Quantifying and Improving the Availability of Cooperative 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
110 Frelinghuysen Rd, Piscataway, NJ 08854

Department of Computer Science Technical Report DCS-TR-517

January, 2003

Abstract

Much research has shown that cluster-based servers can substantially increase performance when nodes cooperate to share and globally manage their resources. In this paper, we apply a quantification methodology to show that this performance increase has a corresponding substantial cost in availability. Specifically, we show that a sophisticated cluster-based web server gains a factor of 3 in performance when nodes cooperate to balance load and jointly manage their memories, but also suffers an increase in unavailability of a factor of 10. We then show how this web server can be augmented with Commercial Off-The-Shelf (COTS) 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 predicted availability of close to 99.99%.

1 Introduction

Popular Internet services frequently rely on clusters of commodity computers as their supporting infrastructure [6]. 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., [1, 6, 7]. In contrast, understanding designs for availability, behavior during component faults, and the relationship between performance and availability of these servers have received much less attention. This is not to say that service designers are oblivious to the importance of high availability. In fact, services are designed and implemented to tolerate faults at many levels [6, 15, 29]. Unfortunately, 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 [26] to quantify the improvement in availability of PRESS, a sophisticated cluster-based web server, 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. Results from this study 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 the behavior of 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; the same independent version with a front-end device to hide node and application faults from end-users together with an extra server node; and the base cooperative

version of PRESS. 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, a critical challenge we address in this work, using PRESS as a case study, is how cluster-based servers can be designed for high availability in the presence of resources that are cooperatively managed and globally shared for high performance. While we focus specifically on the PRESS server here, our results should be applicable to a wide range of cluster-based Internet services such as e-commerce servers or search engines. These servers must typically employ cooperation within sub-clusters for performance and scalability, as in PRESS, and/or across multiple sub-clusters 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 for the server nodes, 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.

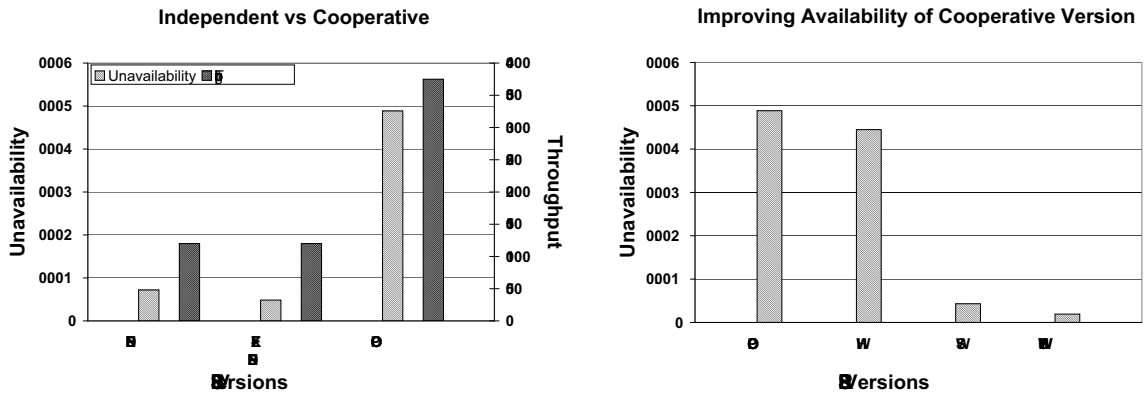
Thus, while the techniques that we apply are well-known, a problem we address is how to merge independent, but possibly overlapping, fault semantics of the different COTS components implementing these techniques. The problem goes beyond the typical treatment of consensus in Byzantine fault tolerance techniques [5, 9] because often the very definition of correct service differs between components. For example, the front-end device, the membership module, and the application server itself may all have different definitions for node availability. Surprisingly, no single definition adequately captures the system state that is necessary to provide a highly available service. As a result, these diverging fault models can lead to inconsistent recovery actions.

To address this problem, we implement a novel technique called Fault Model Enforcement (FME) [25] that can be leveraged to address these discrepancies. The key idea behind FME is that it is too difficult to build a complex cluster-based system that is tolerant to all possible faults. Thus, service designers should define a simplified abstract fault model that the service will tolerate. Then, at runtime, the infrastructure will enforce this abstract fault model, translating any fault that is not in the model to one that is. Essentially, FME defines a single fault model that all components have to adhere to.

