mod \ 2 = i$. If there are k FEs, this would be generalized to $j \ mod \ k = i$.

In the schemes described here, each FE will pick a number of QEs to process the request. We define $QE(f, m)$ as the m^{th} QE picked by a certain FE f. A usage example for such notation is $QE(FE_P, 1)$ which is the first QE picked by the primary FE.

Also, regardless of the fault-tolerance scheme that is used, in order to overcome overloads, each of the fault-tolerance schemes can be tested with or without *dropping messages*. That is, when the dropping option is set, each FE keeps track of how many requests are *pending* at the system at any given moment, to the best of its knowledge. (We say that a request is pending if it was issued to a QE, but no reply has been received for it.) Whenever the number of pending messages at a FE becomes larger than some *high watermark*, that FE starts dropping new requests as soon as they arrive from the client, without even trying to service them, until the number of pending requests becomes lower than the *lower watermark*.

As explained later, in Section 4, it is possible that for some requests that were issued to QEs a reply will never be generated. In order not to count such a request forever as pending, we need to remove such requests from the assessment regarding the number of pending requests in the system. We do this by designating a garbage collection event to occur at an FE G units of time after a request was issued by that FE; if a reply for this request was not received by the time the garbage collector event is serviced, the FE decrements by one its assessment for the number of pending requests.

In comparing the various fault-tolerance schemes, we also need to derive for them the amount of overhead and benefits each scheme will impose on the system. For that, a metric will be the *replication degree*, which is the maximum number of QEs that **may** be needed to process a request. In addition, we need to determine the number of messages exchanged, and the latencies of the schemes.

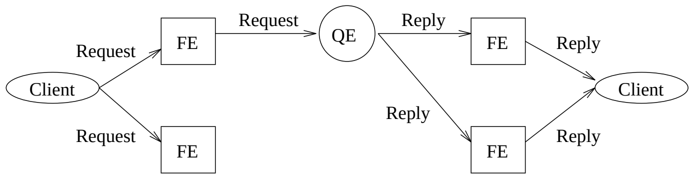
Note that are numerous possibilities for fault-tolerance schemes, and naturally we could not check all of them. Instead, we picked a few that are simple enough to be actually implemented in a real server, and are also representative in their characteristics. (A complex protocol is less likely to be used in practical settings for two reasons: (a) Most implementors would be rebuffed by the sheer complexity of the protocol, and (b) coding a complex protocol is more error prone than a simple protocol, so even if the complex protocol seems to provide better fault-tolerance, most implementors would outcast it.)

Pure Primary/Backup: BASIC

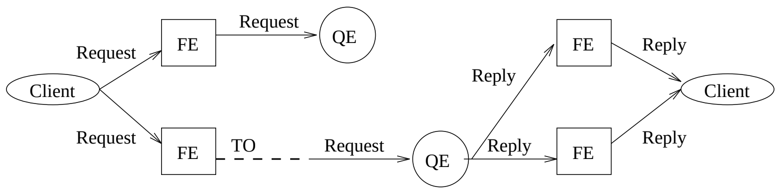
In the first scheme, called BASIC, the FE_P picks one QE according to one of the load balancing approaches that will be explained in the next section, and forwards the request to it. If the FE_B does not receive the reply for that request after half the deadline for this request, then the FE_B times out, picks another QE (namely $QE(FE_B, 1)$), and sends the request to it.

This scheme can handle a failure of either a QE or an FE: If the FE_P fails, then it will not forward the request to any QE, so no reply will be received at the FE_B after half the deadline. Therefore, the FE_B will pick a QE and send it the request. The QE will service the request, and then send the reply to both FEs, and the surviving FE will forward the reply to the client. If FE_P fails after sending the request, FE_B will receive and process the request, and forward the reply to the client.

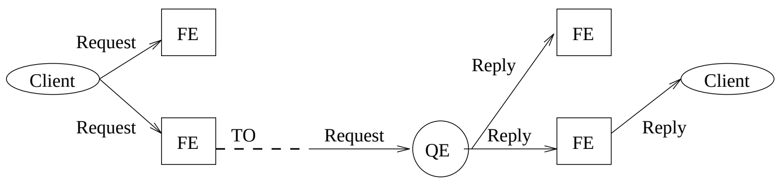
On the other hand, if $QE(FE_P, 1)$ fails, then no reply will be received at the FEs by half the deadline. In this case, again, FE_B will pick a QE and send it the request. Since we assume only a single failure, $QE(FE_B, 1)$ will service the request, and send the reply to both FEs who will forward it to the client.



(a) Basic: failure free scenario



(b) Basic: first QE fails



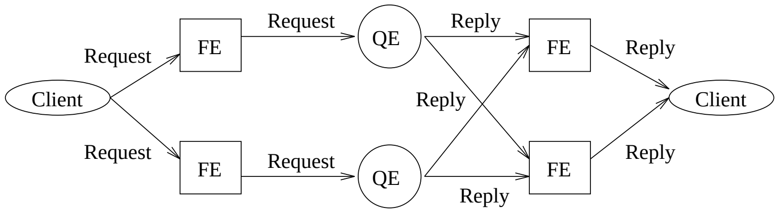
(c) Basic: primary FE fails

Figure 2: The BASIC scheme

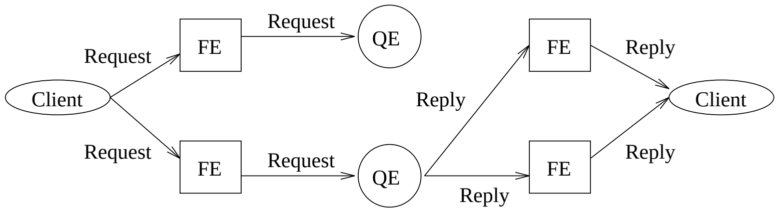
An illustration of the behavior of BASIC when there are no failures appears in Figure 2(a). Figure 2(b) illustrates what happens when a QE fails, and Figure 2(c) illustrates a scenario in which the primary FE fails. From this figure, it is easy to verify that this scheme produces 7 messages for each request when there are no failures, 8 when the first QE that is picked is faulty, and 6 if the the primary FE fails before sending the message to the QE.

