

Quantifying and Improving the Availability of High-Performance Cluster-Based Internet Services *

Kiran Nagaraja, Neeraj Krishnan, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen
{knagaraj, neerajk, ricardob, rmartin, tdnguyen}@cs.rutgers.edu
Department of Computer Science, Rutgers University

Abstract

Cluster-based servers can substantially increase performance when nodes cooperate to globally manage resources. However, in this paper we show that cooperation results in a substantial availability loss, in the absence of high-availability mechanisms. Specifically, we show that a sophisticated cluster-based Web server, which gains a factor of 3 in performance through cooperation, increases service unavailability by a factor of 10 over a non-cooperative version. We then show how to augment this Web server with software components embodying a small set of high-availability techniques to regain the lost availability. Among other interesting observations, we show that the application of multiple high-availability techniques, each implemented independently in its own subsystem, can lead to inconsistent recovery actions. We also show that a novel technique called Fault Model Enforcement can be used to resolve such inconsistencies. Augmenting the server with these techniques led to a final expected availability of close to 99.99%.

1 Introduction

Popular Internet services frequently rely on clusters of commodity computers as their supporting infrastructure [4]. These services must exhibit several characteristics, including high performance, scalability, and availability. The performance and scalability of cluster-based servers have been studied extensively in the literature, e.g., [4, 5, 25]. In con-

* This work was supported in part by NSF grants EIA-0103722, EIA-9986046, and CCR-0100798.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC'03, November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011...\$5.00

trast, understanding designs for availability, behavior during component faults, and the relationship between performance and availability of these servers have received much less attention. In spite of a long recognition of the importance of high availability, and the resulting fault tolerance approaches at many levels [4, 13, 27], the design and evaluation of service availability is often based on the practitioner's experience and intuition rather than a quantitative methodology.

In this paper, we advance the state-of-the-art by applying such a methodology [24] to quantify the improvement in availability of a sophisticated cluster-based Web server, PRESS, as its implementation is augmented with COTS-style components that embody different high-availability techniques. The motivation for this work originated in a recent study, in which we proposed our quantification methodology and used it to study several versions of PRESS to show its practicality. Our results showed that while PRESS achieves high performance by implementing a cluster-wide cooperative load balancing and memory management algorithm, this close cooperation significantly increases unavailability.

Cooperation between cluster nodes can increase unavailability because a fault on one node may negatively affect other nodes. Figure 1(a) illustrates the unavailability (as quantified using our methodology) and throughput of three versions of PRESS running on a 4-node cluster: an independent version that does not involve any cooperation (INDEP); the same independent version with a front-end device to hide node and application faults from end-users together with an extra server node (FE-X-INDEP); and the base cooperative version of PRESS (COOP). Clearly, the cost of cooperation is high in terms of availability: the cooperative version is about 10 times more unavailable than the other versions (99.5% availability versus 99.95%). Nevertheless, the throughput results show that cooperation is indeed extremely useful, increasing performance by a factor of 3.

Thus, we study how cluster-based servers can be designed for high availability in the presence of cooperation. Although we focus specifically on the PRESS server, our results should be applicable to a range of cluster-based Internet services, such as e-commerce servers or search engines.

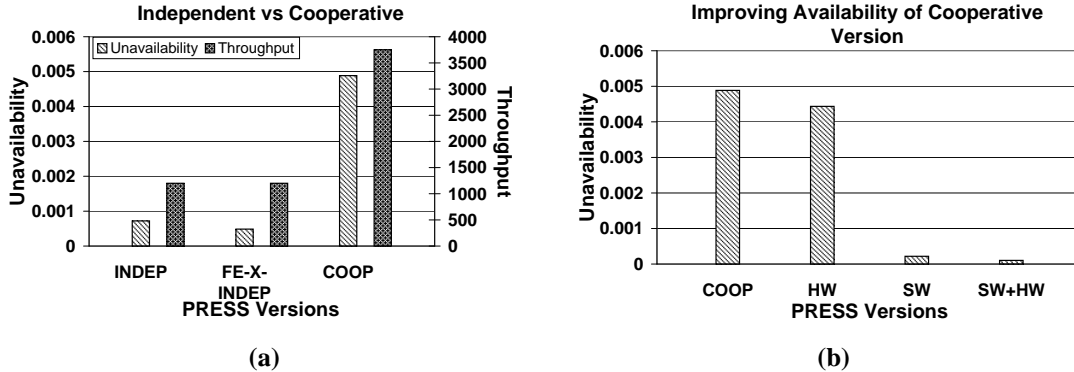


Figure 1: (a) Unavailability and performance of three versions of PRESS. In the INDEP version, server processes run completely independently. The FE-X-INDEP version adds a front-end device and 25% extra capacity to INDEP. In the COOP version, server processes cooperate to jointly manage cluster memory to more effectively cache content; this version does not employ a front end. (b) Theoretical improvement in unavailability when additional hardware (HW) and/or software (SW) are added to the COOP version.

These servers typically employ cooperation for performance and scalability, as in PRESS, and/or across multiple tiers (or groups of nodes) that implement different functionality.

Our strategy for increasing PRESS’s availability is to apply a small number of high-availability techniques that allow the cooperating cluster to detect when a peer node is unreachable, not making progress, or is down. The cooperating cluster can then remove such faulty nodes, re-admitting them once they have been repaired. The specific techniques that we implement include a robust membership service, application-level queue monitoring, front-end masking of failures, and the provision of extra computing resources.

In augmenting PRESS, we took the approach of adding COTS components where possible; where a COTS component was not easily available, we implemented the technique as a COTS component that can be reused in many different contexts. We adopted this approach because today’s Internet services are increasingly being built using COTS hardware and software to meet tight cost and time constraints.

Thus, while the techniques that we apply are well-known, a problem we address is how to merge independent, but possibly overlapping, fault semantics of the different COTS components implementing these techniques. For example, the front-end device, the membership module, and the application server itself may all have different definitions for node availability, possibly leading to inconsistent recovery actions. We show how a novel technique called Fault Model Enforcement (FME) [23] can be used to address this problem. Essentially, the designer defines a single abstract fault model and uses FME to merge the diverging views of the various components into a coherent whole.

To explore whether the above set of high-availability techniques could significantly improve the availability of PRESS, we used our methodology to estimate their im-

pact. These results are shown in Figure 1(b), which plots the unavailability of the base PRESS system (COOP) and the unavailability of PRESS when some additional hardware (front-end device, extra node, backup switch, and RAID) is applied, when all the software techniques mentioned above are applied, and when both software techniques and additional hardware are applied. The unavailability result for COOP was obtained using actual executions and our quantification methodology, whereas the other three results are analytical extrapolations from the base result.

Observe that just adding hardware such as a front-end device does not significantly increase availability. This is because these techniques do not fundamentally address the source of increased unavailability in this case: the propagation of the effects of faults because of intra-cluster cooperation. However, our results suggest that it is possible to regain the availability of the original non-cooperative version while retaining the performance advantages of cooperation by attacking the fault propagation problem using a combination of software techniques and additional hardware.