To explore whether the above set of high-availability techniques could significantly improve the availability of PRESS, we initially used our methodology to estimate their impact. 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 above software techniques are applied, and when both software techniques and additional hardware are applied. The unavailability result for the base PRESS system was obtained using actual executions and our quantification methodology, whereas the other three results are analytical extrapolations from the base result. These results suggest that, at least theoretically, it is possible to regain the availability of the original non-cooperative version while retaining the performance advantages of cooperation. 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, we can come quite close to 99.99% availability.

In summary, we make the following contributions:

- We show that while cluster-wide cooperation for load balancing and resource management can significantly increase performance, it also increases unavailability. The original availability, however, can be recovered with the application of a relatively small number of high-availability techniques. Indeed, these techniques together with



(a) **(b)**

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.*

a small amount of hardware redundancy and sophistication is sufficient to bring a read-only cluster-based server close to four nines of 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.

- We show how the novel FME technique can be applied to an actual cluster-based server and illustrate its power in 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 [26]. 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 predict 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 metric, 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.

2.1 Phase 1: Measuring Performance and Availability 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 measures the service's performance and availability as a single fault of each type and the subsequent recovery are injected into a live service. 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 model used in the modeling phase. The one exception to this guideline is that a server may not exhibit all model stages under certain faults. In these cases, the evaluator must use his understanding of the server to correctly determine which stages are missing (and later setting the time of the stage in the model 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.

2.2 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 aver-

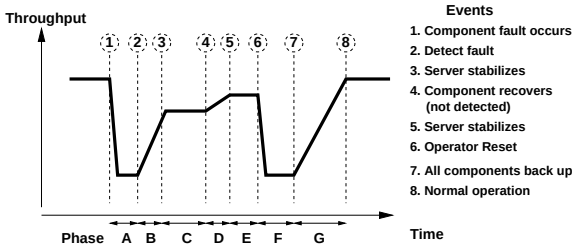


Figure 2: *The 7-stage piece-wise linear model specified by our methodology for evaluating the performability of cluster-based servers.*

age throughput and availability, combining the server's behavior under normal operation, the behavior during component faults, and the rates of fault and repair of each component. The model is built in two parts. The first part describes the system's response to a single fault of each different type in a 7-stage piece-wise linear model. The second part combines the effects of each fault type along with the MTTR and MTTF of each component to arrive at an overall average availability and performance.

Per-fault seven-stage model. Figure 2 illustrates our 7-stage model. Time is shown on the X-axis and throughput is shown on the Y-axis. This model 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 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. These parameters are either measured in phase 1 or an assumed environmental value that must be supplied by the evaluators. For example, the time that a service will remain in stage B is typically measured; the time for stages D + 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 time the system is in such a state to zero.

Overall availability and performance. Having defined the server's response to each fault, we now must combine all these effects into an average performance and average availability metric. To simplify the analysis, we assume that faults of different components are not correlated, fault arrivals are exponentially distributed, 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. If T_n is the server throughput under normal operation, c the faulty component, 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]

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. While this model is obviously not general enough to describe all cluster-based servers, we believe that it is representative of a large class of servers, such as front-end servers (including PRESS) and other read-only servers. Our model also does not consider correlated failures, which may lead to optimistic predictions [33]. However, since there is little quantitative

[1]We refer the reader to [26] for a discussion of why the denominator is correctly $MTTF_c$ rather than $MTTF_c + MTTR_c$.

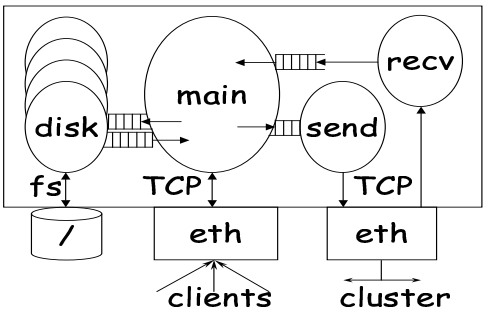


Figure 3: *Basic PRESS architecture.*

data on how faults are correlated, we have made the explicit decision of keeping our model simple and only address single faults.

3 The PRESS Server

PRESS is a highly optimized yet portable cluster-based locality-conscious Web server that has been shown to provide good performance in a wide range of scenarios [7, 8]. Like other locality-conscious servers [27, 1, 4], 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.