Active Replication: ACTIVE

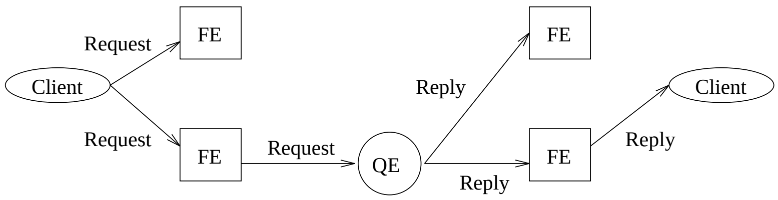
In this scheme, both FEs act as primary for each request. That is, immediately after receiving a request from the client, both FEs select a QE to handle the request and forward the request to the chosen QE. Both QEs process the request in parallel, and send the reply to both FEs, who then forward it to the client.



(a) Active: failure free scenario



(b) Active: one QE fails



(c) Active: one FE fails

Figure 3: The ACTIVE scheme

In this case, as illustrated in Figure 3, there are 10 messages in failure-free runs, 8 messages if the first QE fails, and 6 if the primary FE fails. Note that when the primary fails, there is much less message overhead. Also, since when the primary FE fails, only one QE will send replies, the remaining FE will have less work to do. Thus, after a failure of an FE, the system would be able to sustain higher throughput, although would of course be vulnerable to a single additional failure.

Sequential Requests: SEQ

In SEQ, FE_P also initially picks one QE. However, with this scheme, if FE_P does not receive a reply after **one third** of the deadline, it times out and picks a second (different) QE. The FE_B, on the other hand, waits for **two thirds** of the deadline, and only if it does not receive a reply by then, it chooses a QE which is different than the two that were supposed to be picked by the primary. Formally, $QE(FE_P, 1) \neq QE(FE_P, 2) \neq QE(FE_B, 1)$.

This scheme can sustain simultaneous failures of two QEs, and all possible failure scenarios of a single FE and a single QE. On the other hand, it requires a replication degree of 3, and increases the processing overhead imposed on the FEs. Moreover, the deadlines are much tighter in terms of time-outs. In particular, in case the FE_P fails, there is less time to service the request and forward the reply to the client, making the system more vulnerable to high loads.

For SEQ, as illustrated in Figure 4, there are 7 messages in failure-free runs, 8 messages if the first QE fails, 9 messages if the second QE fails, and 6 if the primary FE fails.

Sequential Requests with a Common QE: Common-SEQ

Common-SEQ is similar to SEQ, only that the second QE picked by the FE_P is the same as the one picked by the FE_B. That is, $QE(FE_P, 2) = QE(FE_B, 1)$ This scheme requires a replication degree of 2, and thus can only handle a failure of a single QE. On the other hand, it can handle more failure scenarios of a single FE or a single QE than BASIC, and has less overhead than SEQ. This scheme generates the same number of messages as SEQ.

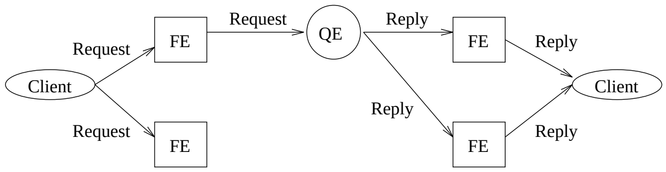
Parallel Requests: PAR

In PAR the FE_P picks two QEs and sends the request to both immediately. In this case, one of the QEs starts serving the request immediately, while the other QE waits for one third of the deadline. If the first QE is not faulty, and therefore serves the request in time, it will send the reply also to the second QE (i.e., $QE(FE_P, 2)$), in addition to sending it to the two FEs. This way $QE(FE_P, 2)$ will not service the same request redundantly and unnecessarily. Only if $QE(FE_P, 2)$ does not receive the reply before one third of the deadline, it will service the request.

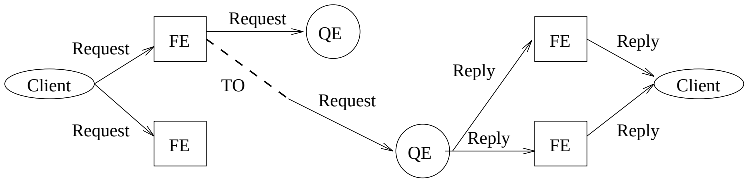
If both $QE(FE_P, 1)$ and $QE(FE_P, 2)$ fail before sending the reply, or if FE_P fails before sending the request to the QEs, FE_B will take over. After 2/3 of the deadline, FE_B will send a request to $QE(FE_B, 1)$, which will be the QE that services the request.

PAR enjoys the same degree of fault-tolerance as SEQ, and of course also requires the same replication degree. However, by sending the requests to both QEs at once, it shifts some of the management overhead to the QEs, which may be less loaded than the FEs.