In the remainder of the paper, we will discuss our implementation of the above techniques and use our methodology to show that, with the use of a relatively small set of techniques, we can recover the availability of the independent version while retaining all the performance benefits of the cooperative version. Indeed, if we further add a small amount of hardware redundancy and sophistication, the cooperative version of PRESS can reach 99.99% availability.

In summary, we make the following contributions:

- We show that cluster-wide cooperation for load balancing and resource management can significantly increase unavailability along with performance. We also show that the original availability can be recovered with

the application of a relatively small number of high-availability techniques. Although most of these techniques are well-known, one key contribution is our quantification of their actual impact on availability.

- We show that integrating the multiple views of faults and recoveries of different availability subsystems into a coherent whole is critical to achieving high availability. In this context, we show how the novel FME technique can be applied to resolving potential conflicts arising from inconsistent views of system state.

2 Quantification Methodology

To quantify the availability of different versions of PRESS, we apply a methodology that we recently introduced in [24]. Our methodology is comprised of two phases: a measurement phase based on fault-injection and a modeling phase that combines an expected fault load together with measurements from the first phase to quantify performance and availability. In this section, we will briefly describe these two phases to provide context for the rest of the paper. With respect to our metrics, we currently equate performance with throughput, which is the number of requests successfully served per second, and define availability as the percentage of requests served successfully.

Phase 1: Characterizing Service Behavior Under Single-Fault Fault Loads. In this phase, the evaluator first defines the set of all possible faults that can occur while the service is running. Then, he describes the system’s response to a single fault of each type by injecting the fault, measuring the system’s behavior during the fault, and fitting the results to a general 7-stage piece-wise linear template.

Figure 2 illustrates our 7-stage template. Time is shown on the X-axis and throughput is shown on the Y-axis. The template starts with the occurrence of a fault when the system is running fault-free. Stage A models the degraded throughput delivered by the system from the time when an error is triggered because of a component fault to when the system detects the error. Stage B models the transient throughput delivered as the system reconfigures to account for the error. We model the throughput during this transient period as the average throughput for the period. After the system stabilizes, throughput will likely remain at a degraded level because the faulty component has not yet recovered, been repaired or replaced. Stage C models this degraded performance regime. Stage D models the transient performance after the component recovers. After D, while the system is now back up with all of its components, the application may still not be able to achieve its peak performance (e.g., it was not able to fully re-integrate the newly repaired component). Thus, stage E models this period of

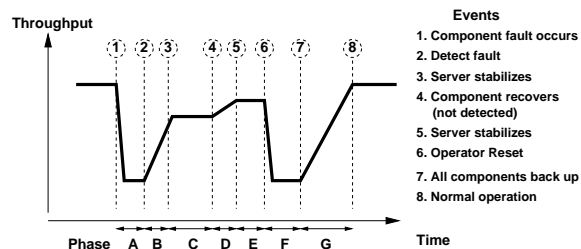


Figure 2: The 7-stage piece-wise linear template specified by our methodology for evaluating the performability of cluster-based servers.

stable but suboptimal performance. Finally, stage F represents throughput delivered while the server is reset by the operator, whereas stage G represents the transient throughput immediately after reset.

For each stage, we need two parameters: (i) the length of time that the system will remain in that stage, and (ii) the average throughput delivered during that stage. The latter is always a measured quantity while the former may be an assumed environmental value that is supplied by the evaluators. For example, the time that a service will remain in stage B is typically measured; the time for stages D and E is typically a supplied parameter. Sometimes stages may not be present or may be cut short. For example, if there are no warming effects, then stages B, D, and G would not exist. When this happens, we set the length of these states to zero.

With respect to the fault injection, the service must have completely recovered from a fault (or have been restarted) before the next one is injected. Further, each fault should last long enough to actually trigger an error and cause the service to exhibit all stages in the 7-phase template unless the server does not exhibit some of the phases for the particular fault. In these cases, the evaluator must use his understanding of the server to correctly determine which stages are missing (so that he can set the time of the stage in the template to 0). Finally, a benchmark must be chosen to drive the server such that the delivered throughput is relatively stable throughout the observation period except for transient warm up effects. This is necessary to decouple measured performance and availability from the injection time of a fault.

We have found the 7-stage template general enough to represent the behavior of widely different cooperative servers in the presence of faults. In addition to the PRESS Web server presented in this work, in our current research we have also applied the same template to a 3-tier on-line bookstore based on the TPC-W benchmark as well as a clustered 3-tier auction service.

Phase 2: Modeling Performance and Availability Under Expected Fault Loads. In the second phase of the methodology, the evaluator uses an analytical model to compute the

expected average throughput and availability, combining the server’s behavior under normal operation, the behavior during component faults, and the rates of fault (mean time to failure, MTTF) and repair (mean time to repair, MTTR) of each component. To simplify the analysis, we assume that faults of different components are not correlated, faults trigger the corresponding errors immediately, and faults “queue at the system” so that only a single fault is in effect at any point in time. These assumptions allow us to add together the various fractions of time spent in degraded modes given a 7-phase template for each fault type gathered in phase 1. If T_n is the server throughput under normal operation, c the faulty component, $MTTF_c$ the mean time to failure of component c , T_c^s the throughput of each stage s in Figure 2 when this fault occurs, and D_c^s be the duration of each stage, our model leads to the following equations for average throughput (AT) and average availability (AA):

$$AT = (1 - \sum_c W_c)T_n + \sum_c \sum_{s=A}^G (\frac{D_c^s}{MTTF_c} T_c^s)$$

$$AA = \frac{AT}{T_n}$$

where $W_c = (\sum_{s=A}^G D_c^s) / MTTF_c$.¹ Intuitively, W_c is the expected fraction of the time during which the system operates in the presence of fault c . Thus, the $(1 - \sum_c W_c)T_n$ factor above computes the expected throughput when the system is free of any fault, whereas the $\sum_{s=A}^G (\frac{D_c^s}{MTTF_c} T_c^s)$ factor computes the expected average throughput when the system is operating with a single fault of type c . Note that T_n represents the offered load assuming that the server is not saturated under normal operation, so AT/T_n computes the expected fraction of offered requests that are successfully served by the system.

Limitations. A limitation of our model is that it does not capture data integrity failures; that is, failures that lead to incorrect data being served to clients. Rather, it assumes that the only consequence of component failures is degradation in performance or availability. The model also assumes that faults do not overlap. However, we believe that the likelihood of overlapping faults for realistic fault rates is negligible, unless faults are correlated. Currently, our model does not consider correlated faults due to the limited amount of publicly available data on such faults.

3 The PRESS Server

PRESS is a highly optimized yet portable cluster-based locality-conscious Web server that has been shown to pro-

¹We refer the reader to [24] for a discussion of why the denominator is correctly $MTTF_c$ rather than $MTTF_c + MTTR_c$.

