Using Fault Injection to Evaluate the Performability of Cluster-Based Services

Kiran Nagaraja, Xiaoyan Li, Bin Zhang, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen
{*knagaraj, xili, binzhang, ricardob, rmartin, tdnguyen*}*@cs.rutgers.edu*

Technical Report DCS-TR-491
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

May 17, 2002
Revised August 21, 2002

Abstract. *We propose a two-phase methodology for quantifying the performability (performance + availability) of cluster-based Internet services. In the first phase, evaluators use a fault-injection infrastructure to measure the impact of faults on the server's performance. In the second phase, evaluators use an analytical model to combine an expected fault load with measurements from the first phase to assess the server's performability. Using this model, evaluators can study the server's sensitivity to different design decisions, fault rates and other environmental factors. To demonstrate our methodology, we study the performability of 4 versions of the PRESS web server against 5 classes of faults. We use Mendosus, a new fault-injection and network emulation infrastructure, to effect phase 1 of our methodology. We then use our model to quantify the gain or loss in performability as PRESS was modified for increasing performance. We also use our model to study the impact of reducing live operator support and adding RAIDs on PRESS's performability.*

1 Introduction

Popular Internet services frequently rely on large clusters of commodity computers as their supporting infrastructure [5]. These services must exhibit several important characteristics, including high performance, scalability, and availability. The performance and scalability of cluster-based servers have been studied extensively in the literature, e.g., [2, 5, 8]. In contrast, understanding designs for availability, behavior during component failures, and the relationship between performance and availability of these servers have received much less attention.

Two immediate goals of our work are: (1) to develop a methodology and metrics for evaluating and quantifying the expected behaviors of cluster-based servers under realistic deployment conditions, where faults are an unavoidable fact of life, (2) to thoroughly understand the behavior of cluster-based servers in the presence of component failures and define a server design and evaluation methodology that can produce extremely available and high performing cluster-based servers. Our longer term goal is to enable the development of Internet services that achieve the same level of availability as services that are so available that we take them for granted, such as the wired telephone system.

Today's service designers are not oblivious of the importance of high availability. In fact, services are currently designed and implemented to tolerate faults at many levels [5, 12, 23]. Unfortunately, the design and evaluation of service availability is still an art based on the practitioner's experience and intuition rather than a scientific methodology. Further, the difficulty of putting together a representative test-bed as well as a comprehensive fault-injection infrastructure often forces designers to resort to *ad hoc* and gross fault-injection techniques such as unplugging components by hand [18]. Such *ad hoc* techniques are undesirable because they can lead to either a false sense of confidence in the system's robustness or wasted resources in over-engineered systems.

We advance the state-of-the-art by developing a methodology for studying and quantifying the performability—as shall be seen below, performability is a function of performance and availability—of cluster-based servers and a companion infrastructure for fault injection. In particular, we first present a 2-phased methodology for server design and evaluation that uses fault-injection and analytic modeling to focus on availability as well as performance. The first phase of the

methodology begins with an exploration of the actual impact of various faults on a server in isolation. Even a single failure can have a tremendous impact on server behavior. For example, major outages can be caused by the failure of a single disk, such as the one that holds the operating system [3]. Our approach is to inject a single fault into each subsystem in turn and observe the server's live response. By emulating live faults, as opposed to using simulation or analytic modeling, we can observe actual system interactions, and characterize the performance during faults as well as recovery. In essence, in the first phase, we micro-benchmark the server for both availability and performance in the presence of faults, rather than the more traditional approach of micro-benchmarking performance in the absence of faults.

The second phase uses an analytic model to combine an expected fault model [21, 28], measurements from the first phase, and parameters of the expected deployment environment to predict performability. Designers can use this model to study the potential impact of different design decisions on the server's behavior. For example, designers can study the impact on availability if code was added to more quickly detect and recover from some particular fault. We introduce a single performability measure to enable designers to easily characterize and compare servers in terms of both their performance and availability.

We next introduce Mendosus, a fault-injection and network emulation infrastructure designed to support the first phase of our methodology. Mendosus is novel in that it combines network emulation, which eases the problem of evaluating servers' behaviors on different platforms, and fault-injection using fault models geared particularly toward cluster-based services.

Finally, to show the practicality of our methodology, we use it to study the performability of PRESS, a cluster-based web server [8]. A significant benefit of analyzing PRESS is that, over time, the designers of PRESS have accumulated a number of different versions with increasing performance. Using our methodology, we were able to quantify the impact of changes from one version to another on availability, and therefore, performability, giving a more complete picture than just the previous data on performance. For example, a PRESS version using TCP achieved a higher overall performability score even though it does not perform as well as a version using VIA. We also show how our model can be used to study the hypothetical impact of design or environmental changes; in particular, we use our model to study PRESS's sensitivity to operator coverage and using RAIDs instead of independent SCSI disks.

We make the following contributions:

- We propose a methodology that combines fault injection, experimentation, and modeling to quantitatively evaluate servers' availability as well as performance.
- We present a comprehensive infrastructure for fault injection into servers. We hope to make Mendosus available to the community in the near future.
- We evaluate the performance and availability characteristics of several versions of a sophisticated cluster-based server to show the applicability of our methodology. Using this evaluation, we also draw several conclusions about how servers should be designed.

2 Methodology and Metric

As already mentioned, our methodology for evaluating servers' performability is comprised of two phases. In the first phase, the evaluator defines the set of all possible faults, then injects them (and the subsequent recovery) one at a time into a running system. During the fault and recovery periods, the evaluator must quantify throughput and availability as a function of time. For servers such as PRESS, the *throughput metric* is *the number of requests successfully served per second* and *availability* is *the percentage of requests served successfully*. In the second phase, the evaluator uses an analytic 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 and repair of each component. To parameterize this model, the evaluator defines an expected fault load in terms of the Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR) for each fault. Also, for each fault, the evaluator maps the measurements acquired in phase 1 for that fault to a 7-stage piece-wise linear model of performance over time.

A current limitation of our model is that it does not capture data integrity errors; that is, errors that lead to incorrect data being served to clients. Rather it assumes the only consequence of 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, including the PRESS web server.

As shall be seen, our 7-stage model has several advantages compared with traditional stochastic models. First, the 7-stage model is a better fit to the actual system; the server's state transitions follow a linear order, and the timing between phases is deterministic. A probabilistic stochastic process is not a good fit for this situation. This deterministic response to a fault is not really surprising

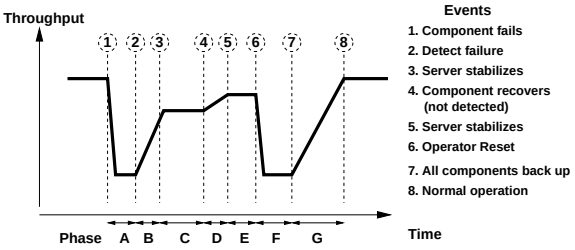


Figure 1: *The 7-stage piece-wise linear model specified by our methodology for evaluating the performability of cluster-based servers.*

given programmer's build deterministic failure and recovery actions in response to faults.

The second advantage of our model over a stochastic approach is that it allows the designer to easily capture the time spent in each state, as opposed to average time spent in the state. For example, modeling a deterministic operator response time is easy in our model, but much more difficult using a stochastic process. While simple methods exist to find the average time spent in a given state in a stochastic model, we argue in the fault-response context a better approach is to reason about the absolute time spent in a given state.

2.1 Phase 1: Measuring Performance Under Single-Fault Fault Loads

There a two tricky issues when injecting faults and recoveries. First, when measuring the server's performance in the presence of a particular fault, the fault must last long enough to allow all stages in the model of phase 2 to be observed and measured. 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 abstract model to 0). Second, 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 from the injection time of a fault.

2.2 Phase 2: Modeling Server's Performability Under Expected Fault Loads

Our model for describing average performance an availability is built in two parts. The first part of the model describes the system's response to a single fault in a 7-stage model. The second part of the model combines the effects of each fault, as described in the first part, along with the MTTR and MTTF of each component to arrive at an overall average availability and performance.