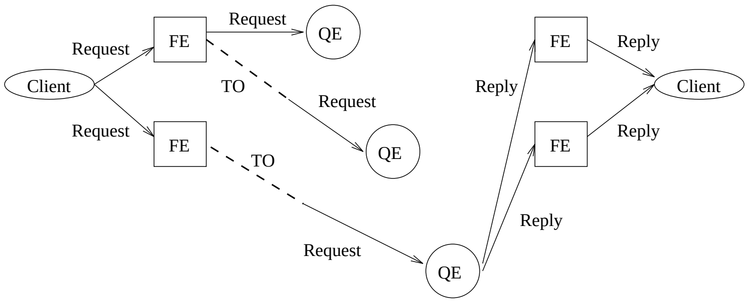
Figure 5 illustrates the messages generated with PAR. In this case, there are 9 messages in failure free runs, 8 messages if the first QE fails, 9 messages if the second QE fails, and 6 if the primary FE fails.



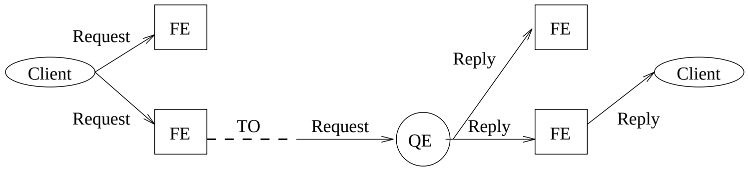
(a) SEQ: failure free scenario



(b) SEQ: first QE fails



(c) SEQ: second QE fails

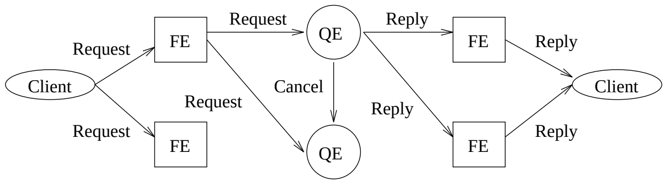


(d) SEQ: primary FE fails

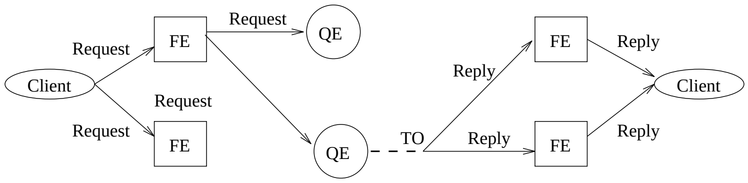
Figure 4: The SEQ scheme

Parallel Requests with a Common QE: Common-PAR

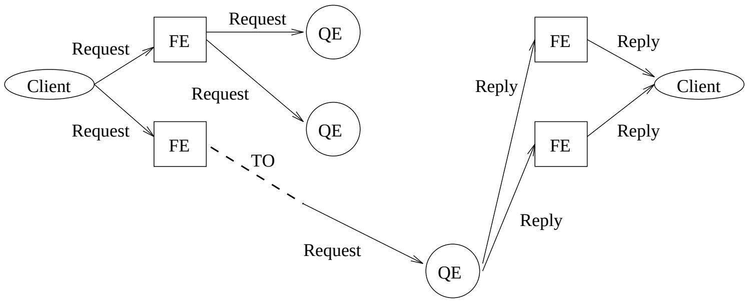
Common-PAR is analogous to PAR, with the difference that the QE picked by the FE_B is the same as the second QE picked by the FE_P. That is, same restriction as in Common-SEQ: $QE(FE_P, 2) = QE(FE_B, 1)$. This scheme generates the same number of messages as PAR.



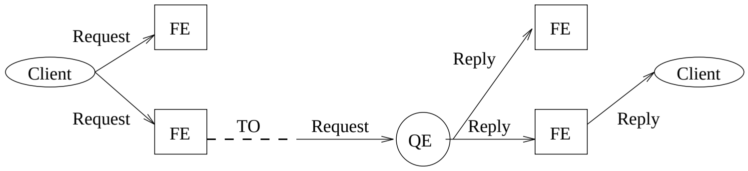
(a) PAR: failure free scenario



(b) PAR: first QE fails



(c) PAR: second QE fails



(d) PAR: primary FE fails

Figure 5: The PAR scheme

4 Load Balancing

In the previous section we have described the different schemes that are used in order to guarantee fault-tolerance. In this section we describe four load balancing approaches that can be used by the fault-tolerance schemes to pick the QEs.

These approaches are called RANDOM, RR, REQS, and LOAD. RANDOM picks QEs in *random* fashion, while in RR QEs are picked using a *modified round-robin* scheme: each FE starts from a different QE and proceeds in round-robin fashion picking the next QE for each new request. The advantage of both of these approaches is that they are simple to implement. However, since they do not take into account the load on the QEs, they may perform poorly. Moreover, in the randomized scheme (RANDOM) there is a positive probability that the same faulty QE will be chosen by both the FE_P and FE_B for the same request. Thus, RANDOM may actually decrease the degree of fault-tolerance offered by the system.

Assuming there are k QEs that can service each request in the system, we can compute the probability that a faulty QE will be picked for all attempts to service a single request. Clearly, in BASIC and ACTIVE the probability for this happening is $\frac{1}{k^2}$, since in each of these schemes two QEs are picked at random. For similar reasons, using SEQ, Common-SEQ, PAR, and Common-PAR the probability is $\frac{1}{k^3}$. Note that this probability depends on the number of QEs that can handle each request, which might be significantly smaller than total number of QEs. For example, if only 2 QEs can handle a particular request, then the probability of losing a request that can be serviced by a QE that failed becomes as high as 25%, which is already very significant.