Communication 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 as we shall see, PRESS can also be configured to work with an intelligent 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 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 id 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. 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 is splintered into two sub-clusters, one with 3 nodes and one with 1 node, and is unable to re-integrate because the faulty node did not crash, violating the fault model assumed by the designers of PRESS. Thus, the server's performance is sub-optimal, even after the component has been repaired, until an

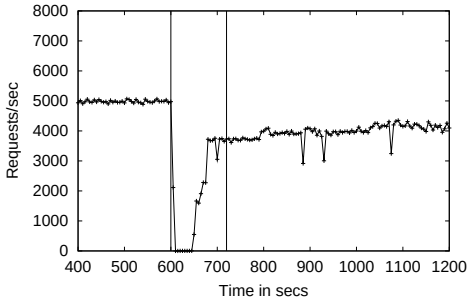


Figure 4: *Throughput of PRESS running on 4 nodes when a disk fault is injected. The vertical lines denote the period the fault is active.*

operator can reset it by restarting the singleton subcluster.

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. There are three advantages to such an approach. First, as is well-known, it is easier to correctly design, implement, and test small subsystems rather than a large monolithic system. Second, these subsystems can be individually reused in other servers, for which the entire set of techniques we employ here may not be needed or appropriate. Finally, this seems to be the most common approach to service design in the industry today, where layered services are composed from COTS subsystems produced by many different organizations.

Our adoption of this approach led to an interesting problem. When different but related availability techniques are implemented in separate subsystems, their view of the system state may overlap. Moreover, because 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 our novel Fault Model Enforcement (FME) [25] technique provides one approach to 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) [22] 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[2]; 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, 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 or goes down. This action is the addition/deletion of entries in the table used by the front-end to distribute requests.

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 noticeable loss in throughput.

4.2 Group Membership

As already discussed, a major cause of unavailability for PRESS is the splintering of the cooperating cluster and its inability to subsequently rejoin all nodes into

[2]LVS actually supports three different methods of forwarding client packets. We chose IP tunneling because it gives the best scalability.

a single group, even after the fault has been repaired. 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 gracefully from such splintering once the underlying fault has been repaired.

Specifically, we implement a variation of the three-round membership algorithm described in [12]. 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.

The above 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 also *not* be able to communicate to 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. The use of `NodeDown()` will be explained in more detail below.

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 when called by the membership client thread. This membership infrastructure obviates the need for heartbeats in the original version of PRESS, so we eliminated the associated code from it.

4.3 Queue Monitoring

While heartbeat monitoring is a popular technique, it is far from sufficient. The major 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 stoppage of service, 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 faults in subsystems, particularly transient ones, may never turn into errors at the application level, if the application does not make use of that subsystem. For example, a temporary disk hang may never affect PRESS if there are no misses out of the memory cache during that time. 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 that are connected by event or task queues [35, 7].

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. When the queue build-up exceeds a predefined threshold, we initiate a recovery by simply removing the faulty node from the cooperation set.

4.4 Combining Group Membership and Queue Monitoring

Neither group membership nor queue monitoring is sufficient by itself. Unfortunately, when these techniques are combined, inconsistencies can arise from

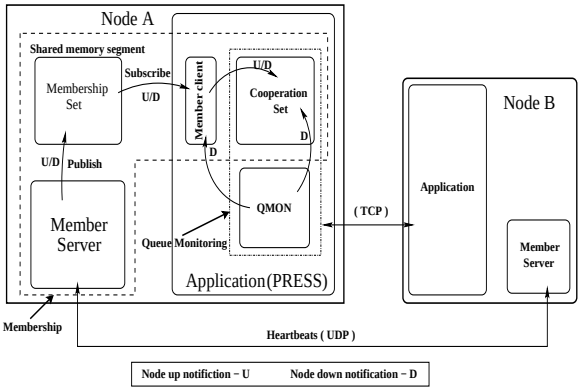


Figure 5: *External membership and application queue monitoring interactions.*

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 group membership and the queue monitoring mechanisms diverge.

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 because it has more specific information about the service, 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 components. 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. Figure 5 shows the architecture of the membership service, the queue monitoring mechanism, and their combination. Without FME, we do not attempt to resolve potential conflicts; when it detects that a peer p has failed, the queue-monitoring code removes p from the coordination set and uses the `NodeDown()` function to tell the membership service of this action. The next time the membership thread checks the shared memory segment, if the membership server believes that p is in the group, it simply calls `NodeIn()` with p as a parameter. Thus, for the duration of such conflicts, performance suffers as the affected node(s) keep entering and leaving the cooperation set.

4.5 Fault Model Enforcement

The idea behind *Fault Model Enforcement* (FME) is to enforce an abstract fault model (defined by the service designers) 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, 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 and can contribute to the server's performance. Although somewhat extreme, FME effectively simplifies the design of highly available services by allowing designers to concentrate on tolerating a specific (and simple) set of faults. Also, as shall be seen, 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.

The designers of PRESS implemented a simple fault model originally: only node and application crashes/restarts were handled. By adding group membership and queue monitoring, we have added the ability to handle unreachable and lack-of-progress faults. Thus, these are the faults that define the fault model we must enforce with FME.