Per-Fault Seven-Stage Model Figure 1 illustrates our 7-stage model of service performance in the presence of a fault. Time is shown on the X-axis and throughput is shown on the Y-axis. Stage A models the degraded throughput delivered by the system from the occurrence of the fault to when the system detects the fault. Stage B models the transient throughput delivered as the system reconfigures to account for the fault; the system may take some amount of time to reach a stable performance regime because of warming effects. 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 failed 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. Stage E models the stable performance regime achieved by the service after the component has recovered. Note that in the figure, we show the performance in E as being below that of normal operation; this may occur because the system was unable to reintegrate the recovered component or reintegration does not lead to full recovery. In this case, throughput remains at the degraded level until an operator detects the problem. 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 measured in phase 1. The first is either measured, or is a parameter that must be supplied. For example, the time that a service will remain in stage B assuming that the fault last sufficiently long is typically measured; the time a service will remain in stage 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. In practice, we set the length of time the system is in the stage to server's response to zero. If the assumed MTTR of a component is less than the measured time for stages A and B, then we assume that B is cut short when the component recovers. The evaluator must analyze the measurements gathered in phase 1, the assumed parameters of the fault load, and the environment carefully to correctly parameterize the model.

Modeling 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 and all fault arrivals are exponentially distributed, so that we can add together the various fractions of time spent in degraded modes. Then, 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 1 during c's failure, 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 $W_c = (\sum_{s=A}^{G} D_c^s)/MTTF_c$.

It is interesting to consider why the denominator of W_c is just $MTTF_c$ instead of $MTTF_c + MTTR_c$. The equation for W_c is correct as it is because we assume fault arrivals for each component are exponentially distributed, and thus memoryless. Our assumption means that when a component fails, another fault could arrive and "queue" at the component. This assumption makes our model tractable but it does differ from a fault model where a faulty component cannot become faulty again before it has been repaired. The impact on our model is that we compute the fraction of downtime as $\frac{MTTR}{MTTF}$, not as the more typical $\frac{MTTR}{MTTF+MTTR}$. In practice, because $MTTF >> MTTR$ the numerical impact of this difference in assumptions is minimal. However, a full investigation of discrepancies between these assumptions is beyond the scope of this work.

2.3 Performability Metric

Despite much work that study both performance and availability (e.g., [19, 26]), there's arguably no *single* performability metric for comparing systems. Thus, if we have a number of systems (or versions of the same system) that give different points of performance vs. availability in this two-dimensional space, it is difficult to say which is better. We propose a combined *performability* metric that allows direct comparison of systems using both performance an availably as input criteria. Our approach is to multiply the average throughput by an availability factor; the challenge, of course, is to derive a factor that properly balances both availability and performance. Because availability is often posed in term of "the number of nines" achieved, we believe that a log-scaled ratio of how each server compares

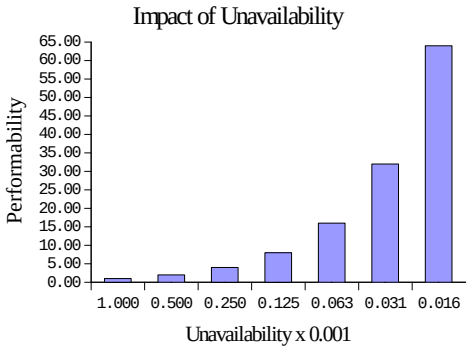


Figure 2: *Impact of unavailability on the performability metric, assuming that throughput stays constant at 100 requests/sec and the ideal availability is 0.99999. The range of unavailability 0.001–0.000016 corresponds to an availability range of 0.999–0.999984.*

to an ideal system would make an appropriate weighing factor for availability, giving the following equation for performability:

$$P = T_n \times \frac{\log(A_I)}{\log(AA)}$$

where A_I is an ideal availability, T_n is the throughput under normal operation, AA is the average availability, and P is the performability of the system.

This metric is an intuitive measure for performability because it scales linearly to both performance and unavailability. Obviously, if performance doubles, our performability metric doubles. On the other hand, if the *unavailability* decreases by a factor of 2, then performability also roughly doubles. This impact of unavailability on our performability metric is illustrated in Figure 2. The intuition behind this relationship between unavailability and performability is that we can approximate $\log(1-u)$ as $-u$ when u is small.[1] Thus,

$$\frac{\log(A_I)}{\log(1-\frac{1}{2}u)} \approx \frac{\log(A_I)}{-\frac{1}{2}u} = 2\frac{\log(A_I)}{-u} \approx 2\frac{\log(A_I)}{\log(1-u)}$$

3 Mendosus

Mendosus is a cluster-based fault injection and network emulation infrastructure that we are building to support phase 1 of our methodology. Mendosus addresses two specific problems that service designers are faced with

[1] Recall that the Taylor expansion of $\log(1-u)$ is $(-u) - \frac{(-u)^2}{2} + \frac{(-u)^3}{3} - \frac{(-u)^4}{4}$... Thus, when u is very small, we can safely approximate $\log(1-u)$ as $-u$.

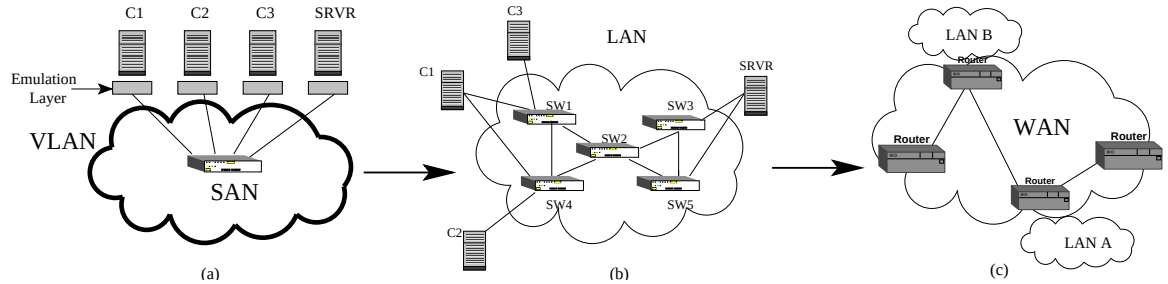


Figure 3: *The goal of the Mendosus project is to use a cluster interconnected by a high-performance LAN or SAN, shown in (a), to emulate a variety of interconnected systems and provide a live fault-injection platform for performability studies. (c) shows our ultimate goal of emulating complex systems, possibly comprised of multiple different LANs, including wireless, interconnected by a WAN. (b) shows the current status of Mendosus, which supports the emulation of clusters interconnected by Ethernet or VIA LANs.*

today: (1) how to put together a sufficiently representative test-bed to test a service as it is being built, and (2) how to conveniently introduce faults to study the service's behavior under various fault loads.

Figure 3 shows our vision of using a cluster interconnected by a high-performance LAN or SAN to emulate systems interconnected by a variety of *virtual networks*. Applications to be tested/evaluated run directly on the nodes in the Mendosus cluster, communicating through the virtual network. Our eventual goal is to support the emulation of a wide range of interconnected systems, including wireless, LAN, and WAN-connected systems.

Hand-in-hand with the capability of emulating a variety of interconnected systems is our goal of providing a comprehensive online fault-injection infrastructure. This includes network delays and outages as well as failure of components in devices and servers. For example, links can fail, switches can drop packets, routers can get congested, node components (e.g., disk and NIC) can fail, etc. This infrastructure will allow designers to test services against a variety of faults, separately and in combination, to better evaluate or validate their performability.

Mendosus is under active development. Currently, Mendosus supports the emulation of a cluster of PCs connected by either a virtual Ethernet LAN, with arbitrary configurations of switches, hubs, and NICs, or by a (limited) VIA SAN. (Development of WAN emulation is currently under way.) In the VIA configuration, Mendosus does not really provide any emulation capability; Mendosus simply supports limited fault injection in the network subsystem.

In this section, we first briefly describe Mendosus's architecture and then discuss the fault models used by the network, disk, and node fault injection modules, which are used extensively in this work, in more detail.