In REQS, the FEs choose the QE with the least number of pending requests, i.e., requests that were sent to them but for which no reply was received. Of course, since an FE cannot keep track of the QEs picked by the other FE, this estimate of the number of requests of QEs is based solely on the requests that this FE has handled. This approach measures the load imposed on the QEs indirectly, through the number of pending requests. This is not completely accurate, since different requests may take different time to service. Thus, intuitively, REQS may be a not-so-refined heuristic. Also, when a QE fails, since it stops replying to requests, the load recorded for it at the FEs will not decrease, so new requests will not be directed to it.

Finally, LOAD uses the *actual* load on each of the QEs to make its load balancing decisions. This can be done since we were only running simulations, and this information is available inside our simulator. Typically, in practice, this cannot be done. However, this method is used as a "best case" or "optimal case" to which we can compare the other schemes. As in REQS, the natural fault detection will be present, when a QE stops or is slow in replying to requests.

5 Simulations

Our simulations were done using timing figures taken from an emulation of an IN coprocessor done in [5]. That emulation was carried out on an IBM^{TM} SP2^{TM} computer, which consists of RS 6000 nodes connected by a fast network. Similar configurations are likely to appear in actual implementations of distributed servers.

In our simulations we have looked at the number of requests whose deadlines were missed *vs* those that were serviced in time, when different fault-tolerance schemes, different load balancing schemes, different number of QEs, and different rate of arrivals of requests from the client. We have also looked at the behavior of the queue of events for each node on the system. That is, the events that are waiting to be serviced by that node. The length of this queue is naturally not known, or at least not fully known, to the node, but is available for the simulator.

5.1 Simulator and Simulation Parameters

Our evaluation was done by implementing a discrete-event simulator where the events driving the simulation are the arrival, start, and completion of a call in the client nodes and front ends as well as

parameter name	values assumed
sim time	$1,000$ or $10,000$
avg service time	distributed with mean $50, 250, 500$
call rate	10K reqs/sec, 15K, 20K, 25K, 30K, 35K
load balancing	RANDOM, RR, REQS, WORK
distribution of service time	poisson, uniform, biased
fault tolerance	BASIC, SEQ, COM-SEQ, PAR, COM-PAR, ACTIVE
num QEs	$2, 4, 6, \ldots, 16$
dropping	$0, 1$
garbage collection period	$3\times$ deadline

Table 1: Parameters for Simulations

in the query elements. We generated sets of calls and ran the different policies on the same call sets. In the experiments, we have fixed the time it takes for a message to be propagated in the network (40 microseconds, corresponding to user-to-user network delay using the fast switch of the SP-2 and the active messages API [4]), the time it takes for the garbage collection (1 microsecond) and the processing requests at the FEs. All failures were injected midway into the experiments.

The simulation parameters (input variables) that can be controlled are described below and summarized in Table 1.

- **simulation time**: length in milliseconds of the simulation.
- the **average search time**, c: the search time at the QEs of the arriving calls is assumed to be uniformly distributed with mean c.
- the **call rate**, γ: This parameter controls the interarrival time between calls. The client sends calls to the FE at that rate.
- **distribution**: controls the distribution of the client request rates; can be uniform, poisson, or biased distribution.
- **load balancing** scheme ; see Section 4
- **fault tolerance** scheme; see Section 3
- **number of QEs**; determines the number of QEs present in the current run.
- **dropping**; enables dropping of calls at the FE when a specific threshold is violated (does not forward the call to the QE for processing.
- **garbage collection period**: interval after which unclaimed requests are freed.

5.2 Simulation Measurements

Our simulations have yielded several anticipated results, as well as several non-trivial results. As one would expect, the maximum sustainable throughput of the system increased with the number of

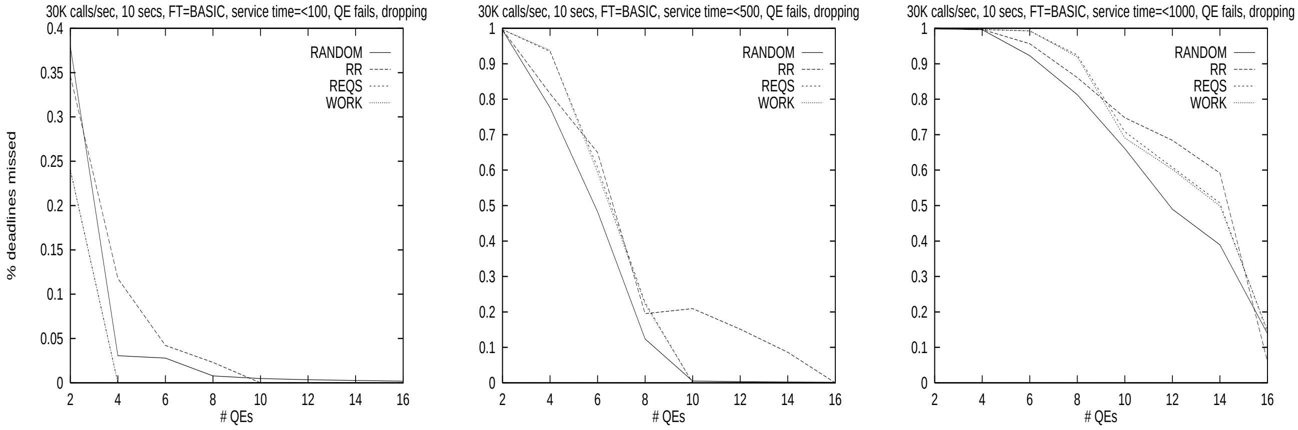


Figure 6: The percentage of missed deadlines as a function of the number of QEs when one QE fails while using the BASIC fault-tolerance scheme with arrival rate of 30,000 requests per second.