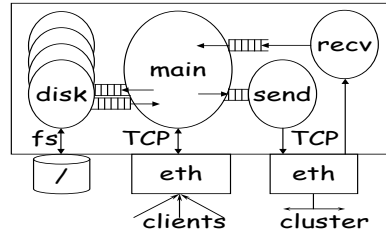


Figure 3: Basic PRESS architecture.

vide good performance in a wide range of scenarios [5, 6]. Like other locality-conscious servers [3, 25], PRESS is based on the observation that serving a request from any memory cache, even a remote cache, is substantially more efficient than serving it from disk, even a local disk. In PRESS, any node of the cluster can receive a client request and becomes the *initial node* for that request. When the request arrives at the initial node, the request is parsed and, based on its content, the node must decide whether to service the request itself or forward the request to another node, the *service node*. The service node retrieves the file from its cache (or disk) and returns it to the initial node. Upon receiving the file from the service node, the initial node sends it to the client.

To intelligently distribute the HTTP requests it receives, each node needs locality and load information about all the other nodes. Locality information takes the form of the names of the files that are currently cached, whereas load information is represented by the number of open connections handled by each node. To disseminate caching information, each node broadcasts its action to all other nodes whenever it replaces or starts caching a file. To disseminate load information, each node piggy-backs its current load onto any intra-cluster message.

Software Architecture. Figure 3 shows the basic architecture of PRESS, which is comprised of one main coordinating thread and a number of helper threads used to ensure that the main thread never blocks. The helper threads include a set of `disk` threads used to access files on disk and a pair of `send/receive` threads for intra-cluster communication. The versions of PRESS that we study use TCP for intra-cluster communication.

Reconfiguration. In its most basic form, PRESS relies on round-robin DNS for initial request distribution to nodes (although PRESS can also work with a load balancing front-end device). PRESS has been designed to tolerate node and application process crashes, removing the faulty node from the cooperating cluster when the fault is detected and re-integrating the node when it recovers. The detection mechanism employs periodic heartbeat messages. To avoid sending too many messages, we organize the cluster nodes in a directed ring structure. Each node only sends heartbeats to

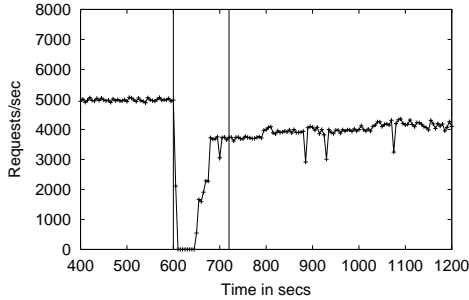


Figure 4: *Throughput of PRESS running on 4 nodes when a disk fault is injected. The vertical lines denote the period the fault is active.*

the node it points to. If a node does not receive three consecutive heartbeats from its predecessor, it assumes that the predecessor has failed. Temporary recovery is implemented by simply excluding the failed node from the server. Multiple node faults can occur simultaneously. Every time a fault occurs, the ring structure is modified to reflect the new configuration.

Once an excluded node has been repaired, it rejoins the server by broadcasting its IP address to all other nodes. The currently active node with lowest node ID number responds by informing the rejoining node about the current cluster configuration and its node ID. With that information, the rejoining node can reestablish the intra-cluster connections with the other nodes. After each connection is reestablished, the rejoining node is sent the caching information of the respective node.

Availability. Our previous study shows that this cooperative version of PRESS can only achieve close to 99.9% availability [24]. Figure 4 shows PRESS’s throughput on four nodes when a disk fault is injected, demonstrating two major causes of this version’s low availability compared to the independent version where servers on different nodes do not cooperate. First, even though the fault only occurs on one node, the throughput of the entire cluster drops to 0 until the fault has been detected through the loss of three heartbeats. Second, the cluster splinters into two sub-clusters. A cluster is *splintered* when there are mutually disjoint co-operation sets, in this case one with 3 nodes and one with 1 node. Performance in a splintered state is sub-optimal because resources are not globally managed. In this case, PRESS is unable to re-integrate because the faulty node did not crash, violating the fault model assumed by the designers. Performance thus remains sub-optimal, even after the component has been repaired, until an operator can restart the singleton sub-cluster.

4 High-Availability Techniques

In this section, we describe the high-availability techniques that we have implemented to improve PRESS’s availability. Our approach was to apply an evolutionary software approach; that is, rather than re-design PRESS from scratch with availability in mind, we augmented the existing implementation with techniques for increasing availability. Further, we implemented these techniques as separate, self-contained subsystems rather than a monolithic structure. Such an approach not only allows more careful design, implementation and testing of individual subsystems, but also easy piece-wise use in other servers. Finally, our approach is guided by the current industry practice of constructing layered services from COTS subsystems.

Our adoption of this approach led to an interesting problem. When different but related availability techniques are implemented in separate subsystems, their views of the system may overlap; each subsystem has a different set of nodes in various states (e.g. up, down, timed-out). A second contributing factor to overlap is that each subsystem has slightly different definitions for fault symptoms and what it means for a component to fail. This overlapping can lead to conflicting recovery behaviors. We show that Fault Model Enforcement (FME), a technique recently proposed in [23], provides one way of overcoming such conflicts.

We start our discussion by adding a front-end and extra processing capacity to PRESS to detect and hide node failures from end clients. This is a good starting point because it is a standard industrial solution. Then, we extend PRESS to include a robust membership algorithm and application-level queue monitoring. Finally, we describe FME.

4.1 Front-End Request Fail-Over and Extra Capacity

Today, there are numerous front-end products for cluster-based servers. For this study, we chose to use the Linux Virtual Server (LVS) [20] because it did not require additional specialized hardware. In LVS, the actual server nodes are hidden behind a front-end request distributor. The front-end uses IP tunneling to transparently forward client packets to server nodes; server nodes reply directly to the clients without going back through the front-end for scalability.

The LVS front-end has two basic capabilities. The first capability is to dynamically load balance incoming requests using one of several request distribution algorithms. We simply use the round-robin algorithm given that PRESS already includes sophisticated code for request (re)distribution. The second, and most important capability for our purposes, is to mask node failures by not routing any incoming requests to a failed node. To do this, the front-end needs to monitor the server nodes and detect when they fail. We use Mon [28],

a service monitoring daemon for failure detection. Mon is configured to test the availability of each node and trigger an action when it comes up/goes down. This action is the addition/deletion of entries in the table used by the front-end to distribute requests. The probes are done using ICMP echo messages (i.e., pings) sent every 5 secs. If the daemon does not receive three successive replies from a node, it assumes that the node is down.

Front-end failures can be handled by having a redundant front-end, heartbeats, and IP take-over. Even though our current LVS setup does not include a redundant front-end, we do model this ideal configuration. In addition to the front-end, we also experiment with extra hardware capacity in the form of an extra server node. Extra capacity allows the server to deal with single failures without loss in throughput.

4.2 Group Membership

As already discussed, a major cause of unavailability for PRESS is the splintering of the cooperating cluster upon a fault that the system was not designed to handle. Thus, our first step in improving PRESS's availability with respect to the collaboration between the server nodes is to implement a robust membership protocol to allow PRESS to recover correctly from such splintering once the underlying fault has been repaired.