3.1 Architecture

Mendosus is comprised of four software components running on a cluster of PCs physically interconnected by a Giganet VIA network: (1) a global controller, (2) a per-node LAN emulator module, (3) a set of per-node fault-injection kernel modules, and (4) and a per-node user-level daemon that serves as the communication conduit between the global controller and the kernel modules.

The global controller is responsible for deciding when and where faults should be injected and for maintaining a consistent view of the entire network. When emulation starts, the controller parses a configuration file that describes the network to be emulated and components' fault profiles. It forwards the network configuration to the daemon running at each node of the cluster. Then, as the emulation progresses, it uses the fault profiles to decide what faults to inject and when they should be injected. It communicates with the per-node daemons as necessary to effect the faults (and subsequent recovery). Note that while there is one global controller per emulated system, it does not limit the scalability of Mendosus: the controller only deals with faults and does not participate in any per message operations.

The per-node emulation module maintains the topology and status of the virtual network to route messages (this information is passed from the controller to the emulation module through the per-node daemon). To emulate routing in Ethernet networks, a spanning tree is computed for the virtual network. Each emulated NIC is presented as an Ethernet device; a node may have multiple emulated NICs. When a packet is handed to the Ethernet driver from the IP layer, the driver invokes the emulation module to determine whether the packet should be forwarded over the real network (and which node it should be forwarded to). The emulation module determines the emulated route that would be taken

Faulty Component	Fault	Example Error Source	Parameters
NIC	NIC temporarily down	driver hangs	down time
Link	Connection temporarily down	accidental unplugging of cable	down time
Link	Link drops packet	inadequately shielded cables	loss rate
Switch/Hub	Switch temporarily down	power failure	down time
Switch/Hub	One port temporarily down	duplex mismatch	down time
Switch/Hub	Switch drops packet	queue overflow	drop rate
Switch	Switch losses packet	routing error	loss rate

Table 1: *Types of network faults that Mendosus can currently inject. Each network component can have an associated fault profile, specifying the kind of faults that can occur and distributions and/or time traces of fault arrivals.*

by the packet. It then queries the network fault-injection module whether the packet should be forwarded. If the answer is yes, the packet is forwarded to the destination over the underlying real network. The emulation module uses multiple point-to-point messages to emulate Ethernet multicast and broadcast. A leaky bucket is used to emulate Ethernet LANs with different speeds.

Finally, the set of fault-injection kernel modules effect the actual faults as directed by the central controller. Currently, we have implemented 3 modules, allowing errors to be injected into the network, SCSI disk subsystems, and individual nodes. The challenge in implementing these subsystems is to accurately understand the set of possible real faults and the error reporting that percolate from the device through the device drivers, operating systems, and ultimately, up to the application. We describe the fault models we have implemented in more details in the next several sections.

3.2 Network Fault Model

The network fault model includes faults possible for network interface cards, links, hubs, and switches. Table 1 lists all faults that Mendosus can currently inject into the networking subsystem. As specified, all faults are transient although a permanent fault can be injected by specifying a down time that is greater than the time required to run the fault-injection experiment.

Our fault-injection module is embedded within an emulated Ethernet driver. Recall that the emulated driver also includes our LAN emulator module, which contains all information needed to compute the route that each packet will take. Fault injection for the network subsystem is straightforward when the communication protocol already implements end-to-end error detection. Faults are effected simply by checking whether all components on the route are up. If any are down, the packet is simply dropped. If any components are in a intermittent error state, then the packet is dropped (or sent) according to the specified distribution.

Recall that the central controller is responsible for instructing the network fault injection modules on when, where, and what to inject dynamically. Instructions from the central controller are received by the local daemon and passed to the injection module through the ioctl interface. The fault-injection and emulation modules must work together in that faults may require the emulation module to recompute the routing spanning tree. The central controller is responsible for determining when a set of faults leads to network partition. When this occurs, the controller must choose a root for each partition so that the nodes within the partition can recompute the routing spanning tree.

3.3 SCSI Disk Fault model

The SCSI subsystem is comprised of the hard disk device, the host adaptor, a SCSI cable connecting the two, and a hierarchy of software drivers. Higher layers in the system try to mask failures at lower levels and only the fatal failures are explicitly passed up the hierarchy. We model failures that are noticeable by the application either by explicitly forcing error codes reported by the OS, or implicity by extended delays in the completion of disk operations.

We broadly classify possible faults into two categories: *transient* and *sticky (non-transient)*. For transient faults, the disk system recovers after a small finite interval (order of a few seconds in most cases); sticky faults require human intervention for correction. Examples of transient faults are SCSI timeouts, recoverable read and write failures, etc., while disk hang, external SCSI cable unplugging and power failures (to external SCSI housing) classify as sticky failures.

The impact of these faults on the system depends on their criticality. Failures that can be masked by intermediate software or hardware components, we term as *recoverable* failures, while those that cannot be, are termed *unrecoverable*. A parity error in the SCSI bus is recoverable by a retry, while a disk hang due to a firmware bug is unrecoverable without external intervention. While recoverable faults (those which need re-

Fault	Characteristic	Criticality	Example Error Source
Disk Hang	Non-transient	Unrecoverable	Disk firmware bug
Disk Offline	Non-transient	Unrecoverable	Maintenance, fatal error
Power Failure	Non-transient	Unrecoverable	External SCSI power failure
Read failure	Non-transient	Unrecoverable in Linux	Bad Sector (un-remappable)
Write failure	Non-transient	Unrecoverable in Linux	Bad sector
Timeout	Non-transient	Unrecoverable	SCSI Cable Failure
Parity Errors	Transient	Recoverable	SCSI Cable glitch
Bus Busy	Transient	Recoverable	Commands on bus
Queue Full	Transient	Recoverable	Host queue full

Table 2: *Types of SCSI errors that Mendosus can currently inject.*

tries or corrective action) may introduce tolerable delays (e.g., SCSI bus parity errors are handled by retrying the operation), unrecoverable faults may lead to stalling of execution. However some unrecoverable faults, if recognized, can be handled by using alternate resources. This involves implementing smarter, fault-aware systems.

Table 2 shows the faults we can inject into the SCSI subsystem. The fault injection module is interposed between the adapter-specific low-level driver and the generic mid-level driver. Instructions for injecting faults received from the central controller by the local daemon are communicated to the fault injection module through the proc filesystem. To effect faults, the fault injection module traps the queueing of disk operation requests to the low-level driver and prevents or delays the operation that should be faulty from reaching the low-level driver. In the former case, the module must return an appropriate error message.

The mid-level driver implements an error handler which diagnoses and corrects rectifiable faults reported by the low-level driver, either by retrying the command or by resetting the host, bus, device or a combination of these. The unrecoverable read and write failures, caused by bad sectors unremappable by the disk controller are not handled by the upper drivers or the file system in Linux. This causes read and write operations to the bad sector to fail forever. The disk can be taken offline by the new error handler code introduced in 2.2+ Linux kernels when all efforts to rectify an encountered error fail. The disk can also be offline if it has been taken out for maintainence or replacement.

3.4 Node and Process Fault Model

Currently, our node and process fault model is simple. Mendosus can inject three types of node faults: hard reboot, soft reboot, and node freeze. All can be either transient or permanent, depending on the specified fault load. In the application process fault model, Mendosus can inject an application hang or crash. We may consider more subtle node/process faults such as corrupting memory or caches in the future if warranted.

This fault model is implemented inside the user-level daemon at each node. For our study of PRESS, the server process on each node is started by the daemon. An application hang fault is injected by having the daemon send a SIGSTOP to the server process. The process can be restarted if the fault is transient by sending a SIGCTN to it. A process crash is injected by killing the process.

Node faults are introduced using an APC power management power strip. Reboot faults are introduced by having the daemon on the failing node contact the APC power strip to power cycle that node. In the case of a soft reboot, the daemon can ask the APC for a delayed power cycle and then run a shutdown script. For a node freeze, the daemon directs a small kernel module to spin endlessly to take over the CPU for some amount of time.