It is beyond the scope of this paper to explore FME's full potential. Here, we aim solely to demonstrate the power and flexibility of an FME approach by using it to reconcile inconsistencies between the group membership and queue-monitoring mechanisms, automatically restart hung application processes, and to allow PRESS to recover more rapidly from disk faults. In particular, we implement an external FME process that periodically (i) monitors the local disk, and (ii) probes the service by sending simple HTTP requests to it. The latter allows FME to restart an application process if it gets hung because of a transient fault. The former allows FME to resolve conflicts between queue-monitoring and group membership when they diverge because of a disk fault. It also adds complementary external resource monitoring, allowing PRESS to detect disk faults faster than the already implemented queue-monitoring-based detection on peer processes.

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. If the application fails to respond to FME's probes but FME believes all subsystems to be operational, FME will restart the application process.

5 Experimental Environment

Measurements shown in the next section were taken on a cluster of 800 MHz Pentium III PCs, each booted with with 206 MB of memory and 2 10K RPM 9 GB SCSI disks. Nodes are interconnected by a 1 Gb/s cLAN network. PRESS was allocated 128 MB on each node for its file cache; the remainder of the memory was sufficient for the operating system. In our experiments, PRESS only serves static content and the entire set of documents is replicated at each node on one of the disks. 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 a connection cannot be completed and to time out after 6 seconds if, after successful connection, the request cannot be completed. The request stream follows a trace gathered at Rutgers; we chose this trace from several that Carrera and Bianchini previously used to evaluate PRESS's performance because it has the largest working set [7]. We make 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, once every 2 weeks. We derived the rates from previous works which empirically observed the fault rates of many systems [2, 16, 23, 18, 24]. We use Mendosus [21] 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.

Fault	**MTTF**	**MTTR**
Link down	6 months	3 minutes
Switch down	1 year	1 hour
SCSI timeout	1 year	1 hour
Node crash	2 weeks	3 minutes
Node freeze	2 weeks	3 minutes
Application crash	2 months	3 minutes
Application hang	2 months	3 minutes
Front-end failure	6 months	3 minutes

Table 1: *Failures and their MTTFs and MTTRs. Application hang and crash together represent an MTTF of 1 month for application failures.*

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 fault-detection threshold of either 512 messages of all types (i.e., the entire queue is full) or 384 request messages.

6 Quantifying Expected Availability

In this section, we present the resulting improvement of availability due to our techniques. We begin by examining the impact of applying the traditional front-end and extra computing resources. Using measurements of the basic PRESS system and modeling, we argue that these traditional techniques have little impact on availability when the back-end nodes must closely cooperate to maximize cluster performance. 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 on our actual implementations of these techniques as described in Section 4. We show that the combination of these two techniques decreases unavailability by 83% over the basic version. Finally, we show that the application of

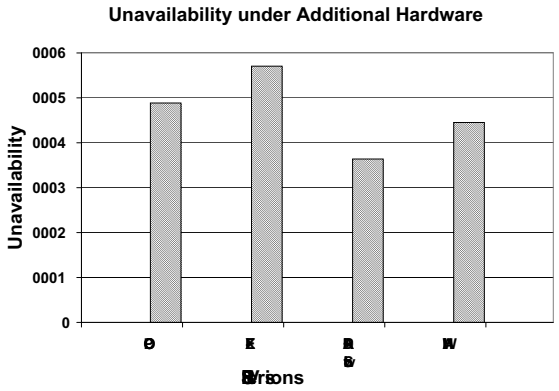


Figure 6: *Effect of adding redundant hardware on the unavailability of the base version of PRESS.*

our simple FME strategy further reduces unavailability, reaching an unavailability that is 88% lower than that of the basic version. This combination of techniques achieves a lower expected unavailability than a set of independent servers, while obtaining the performance of the shared-resource version.

To completement our experimental study, we also uses our model to compute the expected availability of a system that includes all of the techniques described in Section 4 as well as the addition of RAIDs and extra network switches. We show that the expected reduction in disk and network fault-rates using redundant hardware can push the expect availability regime of a shared resource system into the four nines availability class. We have strong confidence in our analytic models given that these models closely track the results obtained using actual measurements of a live system running on a fault-injection infrastructure as described in Section 2.