Specifically, we implement a variation of the three-round membership algorithm described in [10]. Nodes participating in the membership protocol arrange themselves in a logical ring, with each node monitoring its upstream and downstream neighbors using heartbeat messages. Members are added to and removed from the group using a two-phase commit protocol. A node attempts to exclude a neighbor from the group after it fails to receive three consecutive heartbeat messages from that neighbor, acting as the coordinator for the exclusion. New nodes can join a group by broadcasting a join request to a well-known IP multicast address. All current members reply to the broadcast; the new node chooses one of these members to be the coordinator for adding it to the group.

This algorithm handles network partitions by forming independent sub-groups which can then each make progress. It converges as long as all participating members have a consistent view of the network; that is, if A cannot communicate with C , then any other member B able to communicate with A must *not* be able to communicate with C .

We implement the membership protocol as an independent service that publishes the current group membership to a shared-memory segment. Applications can directly attach this shared-memory segment or link with a provided client library. The library spawns a thread that periodically checks the shared-memory segment and calls the application back when there are updates. The application must provide two

functions for the library to call back: one function for when a new member has joined the group (`NodeIn()`) and one for when a member has been removed (`NodeOut()`). It also provides a function (`NodeDown()`) that the application can call if the application detects that a node currently in the group is down. In our implementation of PRESS, each server process maintains a directory of cooperating peers, called its *cooperation set*. The `NodeIn()` and `NodeOut()` functions update this set.

4.3 Queue Monitoring

While heartbeat monitoring is a popular technique, it is far from sufficient. The main reason for this is that, even when a full membership protocol is implemented, it *only* monitors the nodes' ability to send and receive messages. It says nothing about any other functionality of the node or application. For example, in PRESS, if a disk on some node p fails, then the PRESS process running on p will grind to a halt when the disk queue fills up. In turn, all cooperating nodes will shortly come to halt, waiting for the process running on p , bringing system throughput to 0. Yet, during this service interruption, the membership service would happily report that all nodes are up and are members of the group.

Thus, we needed to supplement the membership protocol with some form of resource monitoring. We had two design options: to implement an application-independent service that monitors various node resources, such as disk and memory, or to implement an application-specific mechanism that monitors the progress of the application (server) as a whole. We chose the latter approach because subsystem faults, particularly transient ones, may never turn into errors at the application level if the application does not make use of that subsystem. Even though we chose to implement an application-level mechanism, by implementing a generic self-monitoring queue, we maintain a high degree of generality because many cluster-based services are structured as components connected by event or task queues [5, 31].

As shown in Figure 3, the PRESS architecture exactly centers around a number of components connected by queues, so it was quite easy to implement queue monitoring. We separate the send queue of each process into $n - 1$ self-monitoring queues, one for each peer process in the set. Any fault that causes one of the processes to fall behind will cause its peers' corresponding send queues to build up. The implementation maintains two thresholds for each of the queues. A queue build-up that reaches the first threshold is viewed as an indication of overload but not necessarily of failure of the peer at the other end of the queue. In this instance, most requests destined for the overloaded queues are rerouted to other cooperative peers or the disk queue. In order to continue probing the overloaded queue, however, a small fraction of the requests are still routed to it. If the problem is

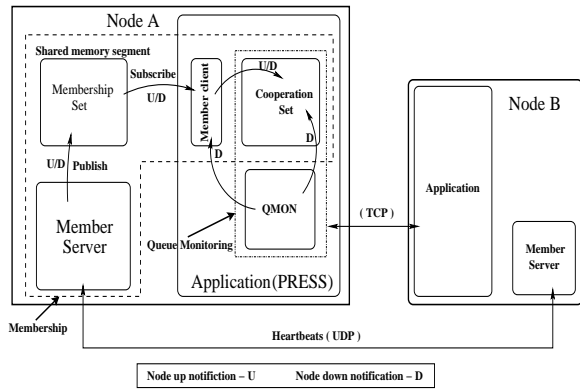


Figure 5: *External membership and application queue monitoring interactions. Arrows inside “Node A” indicate flow of information about nodes’ status.*

persistent, rather than a transient overload, the queue buildup will eventually reach the second threshold. When this happens, we assume that the peer at the other end has failed and so remove that node from the cooperative set.

4.4 Combining Group Membership and Queue Monitoring

Neither group membership nor queue monitoring is sufficient by itself, so one of the systems we study combines these two techniques as in Figure 5. Unfortunately, when they are combined, inconsistencies can arise from their distinct views of whether a node is operating correctly. For example, if one of the server processes hangs for some reason but the membership server on the same node does not, the views of group membership and the queue monitoring mechanisms diverge. In such a scenario, the faulty peer keeps getting removed by the queue monitoring mechanism and added back by the membership service.

One might think that an easy solution is to simply give precedence to one of the two mechanisms. However, this does not work. If precedence is given to queue monitoring, then a problem occurs when the server process resumes its normal operation. Essentially, the question is when the queue monitoring mechanism should start “believing” the group membership service again. On the other hand, the reverse ordering also does not work, since the hung process would impede the entire server for the duration of the fault.

Thus, a closer integration of the two mechanisms seems necessary. *However, such integration is not usually possible given that in practice these techniques would be implemented by different COTS software components, so we did not attempt it.* Instead, in the next subsection we show how to use FME to resolve conflicts between these techniques, allowing their implementation to remain separated and cleanly modularized.

4.5 Fault Model Enforcement

The idea behind *Fault Model Enforcement (FME)* is to enforce an abstract universal fault model (defined by the service designers as the set of faults that the system can correctly detect and recover from) by transforming any faults not modeled to ones that are. Interestingly, this translation may involve the active faulting of a non-faulty component. For example, if disk faults are not part of the fault model but node crashes are, FME may shutdown an entire node if the node’s disk becomes faulty, even though all other components of the node may continue to operate correctly. Although somewhat extreme, FME simplifies the design of highly available services by allowing designers to focus on tolerating a specific (and small) set of faults. Also, FME allows designers to cleanly separate multiple high-availability mechanisms, preserving their modularity and simplicity. Finally, FME can reduce the need for operator intervention by translating unexpected faults into those from which the service can automatically recover.

Defining the Fault Model. FME is a general principle and, thus, has applications in a variety of scenarios. Here we aim to demonstrate the use of an FME approach in the context of PRESS. The PRESS designers implemented a simple fault model originally: only node and application crashes were handled. By adding group membership and queue monitoring, we have added the ability to handle node unreachable and lack-of-progress faults. However, we have seen that views from the reachability and lack-of-progress monitors diverge in some cases. In these cases, the system cannot correctly detect and recover from the fault. Thus, our abstract fault model is defined by faults that make nodes unreachable (e.g., link down), as well as node and application crashes.

Implementation. We use FME to map faults that cause the status views of the queue monitoring and membership components to diverge into ones that are common to both components, i.e., application crash and node crash. Faults that can lead to divergent views include disk faults and application hangs because of transient faults or application bugs. In fact, a disk fault would eventually manifest as a process hang; however, we choose to detect disk faults independently so that we can take the node offline for repair as opposed to simply trying to restart the process.