4 The PRESS Server

PRESS is a highly optimized yet portable cluster-based locality-conscious WWW server that has been shown to provide good performance in a wide range of scenarios [8, 9]. Like other locality-conscious servers [22, 2, 7, 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. Essentially, the server distributes HTTP requests across the cluster nodes based on cache locality and load balancing considerations, so that the requested content is unlikely to be read from disk if there is a cached copy somewhere in the cluster.

PRESS assumes that HTTP requests are directed to the cluster using a standard method, such as Round-Robin DNS or a content-oblivious, load balancing front-end device. Thus, 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

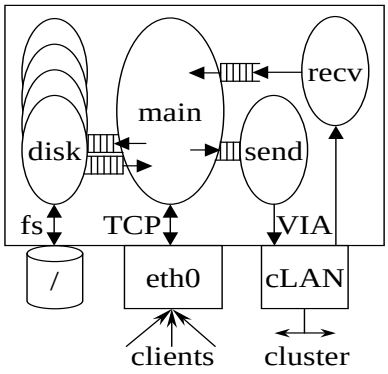


Figure 4: *Basic PRESS architecture.*

the request to another node, the *service node*.

A forwarded request is handled in a straightforward way by the service node. If the requested file is cached, the service node simply transfers it to the initial node. If the file is not cached, the service node reads the file from its local disk, caches it locally, and finally transfers it to the initial node. Upon receiving the file from the service node, the initial node sends it to the client. The initial node does not cache the file received from the service node to avoid excessive file replication across the cluster.

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 4 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.

When PRESS first starts up, each cluster node sets up a communication channel (either a TCP connection or a VIA channel) with each other node in the cluster. The `send/receive` threads use these channels for intra-cluster communication. These threads remain blocked until there is a message to be sent or received. To send a message, `main` queues a *digest* of the message on a data structure shared between `main` and the `send` thread, and unblocks `send`. When the `send` thread wakes up, it uses the appropriate channel to send the message. The `receive` thread is unblocked by the arrival of a message on any of the communication channels. When the `receive` thread wakes up, it determines which channel the message arrived on, reads it, and places a digest of the message at a data structure shared with the `main` thread. The `main` thread periodically polls this data structure.

Reconfiguration. PRESS is often used (as in our experiments) without a front-end device, exposing the IP address of the cluster nodes to clients. To prevent clients from experiencing failed requests (due to a node failure, for instance), some versions of PRESS implement node failure detection, temporary recovery through IP address take-over, and final recovery with the re-integration of recovered nodes. The detection mechanism when TCP is used for intra-cluster communication employs periodic heartbeat messages. To avoid sending too many messages, we organize the cluster nodes in a directed ring structure. A node only sends heartbeats to the node it points to. If a node does not receive three consecutive heartbeats from its predecessor in the ring, it assumes that the predecessor has failed.

Fault detection when VIA is used for intra-cluster communication is simpler. PRESS does not have to send heartbeat messages itself. (It can rely on our VIA implementation to maintain node and communication up/down information. Any detected faults cause the corresponding connections to be terminated/broken.) A node assumes that another node has failed if the VIA connection between them is broken. In this implementation, nodes are also organized in a directed ring, but only for recovery purposes.

Temporary recovery is implemented by simply excluding the failed node from the server or by having the successor node take-over the IP address of the failed node. In the latter case, the successor node sends a gratuitous ARP message to the effect that the new physical address for the failed IP address is its own. *In our experiments, we eliminated the IP take-over part of the recovery process.* Multiple node failures can occur simultaneously. Every time a failure occurs, the ring structure is modified to reflect the new cluster configuration.

The second and final step in recovery is to re-integrate a recovered node into the cluster. If IP take-over is in effect, this involves returning the old IP address to the recovered node. Since we do not use IP take-over in our experiments, we do not describe this scenario further. When IP take-over is not in effect and the intra-cluster communication protocol is TCP, the rejoining node broadcasts 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

Version	Main Features	Expected Behavior	Throughput
I-PRESS	Independent servers	Lowest performance but most robust to failures.	1250 reqs/sec
TCP-PRESS	Cooperative caching servers using TCP for intra-cluster communication	Performs better than independent versions due to higher cache hit rates. Introduces dependencies between nodes, so a failure may affect multiple nodes. If fault leads to permanent break in communication or loss of one or more PRESS processes, then must be restarted manually by operator for full recovery	4965 reqs/sec
ReTCP-PRESS	Cooperative caching servers using TCP for intra-cluster communication and dynamic reconfiguration	Performs as well as TCP-PRESS but should be able to recover automatically from more faults.	4965 reqs/sec
VIA-PRESS	Cooperative caching servers using VIA for intra-cluster communication and dynamic reconfiguration	Performs the best and, like ReTCP-PRESS, should be able to automatically recover from more faults than TCP-PRESS.	6031 reqs/sec

Table 3: *Versions of PRESS available for study, their differences, expected impact on performance and availability, and peak throughput.*

with the other nodes. After each connection is reestablished, the rejoining node is sent the caching information of the respective node. When the intra-cluster communication is done with VIA, the rejoining node simply tries to reestablish its connection with all other nodes. As connections are reestablished, the rejoining node is sent the caching information of the respective nodes.

Versions. Several versions of PRESS have been developed in order to study the performance impact of different communication mechanisms [9]. Table 3 lists the versions of PRESS that we consider in this paper. For each version, we summarize its main characteristics, their expected impact on performance and availability, and their near-peak throughputs on our 4 cluster nodes. The base version of press, I-PRESS, is comprised of a number of independent WWW servers (based on the same code as PRESS) answering client requests. This is equivalent to simply running multiple copies of Apache, for example. The other versions cooperate in caching files and differ in terms of their concern with availability, and the performance of their intra-cluster communication protocols. The throughputs of the various versions of PRESS will be compared against throughput when various faults are injected into the cluster.

5 Evaluating PRESS Under Single-Fault Fault Loads

We now apply the first phase of our methodology to evaluate PRESS's performability. We study all four versions of PRESS to advance the state-of-the-art understanding of how to design highly available servers in addition to showing how to apply our methodology. In this section, we first describe our experimental test-bed then show a sampling of PRESS's behavior under our fault loads. Throughout this section, we do not show results for I-PRESS as they entirely match expectation: the achieved throughput simply depends on how many of the nodes are up and able to serve client requests.

5.1 Experimental Setup

In all experiments, we run a four-node version of PRESS on four 800 MHz PIII PCs, each equipped with 206 MB of memory and 2 10,000 RPM 9 GB SCSI disks. Nodes are interconnected by a 1 Gb/s cLAN network. We can communicate with TCP or VIA over this network. PRESS was allocated 128 MB on each node for its file cache; the remainder of the memory was sufficient for the OS such that little paging took place during our experiments. PRESS only serves static content and the entire set of documents is replicated at each node on one of the disks. In all experiments, PRESS was loaded at 90% of saturation and set to warm up to this peak throughput over a period of 5 minutes.

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. We were forced to this single-network configuration because we cannot saturate some versions of PRESS if we use our 100 Mb/s Ethernet as the external network; the Ethernet NICs become the bottleneck, preventing the clients from generating sufficient load. Fortunately, 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 in our setup. Finally, Mendosus's network emulation system allows us to differentiate between intra-cluster commu-

Subsystem	Fault	Characteristics
Network	Link down	Transient - 5, 180 secs
	Switch down	Transient - 5, 180 secs
Disk	SCSI timeout	Transient - 120 secs
	Disk Hang	Sticky
	Read failures	Sticky
	Write failures	Sticky
Node	Hard reboot	Transient - 180 secs
	Node freeze	Transient - 5, 180 secs
Application	Process crash	Transient - 5, 180 secs
	Process Hang	Transient - 5, 180 secs

Table 4: *Fault loads for PRESS performability study.*

nication and client-server communication when injecting network related faults. Thus, the clients are never disturbed by faults injected into the intra-cluster communication.

Each client generates load by following 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 [8]. We made one artificial change to avoid large variation in throughput due to requests for very long files: we use a synthetic file set with constant-sized files, where the size is equal to the average file size of the actual file set. This modification was necessary to prevent the injection time of each fault from becoming an issue; PRESS was able to achieve relatively stable throughput after an initial warming phase under our synthetic load so that a fault can be injected at any time.