6.1 Basic Results

Extra Hardware. Figure 6 shows the modeled impact of adding a pair of front-ends and extra capacity in the form of one additional server (FE-X), RAIDs and an extra switch (RAID+switch), and both sets of resources together (All HW) to the basic 4-node version (COOP) of the web server. 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 node failures in the system. Yet, the front-end is ineffective at masking node failures because the back-end servers rely on each other. Thus, 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 code, will attempt to access resources on the failed node which leads to the server stalling as a whole. 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 RAIDs 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 years [3]. The models show that the impact of this additional hardware does not change the availability class of the system: adding 4 RAIDs 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 which halt a node or prevent communication to it: link errors, node

[3] We modeled the MTTF improvement of a composite system in terms of the number of components, N, and their MTTF and MTTR: $MTTF_{new} = \frac{(MTTF)^2}{N(N-1)(MTTR)}$ [28].

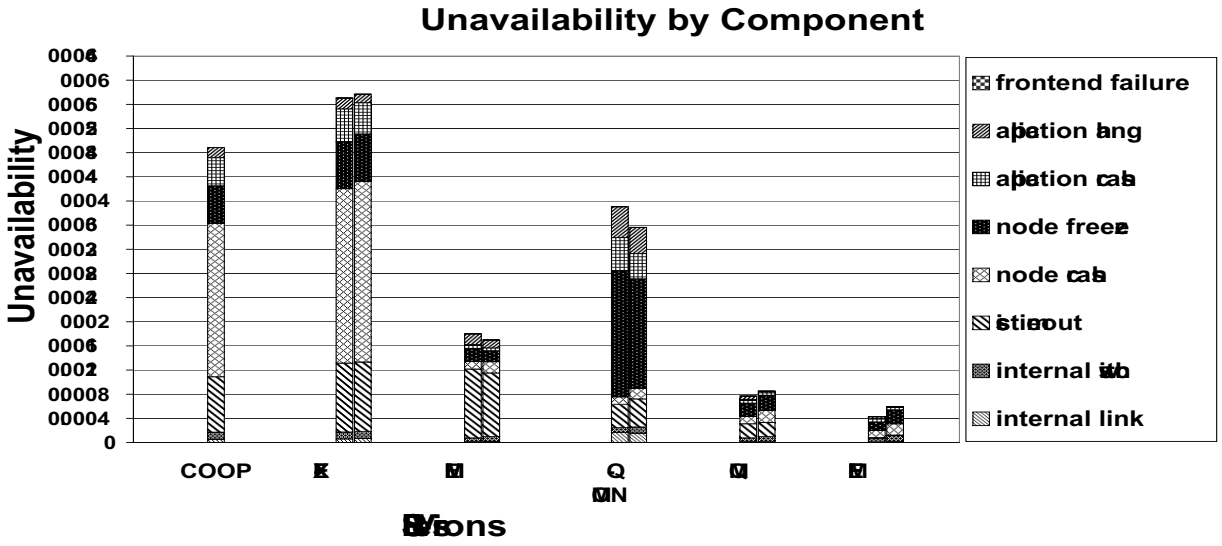


Figure 7: *Comparison of experimental and modeled values for unavailability of PRESS when various high-availability techniques (and combinations thereof) are applied.*

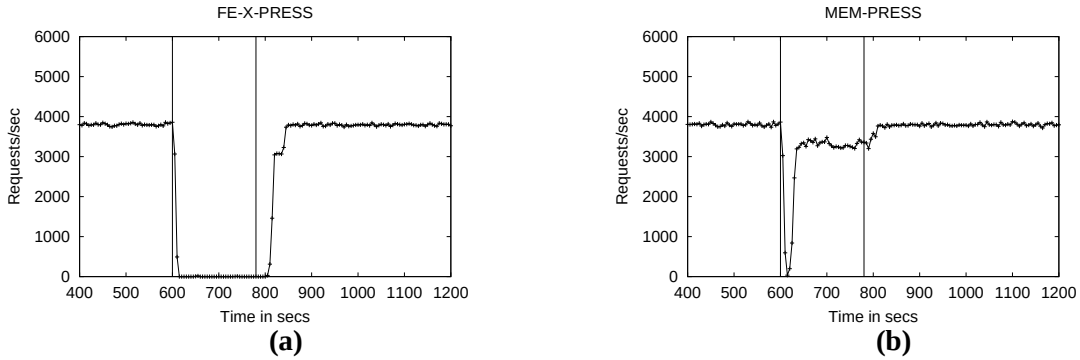


Figure 8: *Throughput of PRESS as a link fault and subsequent repair is injected (a) without membership, and (b) with membership implemented.*

freeze, and node crash errors. Figure 8 illustrates the effect of a link error on the throughput of PRESS with (b) and without (a) the group membership service. The vertical lines delimit the duration of the error. The graphs clearly show that the membership service quickly detects that a node becomes unreachable, allowing the server to adjust its request distribution accordingly. However, the MEM enhancement cannot handle SCSI errors because these only freeze threads accessing the disk. In general, any errors which stop the application without crashing the system cannot be detected with this method. In spite of these shortcomings, however, 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 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, in each of the above cases nodes fail to re-integrate upon recovery.