We implement a per-node FME process that periodically (i) monitors the local disk, and (ii) probes the service by sending simple HTTP requests to the local application server. The FME process uses the SCSI Generic Interface to probe the disk directly. It takes the entire node offline for repair when it detects a disk failure *and* the application fails to respond to its HTTP probes. In this circumstance, FME assumes that the disk failure has led to an application hang or crash. This resolution also enables the front-end load balancer to detect and divert requests away from the faulty

Fault	MTTF	MTTR	Number of Components
Link down	6 months	3 minutes	4 links
Switch down	1 year	1 hour	1 switch
SCSI timeout	1 year	1 hour	8 disks
Node crash	2 weeks	3 minutes	4 nodes
Node freeze	2 weeks	3 minutes	4 nodes
Application crash	2 months	3 minutes	4 processes
Application hang	2 months	3 minutes	4 processes
Front-end failure	6 months	3 minutes	1 front-end

Table 1: *Failures and their MTTFs and MTTRs. Application hang and crash together represent an MTTF of 1 month for application failures. The number of components listed is for a 4-node cluster.*

node. On the other hand, if the application fails to respond to FME’s probes but FME believes the disk to be operational, FME will restart the application process thereby converting the application hang to a crash-restart sequence.

5 Experimental Environment

We now move to evaluating the impact of the availability components that we have added to PRESS. Measurements shown in the next section were taken on a cluster of 800 MHz Pentium III PCs, each with 2 10K rpm SCSI disks. Nodes are interconnected by a 1 Gb/s cLAN VIA network. PRESS was allocated 128 MB on each node for its file cache. In our experiments, PRESS only serves static content and the entire set of documents is replicated at each node. PRESS was loaded at 90% of saturation when running with 4 nodes and set to warm up to this peak throughput over a period of 5 minutes. We use a fifth node to supply extra capacity and a sixth node as the LVS front-end.

The workload for all experiments is generated by a set of 4 clients running on separate machines connected to PRESS by the same network that connects the nodes of the server. (The total network traffic does not saturate any of the cLAN NICs, links, and switch, and so the interference between the two classes of traffic is minimal.) To achieve a particular load on the server, each client generates a stream of requests according to a Poisson process with a given average arrival rate. Each request is set to time out after 2 seconds if the connection cannot be established and after 6 seconds if, after successful connection, the request cannot be completed. The request stream follows a trace gathered at Rutgers [5]. We made two modifications to the trace: (1) we made all files the same size to achieve stable throughput per Section 2, and (2) we increased the average size from 18 KB to 27 KB so that there are still misses when we use all 5 server nodes.

For all experiments, Table 1 shows the expected fault load per component, unless otherwise noted. For example, node crashes arrive exponentially, on average for a single node,

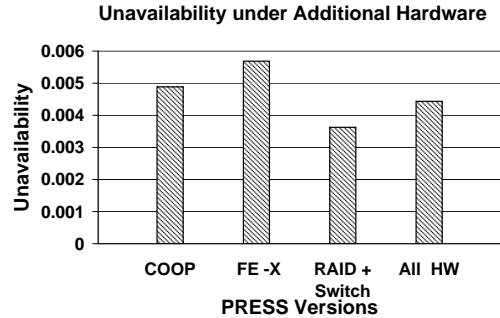


Figure 6: *Effect of adding redundant hardware on the unavailability of the base version of PRESS.*

once every 2 weeks. We derived the rates from previous works which empirically observed the fault rates of many systems [1, 14, 21, 16, 22]. We use Mendosus [19] to inject the expected fault load. Mendosus’s network emulation system allows us to differentiate between intra-cluster communication and client-server communication when injecting network related faults. Thus, the clients are never disturbed by faults injected into the intra-cluster communication.

Finally, the parameters for our fault detection mechanisms are as follows. Heartbeat messages are sent every 5 seconds. The FME tests the disk and probes the application process every 5 seconds. Queue-monitoring uses a threshold of either 512 messages of all types (i.e., the entire queue is full) or 256 request messages to decide the corresponding node has failed, and a lower threshold of 128 request messages to start routing requests away from the overloaded queue.

6 Quantifying Expected Availability

In this section, we present the resulting improvement of availability due to our techniques. Using measurements of the basic PRESS system and extrapolations based on our analytic model of availability, we begin by examining the impact of applying the traditional front-end and extra computing resources. Next, we quantify the impact of the two techniques, group membership and queue monitoring, that help PRESS to achieve more accurate views of the cluster state so that it can exclude faulty resources. This quantification is based on measurements obtained from our actual implementations of these techniques as described in Section 4. After this, we consider the implications of our FME implementation. Finally, we use our analytic model to compute the expected availability of a system that includes all of the techniques, as well as RAIDs and extra network switches.

6.1 Basic Results

Extra Hardware. Figure 6 shows the modeled impact of adding a pair of front-ends and extra capacity in the

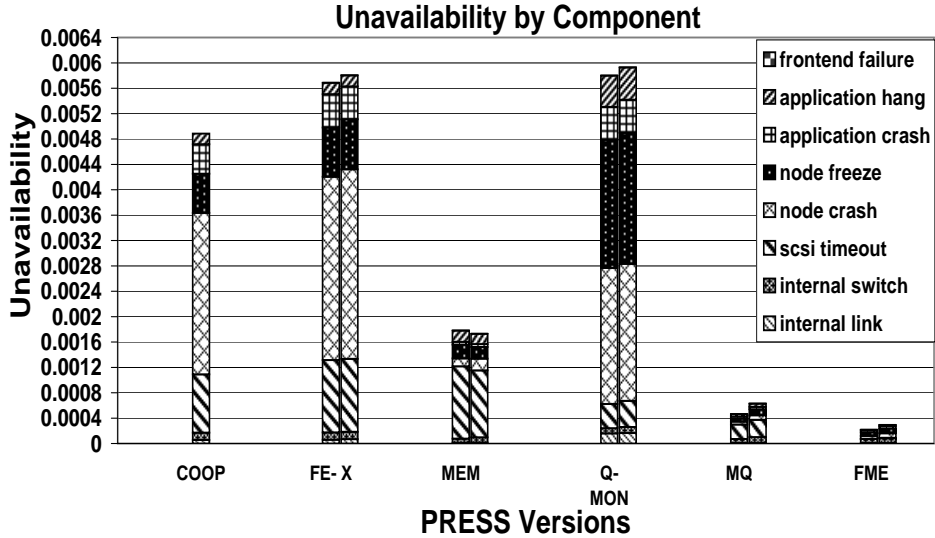


Figure 7: Comparison of experimental and modeled values for unavailability of PRESS when various high-availability techniques (and combinations thereof) are applied.

form of one additional server (FE-X), RAIDed and an extra switch (RAID+switch), and both sets of resources together (All HW) to the basic 4-node version (COOP) of PRESS. First, observe that the FE-X actually increases unavailability. While initially surprising, this is correct because by adding an extra “live” node, we have increased the probability of component faults in the system. Yet, the front-end is ineffective at masking these faults. Even though the front-end does not direct any request to the failed node, the non-faulty back-end nodes, through their request (re)distribution, will attempt to access resources on the failed node, leading the entire server to stall. Nevertheless, when this effect is tackled by our software techniques, a front-end device and extra capacity do reduce unavailability, as we discuss later.