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. These timeouts were actually smaller than what we would have liked—small timeouts sometime lead to large throughput variance at the server—but larger values would lead to clients exceeding the maximum number of open connections allowed by the underlying Linux kernel when requests do not complete at the normal rate because of a server fault.

Finally, Table 4 lists the set of faults that we will inject into a live PRESS system to study its behavior. Faults fall into four categories: network, disk, node, and application. Note that these generic faults can be caused by a wide variety of reasons for a real system; for example, an operator accidentally pulling out the wrong network cable would lead to a link failure. Likewise, many application bugs can lead to the crash of an application process. We cannot focus on all potential causes because this set is too large. Rather, we focus on the class of failures as observed by the system, using an MTTF that covers all potential causes of a particular fault. This set is comprehensive with respect to PRESS in that it covers just about all resources that PRESS uses in providing its service.

5.2 Network Failures

In this section, we study PRESS's behavior under network faults. Figure 5 shows the effects of a transient switch failure. We first discuss what happened in each case, then make an interesting general observation.

TCP-PRESS behaved exactly as expected: throughput drops to zero a short time after the occurrence of the failure because the queues for intra-server communication fill up as the nodes attempt to fetch content across the faulty switch. Throughput stays at zero until the switch comes back up. For ReTCP-PRESS, the first switch failure leads to the same behavior as TCP-PRESS; the reconfiguration code does not activate because the failure is sufficiently short that no heartbeat is lost. The longer switch failure, however, triggers the reconfiguration code, leading ReTCP-PRESS to reconfigure into 4 groups of singleton. The detection time is determined by the heartbeat protocol, which uses a DEADTIME interval of 15 seconds (3 heartbeats). For VIA-PRESS, the switch failure is detected almost immediately by the device driver, which breaks all VIA connections to unreachable nodes. This immediately triggers the reconfiguration of VIA-PRESS into four subclusters.

Interestingly, ReTCP-PRESS and VIA-PRESS do not reconfigure back into a single cluster once the switch returns to normal operation. This surprising behavior arises from a mismatch between the fault model assumed by the reconfigurable versions of PRESS and the actual fault. These versions of PRESS assume that nodes fail but links and switches do not fail. Thus, reconfiguration only occurs at startup and on loss of 3 heartbeats. If a cluster is splintered as above, they never attempt to rejoin. Return to full operation thus would require the intervention of an administrator to restart all but one of the sub-clusters. This, in effect, make these reconfigurable versions *less* robust than the basic TCP-PRESS in the face of relatively short transient faults, and points to the importance of the accuracy of the fault model used in designing a service.

Finally, we do not show results for the link/NIC failure here because they essentially led to the same behaviors as above. Instead of splintering into four subclusters, ReTCP-PRESS and VIA-PRESS splinter into two sub-clusters, one with 3 cooperating nodes and a singleton. ReTCP-PRESS does not splinter until the

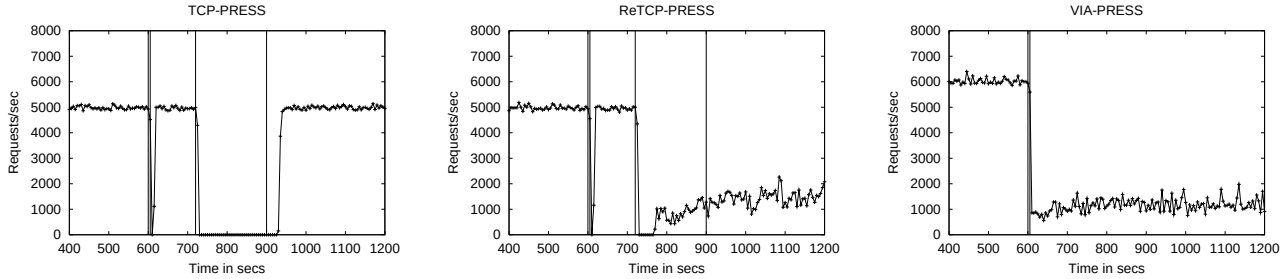


Figure 5: *Effects of transient switch failures. Pairs of vertical lines represent the start and end times of injected faults.*

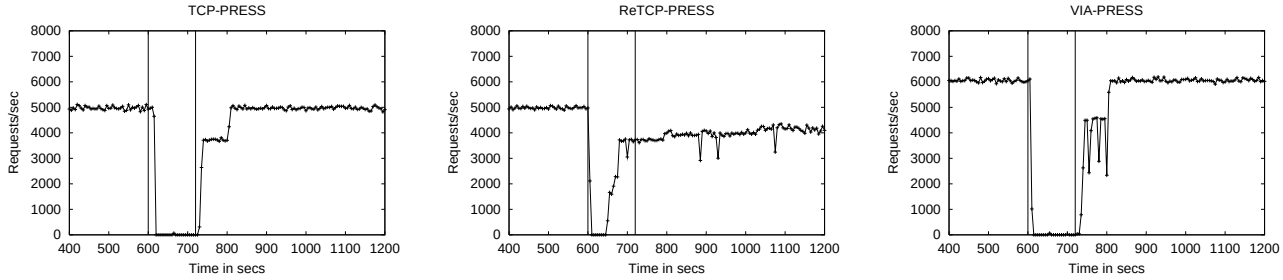


Figure 6: *Effects of transient SCSI time-out faults. Pairs of vertical lines represent the start and end times of injected faults.*

failure exceeds the fault detection interval. VIA-PRESS splinters almost immediately.

5.3 Disk Failures

Recall that each server machine contains two SCSI disks, one holding the operating system and the second the file set being served by PRESS. We inject faults into only the second disk to minimize the interference from OS related disk accesses (e.g., page swapping) while observing the behavior of PRESS under disk failures.

Figure 6 shows PRESS's behavior under SCSI time-outs. TCP-PRESS and VIA-PRESS behave exactly as one would expect. When the fault lasts long enough, all `disk` helper threads become blocked and the queue between the `disk` threads and the `main` thread fills up. When this happens, the `main` thread itself becomes blocked when it tries to initiate another read. Once one of the nodes grinds to a halt, then the entire server eventually comes to a complete halt as well. When the faulty disk recovers, the entire system regains its normal operation.

ReTCP-PRESS, on the other hand, interprets the long fault as a node failure and so splinters into sub-clusters, one with 3 nodes and one singleton. This splintering of the server cluster is caused by missing heartbeats. Similar to the argument for TCP-PRESS and VIA-PRESS above, when all `disk` threads block because of the faulty disk, the `main` thread also eventually blocks when it tries to initiate yet one more read. In this case, however, the main thread is also responsible for sending the heartbeat messages. Thus, when it blocks, its peers do not get any more heartbeats and so assume that that node is down; at this point, the reconfiguration code takes over, leading to the splinter.

We do not show the results for disk hang and read and write errors because PRESS's behaviors are much as expected.