QEs, and this was more noticeable as the average service time for each request was longer, since this is when the QEs become the bottleneck. Similarly, ACTIVE was able to sustain lower throughput than other methods, since it imposes the biggest overhead.

An encouraging, yet non-trivial, result of our simulations is that REQS performed almost as well as LOAD, even when there was large variance in the service time between different requests (recall that REQS has no access to such information). It is our contention that this is being naturally compensated over time by the fact that if a certain QE has to service lengthy requests, then the number of pending requests recorded for it by the FEs would become very large, and thus the QE will not be chosen to handle future requests. Also, both these scheme were able to overcome failures of QEs much better than RR and RANDOM since soon after a QE has failed, it was perceived as overloaded, and was not chosen afterwards to service requests. (Again, the natural fault detection.)

As for fault-tolerance schemes, BASIC was able to sustain the highest throughput, although PAR and Common-PAR came very close. Since they provide a higher degree of fault-tolerance than BASIC, they seem promising. On the other hand, PAR and Common-PAR require the deadline be larger compared to the service time of requests plus the network latency compared to BASIC (they need two timeouts instead of only one), which means that in some cases they cannot be used.

We now turn to a more detailed discussion of our simulations results:

5.2.1 Changing the Number of QEs

Figure 6 illustrates the effect that changing the number of QEs has on the number of deadlines missed. In this experiment we have used the basic scheme, an average arrival rate of requests of 30,000 per second, and measured the number of missed deadlines when a QE fails in the middle of the run. As expected, the system performance improved with the number of QEs, especially when the service time for requests was large enough to cause the QEs to be the bottleneck in the system.

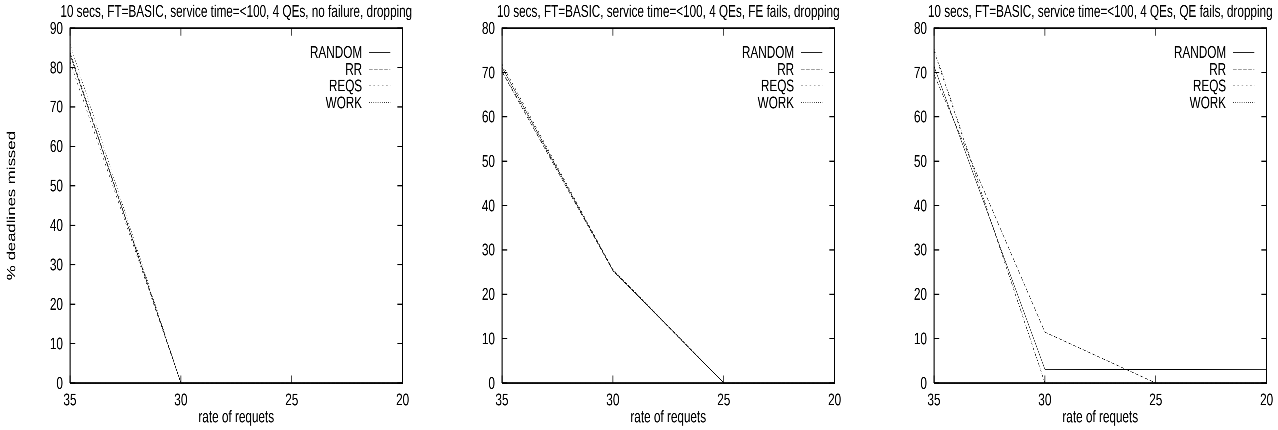


Figure 7: The percentage of missed deadlines compared to the arrival rate of requests while using the BASIC fault-tolerance scheme.

5.2.2 Load Balancing Schemes

Figure 7 illustrates the effect of various load balancing schemes and variable arrival rates of requests on the number of missed deadlines in the system. This comparison is done for the case where there are no failures, when an FE fails, and when a QE fails. With no failures, or when an FE fails, all load balancing schemes behave roughly the same, since the failure detection capability of REQS and LOAD only applies to failed QEs. However, when a QE fail, REQS and LOAD are able to sustain up to 30,000 calls per second without missing any deadlines, while both RR and RANDOM exhibit misses. At a rate of 35,000, all suffer from $\approx 70\%$ misses, since at this point the load on the FEs is far too high, and it overshadows any misses caused by the failed QE. Note that both REQS and LOAD behave the same in these graphs.

5.2.3 Load Balancing Schemes

When the system is underloaded, then by definition, no deadline should missed. However, when a QE fails and the RANDOM load balancing scheme is used, deadlines will probably be missed. This is because with RR there is a positive probability that all QEs picked for a certain request will be the same, and if that QE is faulty, then the request will not be serviced at all. We can calculate[1] the number of missed deadlines for using the RANDOM load balancing scheme as follows. Denote by QEs_{picked} the number of QEs that receive the request (e.g., $QEs_{picked} = 2$ if FT=BASIC). The number of missed is approximated by

$$\frac{numberofrequests}{(numQE)^{QEs_{picked}}} \tag{1}$$

Also, the point of overload in case a QE fails depends heavily on the load balancing scheme used. For example, with the BASIC scheme and an arrival rate of requests of up to 30,000 per second, when

[1] We have also verified this behavior through simulations, but do not show the results and graphs here for lack of space.