Combining these two approaches, the MQ version shows a significant improvement in unavailability over the COOP version, reducing unavailability by 83%. Because the system state view of each of these versions is combined into a single view, the result is that the system can handle all errors in some way. However, there is still room for improvement since node freeze and SCSI faults lead to conflicts between the two mecha-

nisms, with the faulty node cycles between leaving and re-entering the cooperation set. (In fact, application hangs also cause the same problem but since the MTTF for this fault is much larger than node freeze and SCSI fault, its impact is minimal.)

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, node freezes, and application hangs, as these errors are turned into node and application crashes. The methods the PRESS server uses to handle these errors is fundamentally the same as in the MQ version, however, as detailed in Section 4, a small external subsystem simply causes one set of faults to become another.

By tackling these errors, the FME version mitigates most of the negative effects of cooperation between nodes. As a result, the availability benefits of the front-end node and the extra server capacity start to show more clearly. In fact, unavailability increases by almost 50% if we now remove these additional resources.

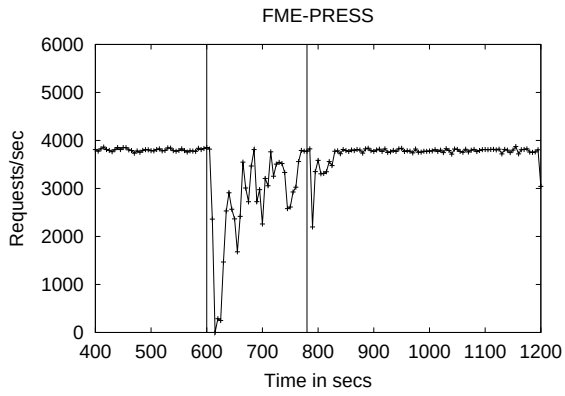


Figure 9: *Throughput of the FME version during a node freeze.*

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. Figure 9 shows the performance of the FME version when a node freeze fault is injected. Observe that after FME crashes the faulty node, the total number of requests served per second oscillates wildly, with an average throughput of 3200 requests/sec. This average is less than the 3800 requests/sec expected by a 4 workstation system. We suspect this is 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 thrashing effect.

6.2 Modeling Other Approaches

Figure 10 shows the expected unavailability (computed by modeling from the experimental COOP results) of three additional versions of the PRESS server. The same FME unavailability as in Figure 7 is shown on the left for reference. The C-MON version is the same as 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. The modeled results show that C-MON represents a significant improvement, cutting unavailability by almost 50% compared to the FME version.

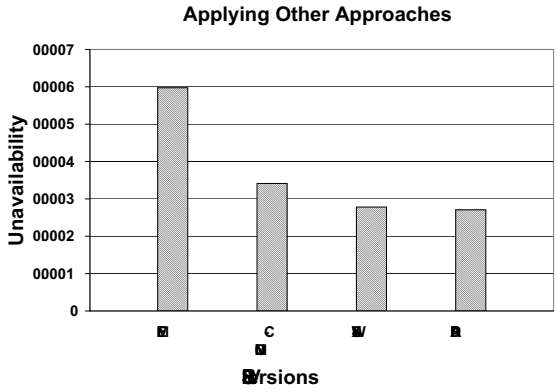


Figure 10: *Results of combining all approaches with connection monitoring by the front-end node and additional hardware resources.*

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 slightly below 0.0003%. 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 (>

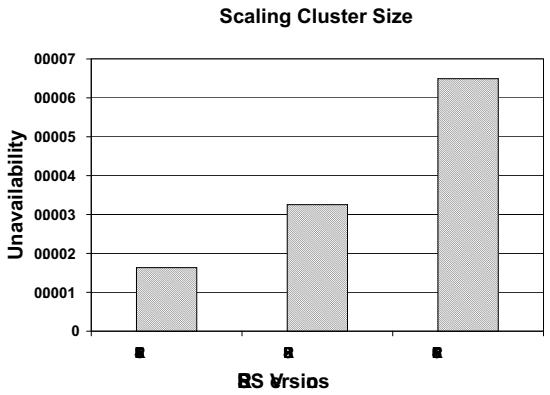


Figure 11: *Modeled unavailability of COOP for increasing cluster size. Each configuration is equipped with a front-end node and 25% extra capacity. The only source of unavailability is modeled as 15 seconds of throughput zero for each fault.*

99.97% or 0.9997) 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 system. In this section, we examine the impact of increasing the number of nodes on availability. Because large cluster systems in real life are usually divided into relatively small sub-clusters, inside of which cooperation can be closer, there is little practical need to scale our results beyond 16 or 32 nodes.