5.4 Node Faults

Figure 7 shows the effects of a hard reboot fault. Because it is not capable of detecting node failures, TCP-PRESS grinds to a halt while the faulty node is down. When the node successfully reboots, however, the open TCP connections of the three non-faulty nodes with the recovered node break. (Interestingly, TCP time-outs could have led to connection termination as well. However, these TCP timeouts tend to be very long, on order of 10-15 minutes. Thus, the rebooting node recovers long before these TCP timeouts trigger.) At this point, the PRESS processes running on these nodes real-

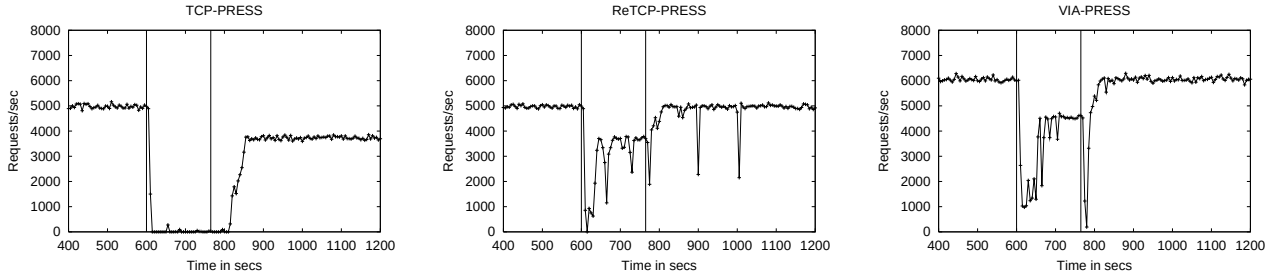


Figure 7: *Effects of a node crash. Pairs of vertical lines represent the start and end times of injected faults.*

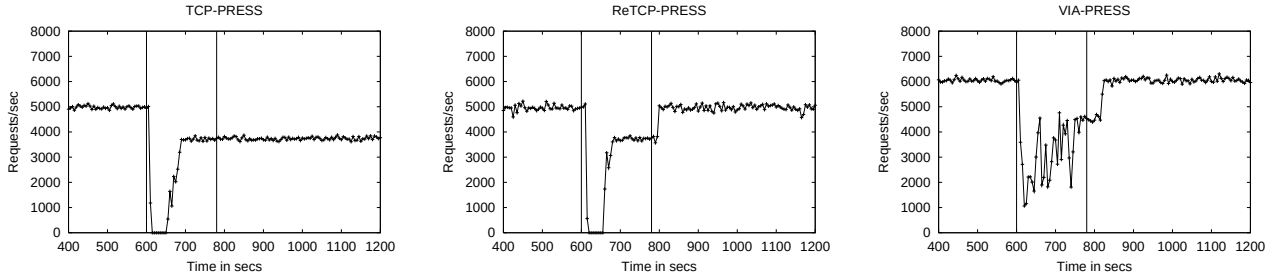


Figure 8: *Effects of one PRESS process crashing. Pairs of vertical lines represent the start and end times of injected faults.*

ize that something has happened to the faulty node and stop attempting to coordinate with it. Thus, server operation restarts with a cluster of 3 nodes. When the faulty node successfully reboots, Mendosus starts another PRESS process automatically. However, since TCP-PRESS cannot reconfigure, correct operation with a cluster of 4 nodes cannot take place until the entire server is shutdown and restarted.

ReTCP-PRESS and VIA-PRESS behave exactly as expected. Operation of the server grinds to a halt until the reconfiguration code detects a failure. The three non-faulty nodes recover and operate as a cooperating cluster. When the faulty node recovers and the PRESS process has been restarted, it joins in correctly with the three non-faulty processes and throughput eventually returns to normal.

5.5 Application Faults

Figure 8 shows the effects of an application process crash. For this fault, all versions behave as expected. ReTCP-PRESS and VIA-PRESS exhibit severely degraded throughput until the crashed process is restarted and re-integrated to the server. TCP-PRESS also suffers with the crash, but cannot return to full throughput because the restarted process never rejoins the server.

6 Modeling PRESS

We now proceed to the second phase of our methodology: using the analytic model to extrapolate performability from our experimental fault-injection results. We first compare the performance, availability, and performability of the different versions of PRESS. Then, we show how we can use the model to evaluate design tradeoffs, such as adding a RAID or increased operator support.

6.1 Parameterizing the Model

We parameterize our model by using the data collected in phase one, the fault load shown in Table 6, and a number of assumptions about the environment. While we cannot list all data extracted from phase 1 here because of space constraints; we refer the interested reader to http://www.panic-lab.rutgers.edu/Research/mendosus/. Table 5 provides a flavor of this data, listing the throughput and duration of each phase of our 7-phase model for VIA-PRESS for two types of faults. The MTTFs and MTTRs shown in Table 6 were chosen based on previously reported faults and fault rates [14, 16, 28]. Note that we do not model all the faults that we can inject because there are no reli-

Phase	Switch Failure		Application Crash	
	Throughput (reqs/sec)	Duration (secs)	Throughput (reqs/sec)	Duration (secs)
A	892.40	75	1889.10	10
B	–	0	3143.55	145
C	1106.70	3525	4537.60	25
D	–	0	4789.13	45
E	1209.60	300	–	0
F	0.0	300	–	0
G	3017.00	300	–	0

Table 5: *Example throughput and duration of the phases in our model for VIA-PRESS for two different faults: switch failure and application process crash. Note that for some types of faults, some phases collapse into a single phase or are not used.*

Fault	MTTF	MTTR
Link down	6 months	3 minute
Switch down	1 year	1 hour
SCSI timeout	1 year	1 hour
Hard reboot	2 weeks	3 minutes
Process crash	1 month	3 minute

Table 6: *Failures and their MTTFs and MTTRs.*

able statistics for some of them, e.g., application hangs. Finally, our environmental assumptions are that operator response time for stage E is 5 minutes and cluster reset time for stage F is 5 minutes. Recall from Section 5.1 that G, the warm up period, was also set to 5 minutes.

6.2 Modeling Results

Figure 9(a) shows the expected average throughput in the face of component faults for the 4 PRESS versions. As has been noted in previous work, the locality-conscious request distribution significantly improves performance. The use of user-level communication improves performance further.

Figure 9(b) shows the average unavailability of the different versions of PRESS. Each bar includes the contributions of the different fault types to unavailability. These results show that availability is somewhat disappointing, on the order of 99.9%, or "three nines". However, recall that the servers were operating near peak; any loss in performance, such as losing a node or splintering, results in an immediate loss in throughput (and in many failed requests). A fielded system would reserve excess capacity for handling faults. Exploring this tradeoff between performability and capacity is a topic for our future research.

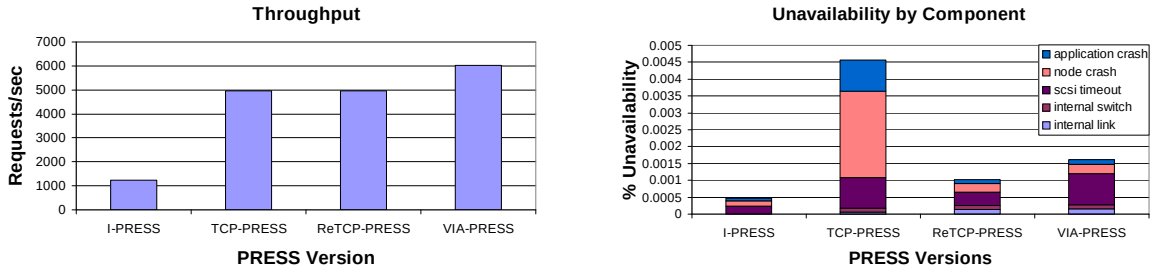
Comparing the systems, observe that I-PRESS achieves the best availability because there's no coordination between the nodes. TCP-PRESS is almost an order of magnitude worse than I-PRESS; this is perhaps expected since TCP-PRESS does a very poor job of tolerating and recovery from faults. More interestingly, ReTCP-PRESS gives better availability than VIA-PRESS. Looking at the bars closely, we observe that this is because ReTCP-PRESS is better at tolerating SCSI timeouts. This is fortuitous rather than by design: as previously discussed, when a SCSI timeout occurs, the heartbeats are delayed in ReTCP-PRESS, causing the cluster to reconfigure and proceed without the faulty node. VIA-PRESS does not reconfigure because the communication subsystem does not detect any error.

Finally, Figure 10 shows the performability of the different PRESS versions. We can see that although the locality-aware, cooperative nature of TCP-PRESS does deliver increased performance, the lack of much, if any, fault-tolerance in its design reduces availability significantly, giving it a lower performability score than I-PRESS. The superior performance of the ReTCP and VIA versions of PRESS make up for their lower availability over I-PRESS. Again, the fact that ReTCP gives a better performability score than VIA-PRESS is the fortuitous loss of heartbeats on SCSI timeouts. Thus, we do not make any conclusion based on this difference (rather, we discuss the nature of heartbeats in Section 7).