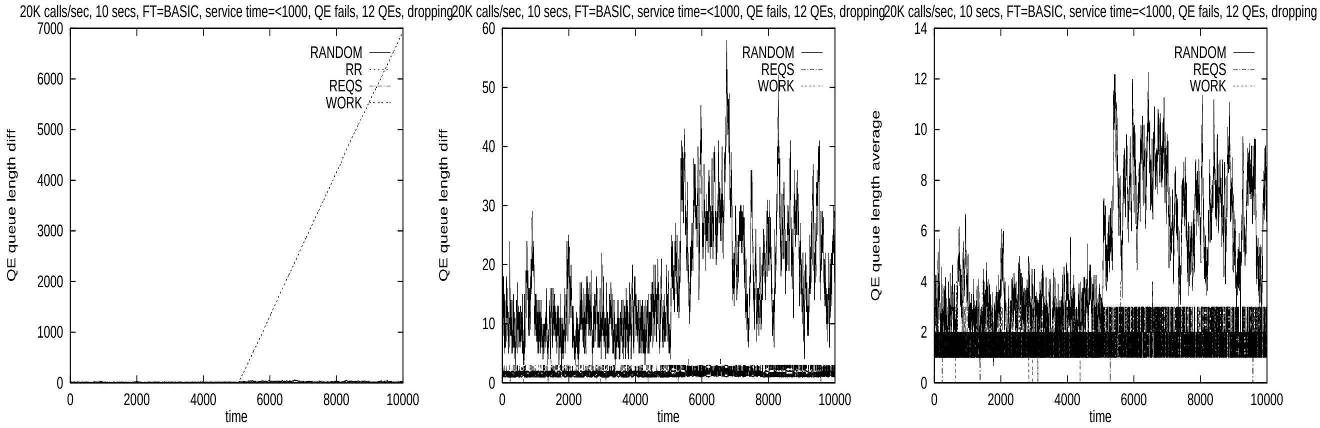


Figure 8: Difference between max and min QE lengths for all (leftmost) and all but RR (middle) schemes; rightmost graph shows the average queue length of the QEs for all but RR scheme.

a QE fails, REQS and LOAD do not miss any deadlines. However, RANDOM exhibits a number of missed deadlines as shown in Eq (1), while with RR the number of missed deadlines is inversely proportional to the number of QEs in the system. This is explained by the fact that there are fewer calls to be re-issued, thus lowering the overhead in the system.

Note, however, that with RR more deadlines are missed than with RANDOM, which is explained by the following somewhat intriguing property of RR. RR is the only load balancing scheme that guarantees to pick two (or three) different QEs for each request, which makes this scheme more robust then other schemes, but vulnerable to transient overloads. This is particular disadvantageous when a first QE fails. The second QE will become overburdened with servicing requests sent to it by the primary FE as as well as (re-issued) requests sent to it by the backup FE when its timer expires. This behavior can be seen in Figure 8 (middle graph). (The other two graphs do not show the RR scheme since due to this problem, the queue length of the "buddy" QE becomes much longer than the others, which would have overshadowed what is happening with the other schemes.) In addition, we have observed that the average queue length for RR between the interval [0:5000] (i.e., before the failure occurred) is around 4 requests.

Another interesting observation is that the dropping of requests does not significantly contribute to controlling the explosion of the queues of the "buddy" QE, after the failure of the other QE. Our dropping policy examines only the number of requests that the FE believes are pending in the system, regardless of how many of them where sent to a particular QE. Even worse, recall that our implementation age pending requests using the garbage collector event (Section 3), and does not start dropping messages unless the number of pending requests goes beyond a pre-defined high watermark. In such situations, it is possible that the influx of requests is small enough not to warrant dropping, but the *amount of work* it generates at the "buddy" QE is large enough that all requests sent to it miss their deadlines. This happens when the high watermark is smaller than the call rate multiplied by the garbage collection period.

Two possible solutions for this problem are (experiments are left for future work):

- Implementation of a real failure detection and reconfiguration to contain the problem. The

FEs would exclude the failed QE from the pool of available QEs and adjust their policies to comply with the new set of QEs.

- Implementation of more sophisticated dropping policies, which reduce this problem. For example, if a different threshold would be supported per QE, the problem would vanish. Note that this is different from REQS load balancing policy, since the information about the QEs would be used to screen out calls, not to choose QEs to send calls to.

In this work we have considered QEs that are powerful and can respond to any requests. The RANDOM, REQS, and LOAD polices work well in this environment, since they have the ability to choose from among many QEs. However, we would see the same wild-queue behavior in a more restricted model, where calls can only be looked up by a subset of QEs. In this case, even a more sophisticated load balancing policy may not help, but a more complete dropping policy would.

We have also looked at the effect of the different load balancing schemes on the length of the queue of the FEs. We have observed in cases of overloads, while using the ACTIVE fault-tolerance scheme, the FEs with all load balancing schemes show approximately the same number of requests in their queues. However, we can see in Figure 9 that after a failure occurs, the number of requests drops more steeply for RANDOM and RR (bottom 2 curves after time 5000) than for REQS and LOAD (top 2 curves after time 5000).

Recall that in ACTIVE both FEs send the request to a QE regardless of whether there is a failure or not. This is because REQS and LOAD have natural fault detection built in (see Section 4), so they do not issue requests to the failed QE. This causes all requests to be eventually processed and returned to the FEs, keeping them busier. On the other hand, RANDOM and RR continue to send requests to the faulty QE, and since this QE does not reply, the FEs have less work to do. Note that with other fault-tolerance schemes, when the system is highly overloaded, timers expire at the FEs which cause them to re-issue requests, and a similar behavior is observed. On the other hand, when the system is underloaded, or only slightly overloaded, a failure causes the load on the FEs to grow, since suddenly they have to reissue requests. However, here, with RR, REQS, and LOAD the load increases more than with RANDOM, since every reissued request will be serviced with the former methods, and therefore the FEs will do more work to forward the replies to the client.