The RAID+switch shows the impact of adding RAIDed to all nodes as well as an extra backup switch. This is a fairly standard industrial practice as disks are a significant source of faults. We modeled these additions as a reduction in the MTTF of disk failures from 1 per year to once per 438 years, and of switch failures from 1 per year to once per 4000 years². The models show that the impact of this additional hardware does not change the availability class of the system: adding 4 RAIDed and an extra switch reduced unavailability from 0.0049 to 0.0037, a 25% reduction. If we add back in a front-end and extra node, the impact of the extra hardware is even smaller.

What these results show is that simply adding redundancy at the lower hardware layers is not sufficient (unless every

layer is addressed), because any error can impact the entire system. Instead, mechanisms that address the fundamental issues of error propagation and recovery due to resource sharing are required. The basic idea of these approaches is to remove and add resources from the global view as components fail and recover.

Membership and Queue Monitoring. Figure 7 shows the impact of adding membership (MEM) and queue monitoring (QMON) to the FE-X version of PRESS (i.e. with a front end and extra node). In each pair of bars, the left bar shows the unavailability obtained through modeling based on fault-injection measurements obtained from the base version of PRESS (COOP), whereas the right bar shows the modeling results based on fault-injection measurements with the fully implemented system. The unavailability of the COOP version is shown for reference. MQ denotes MEM + QMON.

These results show that our modeling from COOP experiments predicts unavailability accurately for these versions. The results also show that the membership service allows PRESS to handle errors that halt a node or prevent communication to it: link errors, node freeze, and node crash errors. However, the MEM enhancement cannot handle SCSI errors because these only freeze threads accessing the disk. In general, any errors that stop the application without crashing the system cannot be detected with this method. In spite of these shortcomings, applying a group membership subsystem to PRESS has more impact on overall availability than adding redundant hardware.

On the other hand, while queue monitoring successfully reduces the unavailability due to SCSI errors and node crashes, it fails to improve availability for application

²We modeled the MTTF improvement of a composite system in terms of the number of components, N, and their MTTF and MTTR: $MTTF_{new} = \frac{(MTTF)^2}{N(N-1)(MTTR)}$ [26].

crashes. Worse, the Q-MON version increases unavailability in the case of node freezes and application hangs. In each of these cases, the problem is that while monitoring application-level queues is fine for detecting failure, nodes are not re-integrated upon recovery.

Combining these two approaches, the MQ version shows a significant improvement in unavailability over the COOP version, reducing unavailability by 87%. Because the system state view of each of the techniques is combined into a single view, the result is that the system can handle all errors. However, there is still room for improvement since application hang and SCSI faults lead to conflicts between the two mechanisms, with the faulty node constantly leaving and re-entering the cooperation set.

Fault Model Enforcement. The rightmost pair of bars in Figure 7 shows the impact of adding FME to the MQ version. The FME approach further reduces the impact of SCSI errors and application hangs, as these errors are turned into node and application crashes. By tackling these errors, the FME version mitigates most of the negative effects of cooperation between nodes. In fact, the FME version reaches an unavailability that is 94% lower than that of the basic version. As a result, the availability benefits of the front-end node and the extra server capacity start to show more clearly; unavailability increases by almost 3 times if we now remove these additional resources. Finally, note that this combination of techniques achieves a lower expected unavailability than a set of independent servers, while obtaining the performance of the shared-resource version.

Finally, note that the discrepancy between the unavailability based on actual measurements of the FME-based server and the unavailability based on measurements from COOP arises mostly as a result of node freeze faults. We suspect this happens because it takes longer for the working sets of the cooperative caches to stabilize when a node is removed abruptly, compared to operating under a cold start regime. We are currently investigating if we can mitigate this effect.

6.2 Modeling Other Approaches

Figure 8 shows the expected unavailability (computed by modeling from the experimental COOP results) of four additional versions of the PRESS server. The same FME unavailability as in Figure 7 is shown on the left for reference. The S-FME version models a more powerful version of the current FME implementation. From our experiments, we noted that a significant number of requests were lost due to the front-end not being aware of the cooperation sets of the back-end nodes. Back-end nodes eliminated from the cooperation set due to network or application failures, but still responding to pings from the front-end monitor, continue to receive the same stream of requests, which overloads the node in its isolated (or failed) condition. S-FME tackles this by

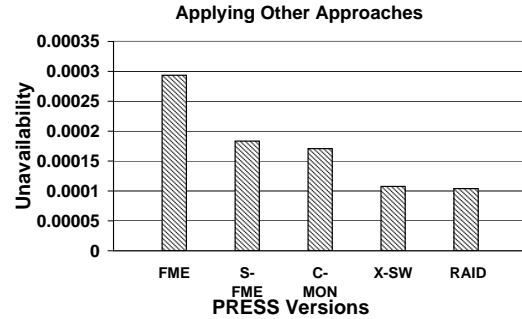


Figure 8: Results of combining all approaches with connection monitoring by the front-end node and additional hardware resources.

monitoring the cooperative set at a global level and taking isolated nodes offline to eliminate such losses. The modeled results show that S-FME represents a significant improvement, cutting unavailability by 40% compared to the current FME version. This is quite an encouraging prospect for future FME implementations.

The C-MON version is the same as S-FME, but we assume that the front-end node can detect errors using TCP connection monitoring in 2 seconds, instead of the ping-based detection with timeout after 15 seconds that we have used so far. The connection monitoring reduces the time for the front-end to detect application crashes and freezes, as well as node errors.

Next, we show the impact of adding extra hardware resources to the cluster. The X-SW version adds a fail-over switch to C-MON. The extra switch has the effect of significantly reducing the MTTF of the communication subsystem, as mentioned before. The results show that the X-SW version brings unavailability to just slightly above 0.0001. Finally, the rightmost bar shows the impact of adding a RAID to each node. Adding a RAID does not improve availability much, as our combination of software techniques and file replication is sufficient to direct requests to some working node often enough.

Finally, observe that the X-SW version pushes the availability of the cooperative version of the server ($> 99.98\%$ or 0.9998) quite close to the four nines availability class. This result is encouraging because it required little modification to any of the server components themselves.

6.3 Scaling to Larger Cluster Configurations

The results we have presented so far were obtained for a 4-node cluster. In this section, we examine the impact of increasing the number of nodes on availability. Because large cluster systems in practice are usually divided into relatively small sub-clusters inside of which cooperation can be closer, there is little need to scale our results beyond 8 or 16 nodes.