Quantifying Design Tradeoffs Analytic modeling of faults and their phases allows to explore the impact of our server designs on performability. Thus, we examine two alternative design decisions to the ones we have explored so far.

The first design change is to reduce the operator coverage. In the previous model, the mean time for an operator to respond when the server entered a non-recoverable state was 5 minutes. This represents the PRESS servers running under the watchful eyes of operators 24x7. However, as this is quite an expensive proposition, we reduced the mean response to 4 hours and observed the performability impact.

Figure 9(b) suggests that disks are a major cause of



(a) **(b)**

Figure 9: *(a) Average modeled throughput and (b) modeled unavailability (1 - availability).*

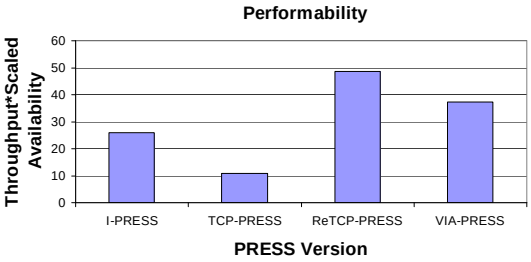


Figure 10: *Performability of each version of PRESS.*

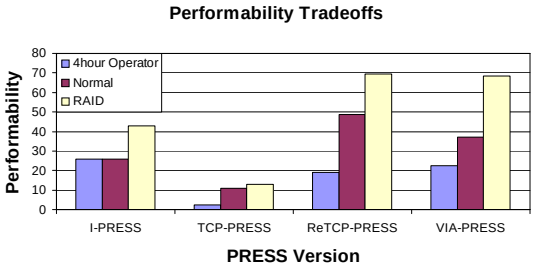


Figure 11: *Impact of reducing the mean operator response from 5 minutes to 4 hours and adding a more reliable disk subsystem.*

unavailability. In this second design change, we added a much more reliable disk subsystem, e.g. a RAID. We modeled the better disk subsystem by increasing the mean time to failure of the disks by a factor of five, but keeping the MTTR of the disks the same.

Figure 11 shows the performability impact of these two design changes. The center bar represent the same "basic" results as in Figure 10. The left most bar is the basic system with a 4-hour operator response, and the right is the basic system enhanced with a RAID.

Our modeling results show that running the cooperative versions in an environment with quick operator response is critical (unless fault recovery can be improved significantly): the performability of all the cooperative versions become less than that of I-PRESS. On the other hand, our results show the I-PRESS is insensitive to operator response time as expected.

The performability models also demonstrate the utility of a highly-reliable disk subsystem. Figure 11 show that by purchasing increasingly reliable disk subsystems, performability of all versions of PRESS is enhanced, e.g., approximately 84% for VIA-PRESS. While not changing the relative performability, our models suggest that the overall *system impact* of redundant disk organizations, such as RAIDs, is substantial.

Scaling to Larger Cluster Configurations The modeling results we have presented thus far are based on our fault-injection experiments with a 4-node cluster configuration. Although these results provide insights into the relative performability of the different versions of PRESS and into key service design tradeoffs, they are representative of small-scale cluster configurations. It is now important to understand the effect of scaling up the cluster size.

Essentially, our model depends on three types of parameters: the mean time to failure of each component ($MTTF_c$), the duration of each phase during the failure of a component (D_c^p), and the (average) throughput under normal operation (T_n) and during each failure 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 $MTTF_c^N/S$. Assuming that the bottleneck resource is the same for the N-node and the SN-node configurations, i.e. the interconnect resources are scaled with the machine size, the durations D_c^p should be the same under both configurations. Under the same assumption, the normal throughput under N nodes, T_n^N, should be S times smaller than

under SN nodes, i.e. $T_n^{SN} = S \times T_n^N$. Unfortunately, the effect of scaling on the throughput of each failure phase is not as straightforward. When the effect of the failure is to bring down a node or make it inaccessible, for instance, the throughput of phases C and D should approach $T_n^{SN} - T_n^{SN}/SN$ under SN nodes, whereas the average throughput of phases B and F 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 E should approach 0 under SN nodes.

Putting all of these effects together, we believe will lead to larger performability values for the different versions of PRESS (due to the significantly larger throughputs involved) for $S > 1$. Nevertheless, *scaling should not affect the trends we observed with 4 nodes*.

We are currently in the process of verifying these observations with real experiments with systems ranging from 2 to 16 nodes. We expect to have complete data on scaling for the final version of this report.

6.3 Validation Approach

We plan to show the predictions of the model are reasonably accurate using two approaches. We have not completed these validations, and indeed a full investigation of the effectiveness of these validation approaches in beyond the scope of this work. In this paper, however, we provide an outline of our two approaches.

The first approach is designed to show that the model accurately describes performability over long time periods. A second benefit of this validation approach is that it provides a measure of the variance in performability, which our model does not capture. The second validation approach bounds the error using a purely analytic method.

Validating Long Time Periods Our model describes average behavior of faults over long periods, however, our observed faults cover only short periods. A key question is thus how accurately our model describes long term behavior. Our validation approach in this case is to first syntheticaly generate the sequence of faults that would occur over a long interval. Using the MTTF of each component, and assuming a known distribution, we create a timeline of failures.

Next, we inject these faults in order of their occurrence using Mendosus, and observe the resulting performability. However, we compress the time between faults, only allowing the system to recover to a normal operating state before injecting the next fault. For example, suppose over a year 2 disk faults and 10 application errors occur, one each month. Using our approach, we would inject 12 faults, in the order they occur, allowing the system to recover each time (or forcing an operator reset, if need be). We would also inject overlapping faults.

Extrapolating our observed performability into the year-long time scale is quite simple; we assume that the server delivers baseline performance during intra-fault periods. We can then compare the extrapolated performability with that predicted by the model. Of course, the performabilty result of this experiment is based on a single sequence of faults, which may not approximate average-case behavior. We expect with this approach to have to run several fault sequences in order to approximate the analytic model.

In addition to experimentally generating a more accurate average performability metric, the use of multiple fault sequences provides us with some measure of the variance between systems. For many systems, a lower variance may be more critical than an absolute performability. For example, it may be more important to trade lower average case performability in return for the best worst-case performance for a given fault load.

Bounding the Independence Error Our model also assumes that the effects of overlapping faults do not interfere. Instead, these effects are queued. For example, if a disk fault arrived during a switch failure, we model the performability of the system is the same as the switch-only scenario. After the switch recovers would we account for the disk failure.

Our approach to bounding the error of overlapping faults is to assume that the performability of the system is zero during the period of the overlap. Because we know the fault arrival process and duration (MTTR), we can compute the probabilities, and thus arrival rates, of each overlap case. However, given that there are $2^c - c$ possible overlap scenarios, we have not yet computed all overlaps. However, given the probability of overlaps are low, we expect them to have little quantitative, and even less qualitative, impact on our findings.

7 Lessons Learned

During our performability evaluation of PRESS, we learned a number of lessons. In this section, we discuss the three most important of these lessons.

The first lesson is that high performance servers can be highly sensitive to even individual faults, as they try to harness whatever resources necessary to achieve the highest performance possible. As we have seen, single faults such as a disk timeout can cause a complete halt in the operation of PRESS. This implies that to achieve high availability as well as high performance, these systems must monitor the status of most (if not all) of its

components. Status information must be disseminated widely throughout the server, so that requests can either be routed around the faults or discarded as early as possible in order to limit the amount of resources wasted on requests that cannot be completed.

The second lesson is that runtime fault detection and diagnosis is a difficult issue to address. Consider the heartbeat system implemented in ReTCP-PRESS, which was fortuitously delayed on a SCSI timeout error because it shares the communication queue with other threads. What should a loss of heartbeat indicate? Should it indicate a node failure? Does it indicate any failure on the node? (If so, then was our implementation guaranteed to be delayed for any other hardware error than disk errors?) How can we differentiate between a node or communication error? Should we differentiate between node and application failures? Again, this implies that systems must carefully monitor the status of all its components, as well as have a well defined reporting system, where each status indicator has a clearly defined semantic.