Let us start the discussion by considering the basic version of PRESS, COOP. As we increase the size of the cluster, the maximum throughput of the system and the probability of faults grow linearly with it. Under these circumstances, the availability of the system should stay constant. However, recall that many faults reduce the throughput of this version to 0 until the fault is detected and the system recovers. This means that the number of requests lost due to faults can also potentially grow linearly with the size of the cluster. Thus, unavailability can increase linearly with cluster size in the worst case.

Figure 11 shows that this indeed happens. The figure plots the expected unavailability of COOP with a front-end node and 25% extra capacity, as a function of cluster size, for the fault load in Table 1. To simplify the analysis, we assume that all faults cause throughput to go to zero for 15 seconds. This simple assumption is pessimistic in that all faults cause zero throughput, but optimistic in that it ignores recovery periods requiring warm-up effects, as well as operation intervention.

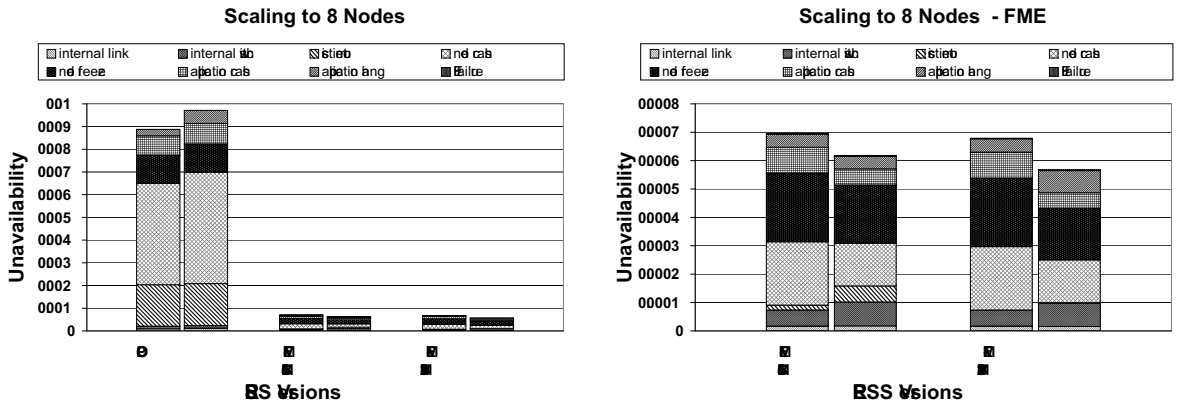
These results suggest that quickly increasing unavailability is an important reason for sub-clustering, when cooperation is exploited. Furthermore, these results suggest that applying high availability techniques is even more critical as we increase cluster size.

The results from this simple modeling exercise are enlightening, but not very accurate. Attempting to accurately model the impact of scaling is much harder 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 this assumption, 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 unaccessible, for instance, the throughput of phase C should approach $T_n^{SN} - T_n^{SN}/SN$ under SN nodes, whereas the average throughput of phases B and G should approach $(T_n^{SN} - T_n^{SN}/SN)/2$ and $T_n^{SN}/2$ respectively. Just as under N nodes, the throughput of phases A and F should approach 0 under SN nodes.

Figure 12(a) and (b) present unavailability results for the COOP and FME versions of PRESS for an 8-node



(a) **(b)**

Figure 12: *(a) Scaling results showing the unavailability for the base COOP and the FME versions for a 8-node configuration. 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) Zoom of the FME results.*

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 these figures. The first is that the unavailability of COOP roughly doubles in comparison to the 4-node results. We predicted this effect with the simple extrapolation of Figure 11. In contrast, 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. This suggests that FME should scale very well to larger systems, and that its benefits should increase with cluster size.

The second interesting observation is that our modeling based on 4-node measurements leads to fairly accurate results. This suggests that our scaling rules are appropriate, at least for the relatively small cluster sizes we are interested in. However, one must use caution when applying our scaling rules uniformly under different scaling 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.

Enhancement	Additional NCSL	Unavailability Reduction
Membership	1307	71%
Queue Monitoring	138	38%
Queue Monitoring + Membership	1445	83%
Queue Monitoring + Membership + FME	1596	88%

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

6.4 Summary of Results

Overall, our results show that it is possible to recover (and even improve on) the availability properties of a cluster of independent servers, without losing the performance advantages of server cooperation. In fact, it is possible to achieve close to four nines availability without massive replication of resources.