5.3 Fault-Tolerant Schemes

The point where the system becomes overloaded, and therefore starts missing deadlines, depends heavily of the type of fault-tolerance scheme used. Our simulations have shown that the ACTIVE scheme can support a throughput of up to 30,000 requests per second without missing deadlines. This is worse than the fault tolerance schemes that were able to withstand a rate of up to 40,000 requests per second without showing missed deadlines. This is because with ACTIVE both FEs have to send a message to a QE for each request, while with the other schemes, as long as there are no failures, each FE sends a message to a QE only for half of the requests.

By examining the queues of the FEs under different fault-tolerance schemes, we can also detect some interesting phenomena. Figure 10 illustrates the behavior of these queues during overload, before and after a failure of one of the FEs. We can see from the figure that SEQ and Common-SEQ as well as PAR and Common-PAR behave pairwise similarly. This is because the amount of work is

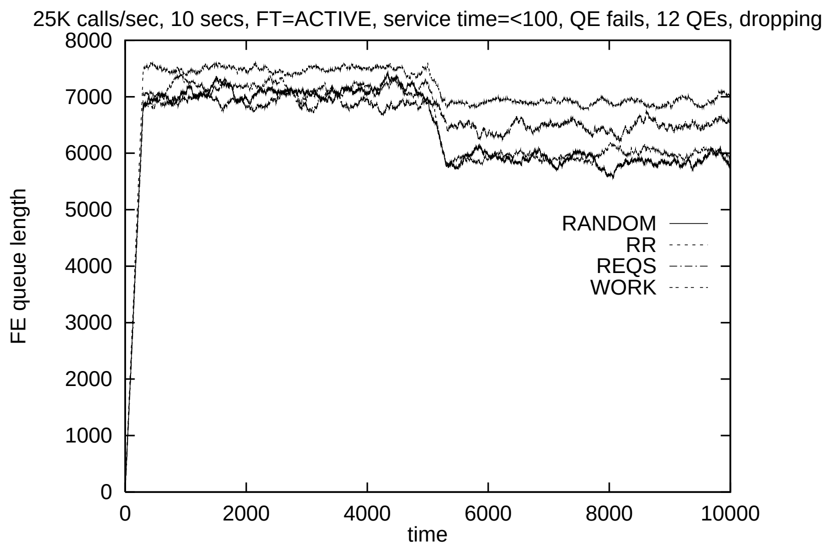


Figure 9: Queue lengths when using different load balancing schemes

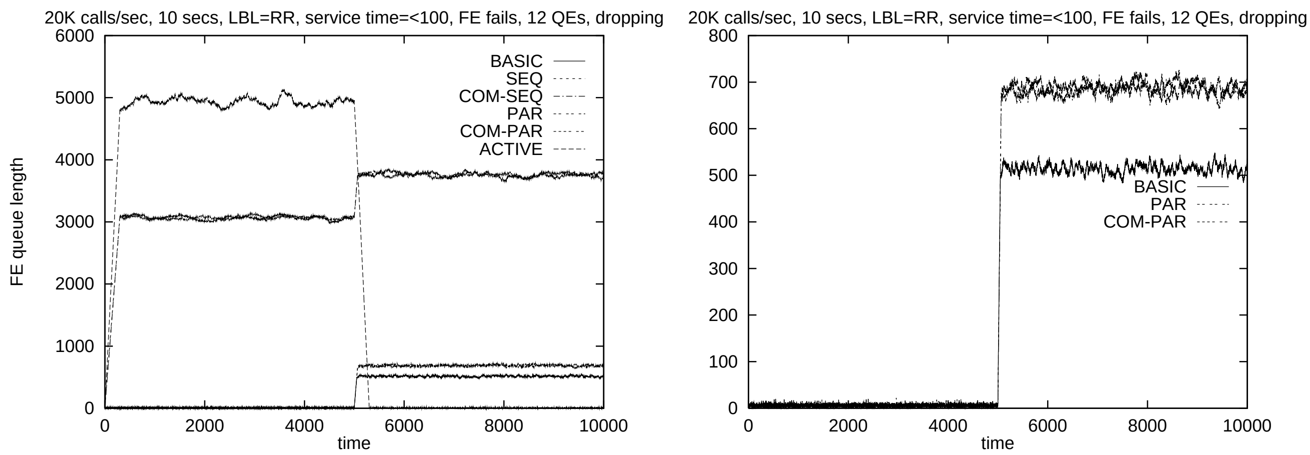
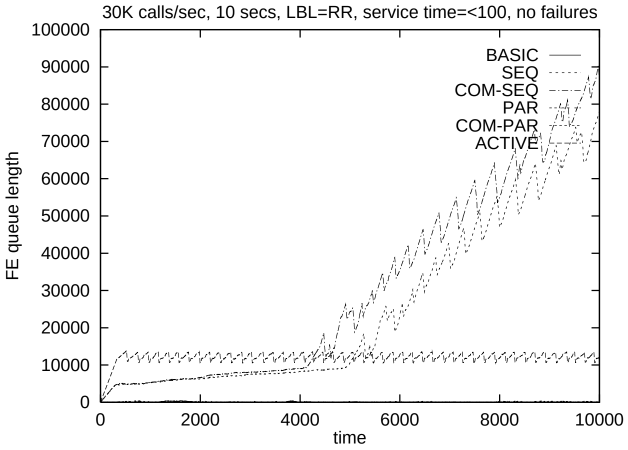


Figure 10: Queue lengths with different FT policies when the system is underloaded, but a FE fails. The left graph shows all schemes while the right graph shows the details for BASIC, PAR, and COM-PAR.