Finally, the third lesson is that efforts to achieve high availability will only pay off if the assumed fault model matches actual fault scenarios well. Mismatches between PRESS's fault model and actual faults led to some surprising results. A prime example of this is PRESS's assumption that the only possible faults are node or application crashes. This significantly degrades the performability of ReTCP-PRESS and VIA-PRESS because any faults other than a node or application crash that led to splintering of the cluster (e.g., link failure) eventually required the intervention of a human operator before full recovery could occur.

One obvious answer to this last problem is to improve PRESS's fault model; after all, PRESS's fault model is currently very limited. However, the more complex the fault model, the more complex the detection and recovery code, leading to higher chances for bugs. Further, detection would likely require additional monitoring hardware, leading to higher cost as well. One idea that may be worth exploring is to define a limited fault model and then to enforce that fault model during operation of the server. For example, for PRESS, we might enforce the node crash model, meaning that any fault that leads to the separation of a process/node from the main group should lead to the automatic reboot of that node. We believe that it is possible to build such an enforcement infrastructure from already existing equipment, such as the APC power management power strips. While this seems like a drastic approach, keep in mind that the fault model can be richer than that assumed in PRESS. The key point is that a service designer should define carefully the fault model that he/she will be designing his system to, possibly limiting its complexity to something that can be readily dealt with in designing for high availability. Then, a fault enforcement system can be put into place to translate any unexpected faults into one or more expected faults.

8 Related Work

Researchers have studied the problem of fault-tolerance extensively. Discussing this body of literature is beyond the scope of this paper, however. Instead, we concentrate on efforts that have focused more closely on highly available servers. Patterson et al. recently proposed that unexpected faults must be accepted as a fact of life, and that systems should be built to recover rapidly, in addition to being fault-tolerant [1]. This view of faults is similar to our own, although our proposed methodology concentrates more on evaluating performability independent of the approach taken to improve performance or availability. Perhaps more similar to our work is that of Brown et al. [6], which outlines a methodology for benchmarking systems' availability. Our work here focus more closely on cluster-based servers. Also, a significant part of our effort has been invested in the construction of Mendosus as a comprehensive fault-injection infrastructure.

System availability studies such as those by Murphy et.al [20] and Asami [3] are providing much needed data on actual fault profiles.

Works in the past have proposed robustness [25] and reliability benchmarks [29] that quantify the degradation of system performance under failures. Previous work has noted that different cluster organizations have different availability impacts [11]. Our work focus on a methodology that encompasses both.

Mendosus is related to a number of efforts that have explored hardware and software based fault-injection techniques [10, 13, 17, 24]. Mendosus uses an approach similar to Orchestra [10], which uses a software based, script-driven fault probing method. Their focus, however, was on testing distributed protocols against network faults by introducing a layer within the communication stack. We focus instead on studying cluster-based performability and so have concentrated on injecting faults throughout this platform, including networking faults that are consistently visible across all nodes in the cluster. This latter capability depends critically on Mendosus's emulation capabilities.

NFTAPE [27] is similar to Mendosus in that it provides a framework for composing a variety of light weight fault injectors (LWFI) into a generic fault-injection tool. Their component-based approach, with a single controller performing the common tasks of logging, configuration and communication is similar to the

organization of Mendosus. In addition to the framework, however, we have defined specific fault models that can be applied to the study of cluster-based performability.

Finally, Mendosus is related to a number of emulation systems such as Emulab [30], a configurable Internet emulator, and DelayLine [15], a WAN emulator library that provides a socket interface to applications and allow them to be tested under varying network performance conditions. While network emulation is an important component of our infrastructure, ultimately, its only a tool toward our goal of increasing our understanding of cluster-based services' performability in order to improve our design methodology.

9 Conclusions and Future Work

The need for appropriate methodologies and infrastructures for the design and evaluation of highly available server is rapidly emerging as availability becomes an increasingly important metric for network services. In this paper, we have introduced a methodology that uses fault-injection and analytic modeling to quantitatively evaluate the performance *and* availability (performability) of cluster-based services. Designers can use our methodology to study *what if* scenarios, predicting the gain or loss in performability of design changes. We have also introduced Mendosus, a fault-injection and network emulation infrastructure designed to support our methodology.

We evaluated the performability of four different versions of PRESS, a sophisticated cluster-based server, to show how our methodology can be applied. In addition, we also showed how our methodology can be used to assess the potential impact of different design decisions and environmental parameters. An accompanying benefit of studying the various versions of PRESS is that our results provided several insights into server design, particularly concerning runtime fault detection and diagnosis.

We plan several directions for future work. First, we will extend Mendosus to emulate a wider class of networks, in particular wireless LANs and WANs. We also plan to allow for finer-grained control of when faults are injected, for example when a specific code point or item of data is seen. The manual inspection and entry of all the generated models parameters was quite tedious, we thus plan to add monitoring and logging infrastructure as well.

Second, we will extend our modeling work by considering more and different types of faults. In particular, defining a fault model comprised of higher level faults such as "a network partition" and mapping it into the low level faults that Mendosus can inject is one direction. In addition, we will extend the metrics used to include latency and utilization, as well as more novel metrics such as harvest and yield [11]. A longer term modeling goal is to examine the impact of non-exponential fault distributions, which would allow us to model components that degrade over time.

Longer term, we will explore the tradeoffs between performance, availability, and redundancy. Extra capacity can be used for either fault-tolerance or performance. However, the relationship between these three is not yet well-understood. Applying our techniques to a wider range of databases, web servers, and storage systems will help us towards this end.

10 acknowledgments

This work was supported by NSF grant awards 9986046 and 0103722. Enrique Carrera's and Srinath Rao's assistance was invaluable in our understanding the PRESS servers. We also thank Gerard Richter for his critical and timely insight as to the nature of our performability metric. Finally, we wish to thank the anonymous reviewers for their feedback which greatly improved this document.

References

[1] P. D. A., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[2] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of USENIX'2000 Technical Conference*, San Diego, CA, June 2000.

[3] 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.

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

[6] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA*, June 2000.

[7] E. V. Carrera and R. Bianchini. Evaluating Cluster-Based Network Servers. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, pages 63–70, Pittsburgh, PA, August 2000.

[8] 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.

[9] E. V. Carrera, S.Rao, L.Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.

[10] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-298-96, University of Michigan, Ann Arbor, MI, 1996.

[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] 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)*, pages 319–332, Oct. 2000.

[13] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[14] T. Heath, R. Martin, and T. D. Nguyen. Improving Cluster Availability Using Workstation Validation. In *to appear in Proceedings of the ACM SIGMETRICS 2002*, Marina Del Rey, CA, June 2002.

[15] D. B. Ingham and G. D. Parrington. Delayline: a wide-area network emulation tool. *USENIX, Computing Systems*, 7(3):313–332, 1994.

[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] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 336–344, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[18] Manoj Joshi and Nicholas Brenton, Cisco Systems, Jim Helfrich and Patrick Jones, Network Appliance. High availability for network-attached storage. Technical Report TR 3115, Network Appliance and Cisco Systems, 2000.

[19] J. F. Meyer. Performability evaluation: Where it is and what lies ahead. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 334–343, Erlangen, Germany, Apr. 1995.

[20] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, pages 341–353, 1995.

[21] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.

[22] 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.

[23] 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*, pages 1–15, Charlston, SC, Dec. 1999.

[24] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. FIAT w fault injection based automated testing environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, 1988. IEEE Computer Society Press.

[25] D. Siewiorek, J. Hudakund, B. Suh, and Z. Segall. Development of a benchmark to measure system robustness. In *In Proceedings 23rd International*

Symposium Fault-Tolerant Computing, pages 88–97, 1993.

[26] R. M. Smith, K. S. Trivedi, and A. V. Ramesh. Performability Analysis: Measures, an Algorithm, and a Case Study. *IEEE Transactions on Computers*, 37(4), April 1998.

[27] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100, March 2000.

[28] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The 1999 Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[29] 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.

[30] University of Utah Flux Research Group. Utah Network Testbed. Available at http://www.emulab.net.