Another important observation is that one can achieve such results by composing the system out of 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 entire system was 1596 lines of code (excluding comments). Given that the COOP version of is PRESS has 15225 lines of un-

commented code, our modifications are only an 11% change to the original code base. Further, the new components were implemented such that they themselves can be reused as COTS components in other services.

Our results also show there is no silver-bullet: using any single high-availability technique cannot make the entire system highly available; rather, a combination of techniques is required.

Finally, our results demonstrated that increasing the cluster size has a tremendous 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.

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 [14, 19, 31, 32], as well as empirical measurement of actual fault rates [2, 16, 23, 18, 24], 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 [13] argued that there are irresolvable tradeoffs between availability, consistency, and performance in these services, but did not quantify the impact of these tradeoffs. A more recent work developed models which can be used to quantify the impact of faulty components on quality of service metrics [30]. Along the lines of consistent problem determination, [10] 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 [20] and Oracle database [11]. These works are complementary to ours but are different in that they do not address the quantification of availability for cooperative cluster-based servers nor specific techniques for regaining availability in the context of cooperation.

In the traditional dependable systems context, recent work explored constructing frameworks for high availability on commodity distributed systems [17]. 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 [3]. In the fault injection context, one study [34] 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 faults' performance impact. 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 availably in the context of a cooperative system where replicas use each other to increase performance; rather, they assumed replicated systems functioned completely independently.

The performance of PRESS was first studied at the last PPoPP [7]. This paper extends the original study 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 should also apply to other cooperative servers, such as cluster-based e-commerce servers or search engines.

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. We first observed that a server that globally manages and shares resources across the cluster can improve performance significantly, but only at the cost of significant loss of availability. We found that the accepted approach of adding a front-end device and redundant hardware failed to significantly impact system availability. These techniques simply do not address the fundamental cause of unavailability: the cooperation between nodes for high performance introduces inter-node dependency that often allows the effects of a single fault to propagate throughout the entire cluster. In essence, this inter-node dependency makes the most unavailable component the bottleneck for overall system availability. Thus, adding hardware that only reduces the fault-rate of some of the components, such as RAIDs and redundant networks, failed to have significant overall im-

pact because they simply move the bottleneck for system availability to the next most unreliable component. In fact, adding redundant capacity can have a negative effect on availability, as this increases the MTTF of the cluster.

Taking the next logical step, we applied a set of techniques to remove resources from the global pool as they fail and re-integrate them once they have been repaired. We took the approach of applying these techniques in the form of COTS modules that are implemented separately rather than as a monolithic system, realizing that most services are built from pre-existing subsystems. For example, we leveraged LVS, an existing Linux-based front-end device, as a COTS module. We found this strategy more difficult to apply in practice than it might appear. The key problem was that, although each of the techniques works in isolation, they had slightly different methods of detecting and recovering from faults. These differences can sometimes lead to inconsistent recovery actions.

As a result, we found that it is critical to integrate the multiple overlapping views of the system state provided by each of these mechanisms. We argue that simply ordering the views, that is, give precedence to the views, is not sufficient. Instead, we applied a novel technique called FME to this conflict resolution problem. We use FME to translate faults that lead to inconsistent views (and so possibly conflicting recovery actions) to faults that are treated consistently across the multiple techniques. FME can also be used to reduce the need for operator intervention by translating unexpected faults to those from which the service can automatically recover. Our initial implementation shows that FME is a simple and tractable approach in practice; we reduced unavailability significantly at the cost of only a few hundred lines of code.

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. The question that we must address next, of course, 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] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[2] 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.

[3] 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.

[4] 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.

[5] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, 1993.

[6] E. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, July/August 2001.

[7] 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.

[8] 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.

[9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, Feb. 1999.

[10] 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.

[11] 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.

[12] F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. 1995.

[13] 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.

[14] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct. 1990.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Linux virtual server project. http://www.linuxvirtualserver.org/.

[23] 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.

[24] B. Murphy and B. Levidow. Windows 2000 Dependability. (MSR-TR-2000-56), June 2000.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *to appear in Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, Decemeber 2002.

[31] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems : Design and Evaluation (third edition)*. A. K. Peters, Ltd., 1997.

[32] 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.

[33] D. Tang and R. K. Iyer. Analysis and Modeling of Correlated Failures in Multicomputer Systems. *IEEE Transactions on Computers*, 41(5):567–577, May 1992.

[34] 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.

[35] 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.