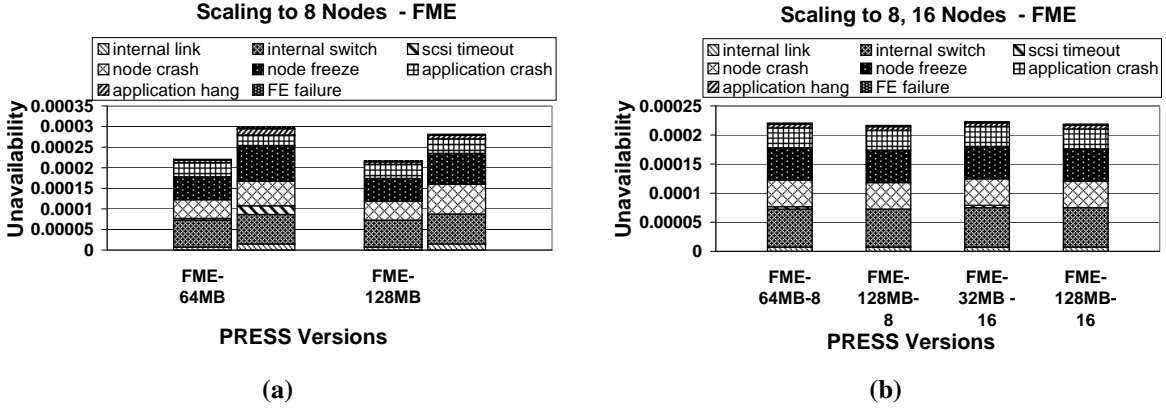


Figure 9: (a) Comparison of scaled model results based on measurements of a 4-node cluster to results obtained directly on an 8-node cluster for the FME version. The 64MB and 128 MB versions illustrate the results when the total cluster memory is kept constant and when it is scaled linearly, respectively. (b) Scaled model results for clusters of 8 and 16 nodes.

Attempting to accurately model the impact of scaling is a hard problem and involves defining exactly which aspects of the system are being scaled. For example, the memory size, storage bandwidth, and network bandwidth could be constant or scaled along with the number of nodes. Likewise, the intensity of the workload and the sizes of the data and working sets could increase or be held constant. Here, we make the simplifying assumptions that the bottleneck resource is the same for all cluster sizes and that the throughput of PRESS increases linearly with size.

With these assumptions, we can extrapolate our results according to a simple set of scaling rules. These rules define how the main parameters of our quantification model are affected by scaling. Essentially, the model depends on three types of parameters: the mean time to failure of each component ($MTTF_c$), the duration of each phase during the fault (D_c^p), and the (average) throughput under normal operation (T_n) and during each fault phase (T_c^p). These parameters are affected by scaling in different ways.

Let us refer to the $MTTF_c$ in a configuration with N nodes as $MTTF_c^N$. $MTTF_c$ in a configuration with S times more nodes, $MTTF_c^{SN}$, is $MTTF_c^N/S$. Given the assumptions mentioned above, the durations D_c^p should be the same under both configurations, and throughput should be $T_n^{SN} = S \times T_n^N$. Unfortunately, the effect of scaling on the throughput of each fault phase is not as straightforward. When the effect of the fault is to bring down a node or make it inaccessible, for instance, the throughput of phase C should approach $T_n^{SN} - T_n^N/SN$ under SN nodes. If throughput drops to 0 in phase A for N nodes, then it should also drop to 0 for SN nodes, so that the average throughput of phase B should approach $(T_n^{SN} - T_n^N/SN)/2$. If, on the other hand, throughput only drops to $T_n^N - T_n^N/N$ in phase A because requests can be rerouted (or dropped) to avoid backing up all the queues of the server, then we would expect the throughput to be $T_n^{SN} - T_n^N/SN$ for SN nodes;

in this case, phase B can be disconsidered. The same reasoning can be applied to phases F and G.

Figure 9(a) presents unavailability results for the FME version of PRESS for an 8-node cluster. In each group of bars, the left bar shows the unavailability achieved by applying our scaling rules to the 4-node measurements, whereas the bar on the right shows the unavailability achieved by applying our base model to actual 8-node measurements. We present results for when the total memory size is kept constant, with each node reserving 64 MBytes for PRESS, and when it is scaled with the number of nodes to 128 MBytes per node.

We can make two interesting observations from Figure 9. First, the unavailability of the FME version remains roughly constant, compared to the 4-node results, as our set of techniques alleviates the negative effects of cooperation. In fact, Figure 9(b) shows that unavailability stays roughly constant when we scale to 16 nodes as well. In contrast, Figure 10 shows that the unavailability of COOP increases linearly with cluster size, doubling at 8 nodes and then doubling again at 16 nodes. This suggests that our techniques scale well with cluster size and, in fact, become ever more critical to achieving high availability.

The second interesting observation is that our scaled modeling results based on measurements of a 4-node cluster are fairly accurate, although there is some divergence (up to 25%). This inaccuracy has more to do with the fact that we have decreased unavailability by an order of magnitude than with scaling the modeling results from 4 to 8 nodes. At 0.0003 unavailability, any inaccuracy arising from abstracting away details of the actual system leads to visible differences between scaled modeling results and those derived from actual measurements. Thus, we believe that our scaling rules are appropriate, at least for the cluster sizes and unavailabilities we consider. However, one must use caution when applying our scaling rules uniformly under different

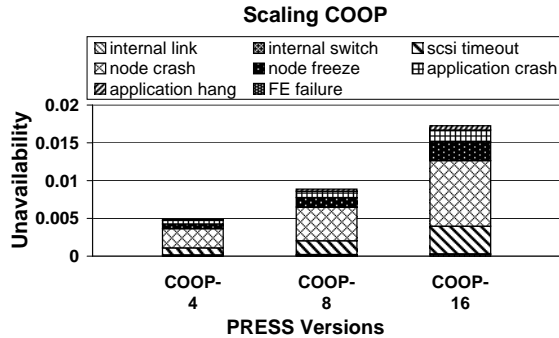


Figure 10: Scaled model results for the original COOP version on clusters of 8 and 16 nodes.

regimes. For example, under a constant data set size, scaling up the cluster will eventually cause the working set to fit in global memory. This effect will reduce the potential unavailability due to transient disk faults, as observed by the slightly lower unavailability of FME PRESS when running on 8 nodes, each with 128 MBytes. In this case, the difference is minor because our techniques are effective in drastically reducing the effect of disk faults on availability.

6.4 Summary of Results

Overall, our results show that it is possible to recover the availability properties of a cluster of independent servers, without losing the performance advantages of server cooperation. In fact, when independent servers are given extra hardware resources, both types of systems can achieve close to four-nines availability.

Another important observation is that one can achieve such results by exploiting off-the-shelf hardware and software with modest modifications. Table 2 lists the number of new (or modified) source code lines and the corresponding reductions in unavailability. The table shows that the source code changes required for an order-of-magnitude improvement in availability for the cooperative system was 1638 lines of code (excluding comments). Given that COOP has 15225 lines of uncommented code, our modifications are only an 11% change to the original code base. Further, the new components were implemented such that they can be reused as COTS components in other services.