the same for the FEs regardless of whether the second and third QE are the same or not. BASIC has the lowest load of all, before (around zero) and after (around 450) faults; this is expected since it is the least robust of the algorithms. However, PAR and Common-PAR follow BASIC very closely (zero before failure, around 550 after failure), and are more robust. SEQ and Common-SEQ present high loads in all situations (3000-4000); after the fault, the load grows, since there is a need to re-issue more requests. Most surprising is the behavior of ACTIVE. Before the failure, both FEs presented **very** high load before the failure (around 5000), because they both need to send a message for each request. However, after the failure of one of the FEs, the remaining FE suddenly has less work to do, since fewer replies are coming back from the QEs. So, after the failure of one FE using the ACTIVE scheme, the system can handle higher loads, although is less fault tolerant. (Note that, with other



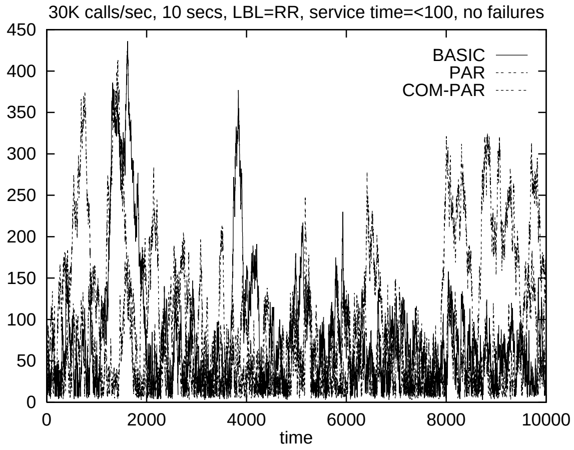


Figure 11: All FE queues when the system is overloaded but there are no failures as a function of time (left) and queue lengths in detail for BASIC, PAR, and COM-PAR schemes (right).

schemes, the system is still less fault-tolerant, but are more loaded than before the failure.)

In Figure 11 (left part) we can see how the queue length of the FE grows as a function of time before and after a QE fails. Note that the BASIC, PAR, and Common-PAR have small queues (the right part of the figure shows those schemes in detail). Again, after the fault of the QE, PAR and Common-PAR show growth behavior because of timeout expiring. On the other hand, ACTIVE remains stable throughout the runs, since it was doing all the extra work even before the failure.

6 Discussion

Choosing the right fault-tolerant and the right load balancing scheme is a crucial decision in designing successful reliable distributed servers. In this paper we presented a detailed comparison of 6 different fault-tolerant schemes combined with 4 different load balancing schemes. We have shown that a load balancing scheme can affect the fault tolerance of the server, and how a fault-tolerant scheme can affect the throughput of the system.

In the future, we would like to incorporate priorities into our model, i.e., that events with higher priority are served before event with lower priority, but also to drop lower priority events before dropping higher priority events. Also, we would like to investigate more sophisticated dropping policies and how they affect the number of messages dropped during overload. Finally, verifying how our results scale to systems in which there are more than two FEs is an interesting questions that we intend to look at.

References

[1] S. Balaji, Lawrence Jenkins, L. M. Patnaik, and P. S. Goel. Workload Redistribution for Fault Tolerance in a Hard Real-Time Distributed Computing System. In *IEEE Fault Tolerance Computing Symposium (FTCS-19)*, pages 366–373, 1989.

[2] A. Campbell, P. McDonald, and K. Ray. Single Event Upset Rates in Space. *IEEE Trans. on Nuclear Science*, 39(6):1828–1835, Dec 1992.

[3] R. Carr. The Tandem Global Update Protocol. *Tandem Systems Review*, June 1985.

[4] C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *Proc. of ACM/IEEE Supercomputing '96*, November 1996.

[5] R. Friedman and K. Birman. Using Group Communication Technology to Develop a Reliable and Scalable Distributed IN Coprocessor. In *Proc. of the TINA 96 Conference*, pages 25–41, September 1996.

[6] S. Ghosh, D. Mossé, and R. Melhem. Implementation and Analysis of a Fault-Tolerant Scheduling Algorithm. *IEEEC*, 1996. to appear.

[7] R.K. Iyer and D.J. Rossetti. A Measurement-Based Model for Workload Dependence of CPU Errors. *IEEE Trans. on Computers*, (6):511–519, June 1986.

[8] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems.* Addison Wesley Pub. Co., Inc, 1989.

[9] C. M. Krishna and K. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Trans on Computers*, 35(5):448–455, May 1986.

[10] A.L. Liestman and R.H. Campbell. A Fault-tolerant Scheduling Problem. *Trans Software Engineering*, SE-12(11):1089–1095, Nov 1988.

[11] A. Mahmood and E. J. McCluskey. Concurrent Error Detection Using Watchdong Processors - A Survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb. 1988.

[12] Yingfeng Oh and Sang Son. Multiprocessor Support for Real-Time Fault-Tolerant Scheduling. In *IEEE 1991 Workshop on Architectural Aspects of Real-Time Systems*, pages 76–80, San Antonio, TX, Dec 1991.

[13] Yingfeng Oh and Sang Son. Fault-Tolerant Real-Time Multiprocessor Scheduling. Technical Report TR-92-09, University of Virginia, April 1992.

[14] F. J. Pollack and K. C. Kahn. The BIIN Mission Critical Computer Architecture. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[15] D.K. Pradhan. *Fault Tolerant Computing: Theory and Techniques.* Prentice-Hall, NJ, 1986.

[16] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.