Our results also show that no single high-availability technique can make the system highly available; rather, a combination of techniques is required.

Finally, our results demonstrate that increasing the cluster size has a serious negative effect on the availability of COOP, whereas it has little (if any) effect on the availability of the FME version. Thus, we expect that the set of techniques we have employed here will allow reasonable scaling of cooperative servers without significant loss of unavailability.

Enhancement	Additional NCSL	Unavailability Reduction
Membership	1307	65%
Queue Monitoring + Membership	1477	87%
Queue Monitoring + Membership + FME	1638	94%

Table 2: Comparison of effort, measured in non-commented source lines (NCSL), required to implement the techniques, with the reduction in unavailability over COOP.

7 Related Work

Researchers have studied the problem of fault tolerance extensively. A full treatment of this body of work is beyond the scope of this paper. Instead, we concentrate on efforts that have focused on improving the availability of cluster-based services. Of course, work analyzing how faults impact systems [12, 17, 29], as well as empirical measurement of actual fault rates [1, 14, 16, 21, 22], are necessary background for a model-based quantification effort such as ours.

Our methodology and infrastructure seem to be the first directed to quantifying the availability impact of a range of techniques as applied to cluster-based services. One of the first works on the subject [11] argued that there are irresolvable tradeoffs between availability, consistency, and performance in these services, but did not quantify the impact of these tradeoffs. Along the lines of consistent problem determination, [7] tries to automatically determine the location of faults in a distributed Web service. Other works studied performance and availability of a single-node COTS Apache Web server [18] and Oracle database [9]. These works are complementary to ours but are different in that they address neither the quantification of availability for cluster-based servers nor specific techniques for regaining availability in the context of cooperation.

In the traditional dependable systems context, researchers have explored constructing frameworks for high availability on commodity distributed systems [15]. Typically, these frameworks implement a range of high-availability mechanisms, such as voting, checkpointing, and heartbeats. A companion work examined the availability of the fault-tolerant framework itself [2]. In the fault injection context, one study [30] examined how hardware faults impacted the performance of common operating system operations such as disk reads. These works are complementary to ours in that such techniques are necessary for increasing availability and reasoning about the performance impact of faults. Our work expands on these studies by investigating the effects of violating the assumption, implicit in the above works, that all sub-systems share a unified fault model. In addition, none of these works examined how to provide high availability in

the context of a cooperative system where replicas use each other to increase performance; rather, they assumed replicated systems functioned completely independently.

The service architecture we studied, i.e. a locality-conscious Web server (PRESS) behind a load balancing front-end node, targets high performance without exposing the IP addresses of servers. Another approach to promoting cache locality with hidden addresses would have been to use front-end software that accepts connections and routes the requests to servers based on the hashed URLs (e.g., [8]). Replies would also go through the front-end on their way to clients. Unfortunately, this approach is not very scalable, since the front-end node can quickly become a bottleneck; vanilla load balancing front-ends such as the one we use can achieve much greater throughput.

The performance of PRESS was studied in [5, 6]. This paper extends the original studies by considering the availability impact of implementing cluster-wide resource sharing for high performance and how to regain the availability properties of non-cooperative servers. Even though we used PRESS in this study, our analysis also applies to other cooperative servers, such as cluster-based e-commerce servers or search engines. In fact, we have successfully applied our quantification methodology to a cluster-based on-line bookstore and a cluster-based auction service.

8 Conclusions

In this paper, we have used a methodology that combines fault-injection and analytic modeling to quantitatively evaluate the impact of several software and hardware high-availability techniques on the availability of a sophisticated cluster-based server. Overall, our results show that a number of techniques must be combined in order to significantly increase the availability of the server. However, our results are quite encouraging because they show that we can push a shared-resource cluster-based server into the four-nines availability regime by applying a small set of COTS hardware and software techniques in an evolutionary manner. We believe that these same observations apply to other cooperative servers. The question that we must address next is whether we can push these servers further, into the five-nines regime (the availability of the telephone system), by continuing this evolutionary approach or whether it will be necessary to re-think how the entire system is organized.

References

- [1] S. Asami. Reducing the Cost of System Administration of a Disk Storage System Built from Commodity Components. Technical Report CSD-00-1100, University of California, Berkeley, June 2000.
- [2] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIF) Environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(2), 2000.
- [3] R. Bianchini and E. V. Carrera. Analytical and Experimental Evaluation of Cluster-Based WWW Servers. *World Wide Web Journal*, 3(4):215–229, December 2000.
- [4] E. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, July/August 2001.
- [5] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.
- [6] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.
- [7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, Washington, DC, June 2002.
- [8] Cisco CSS 11500 Series Content Services Switches, Apr. 2003. Available at <http://www.cisco.com/en/US/products/hw/contnetw/ps792/index.html>.
- [9] D. Costa, T. Rilho, and H. Madeira. Joint Evaluation of Performance and Robustness of a COTS DBMS Through Fault-Injection. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2000)*, New York, NY, June 2000.
- [10] F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. 1995.
- [11] A. Fox and E. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of Hot Topics in Operating Systems (HotOS VII)*, Rio Rico, AZ, Mar. 1999.
- [12] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct. 1990.
- [13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.
- [14] T. Heath, R. Martin, and T. D. Nguyen. Improving Cluster Availability Using Workstation Validation. In *Proceedings of the ACM SIGMETRICS 2002*, Marina Del Rey, CA, June 2002.
- [15] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.

- [16] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th Symposium on Reliable and Distributed Systems (SRDS '99)*, 1999.
- [17] I. Lee and R. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *Proceedings of International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 20–29, 1993.
- [18] L. Li, K. Vaidyanathan, and K. S. Trivedi. An Approach for Estimation of Software Aging in a Web Server. In *Proceedings of the International Symposium on Empirical Software Engineering, ISESE 2002*, Oct. 2002.
- [19] X. Li, R. P. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, Jan. 2002.
- [20] Linux virtual server project. <http://www.linuxvirtualserver.org/>.
- [21] D. D. E. Long, J. L. Carroll, and C. J. Park. A Study of the Reliability of Internet Sites. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 177–186, Sept. 1991.
- [22] B. Murphy and B. Levidow. Windows 2000 Dependability. (MSR-TR-2000-56), June 2000.
- [23] K. Nagaraja, R. Bianchini, R. Martin, and T. D. Nguyen. Using Fault Model Enforcement to Improve Availability. In *Proceedings of the Second Workshop on Evaluating and Architecting System dependability (EASY)*, Oct. 2002.
- [24] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, Mar. 2003.
- [25] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.
- [26] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD*, pages 109–116, Chicago, IL, June 1988.
- [27] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charlston, SC, Dec. 1999.
- [28] Service Monitoring Daemon, Apr. 2003. Available at <http://www.kernel.org/software/mon/>.
- [29] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, Montreal, Canada, 1991.
- [30] T. K. Tsai, R. K. Iyer, and D. Jewitt. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.
- [31] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.