

# Using Fault Model Enforcement to Improve Availability

Kiran Nagaraja, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen  
{knagaraj, ricardob, rmartin, tdnguyen}@cs.rutgers.edu

Department of Computer Science, Rutgers University  
110 Frelinghuysen Rd, Piscataway, NJ 08854

*If the facts don't fit the theory, change the facts.*  
—Albert Einstein

**Abstract**—Today's network services run on complex arrays of computing systems consisting of a myriad of hardware and software components. In this work, we claim that it is impractical to try to tolerate all (or even a significant fraction of) fault types in these systems. We argue instead that a new approach, called fault model enforcement, that maps actual faults to expected faults of an abstract fault model can be used to increase system availability. This enforcement approach works because it transforms faults not factored into the initial design into faults that the system was designed to tolerate. Using fault injection and analytic modeling, we show that this enforcement approach has the potential to decrease the unavailability of a distributed Web server by over 50%.

## I. INTRODUCTION

Today's networked services are complex beasts, bedeviled by software bugs, operator errors and hardware failure. In spite of increasing hardware reliability, the availability of systems does not appear to be improving much with time [1], [2], [3]. Building highly available systems remains an art guided by intuition and experience rather than by quantitative techniques.

The continual increase in the complexity of computer systems is a significant contributor to the availability problem. For example, current network-based services rely on a constellation of interconnected systems—a typical example is cluster-based systems [4]. Traditional database systems using a multi-tiered approach [5] also suffer from a similar complexity problem. Distributed arrays of processing, storage, and networking hold promise of increased availability; however, their massive complexity is difficult to control. Indeed, a typical machine room for one of these services centers around a set of closely guarded racks and data closets woven together by intricate networks for communication, storage, power, and cooling. Modeling every possible disk or cable failure, software bug, or operator glitch is simply not tractable. Buzz-phrases such as “design so everything can fail” offer little quantitative guidance. At best, such folklore serves as a warning against complacency.

We argue that these complex systems do not lend themselves to traditional fault-tolerance techniques. For example, quorum systems are often put forth as highly effective fault-tolerance techniques [6], [7]. In these approaches, each computation is replicated and a process of consensus is used to determine the

correct one. However, this approach is ill-suited for networked services for two reasons. First, it is too expensive to replicate and run consensus for every sub-computation. Second, the act of combining the results together introduces the potential for bugs, and thus faults.

While approaches for increasing the reliability of hardware components [8] have enjoyed enormous success, we argue in Section II that much of this arises because real circuit failures map closely to the designer's fault model. The same is not true for today's networked services. A resistor can only fail in a few ways; a distributed database practically infinitely many. Thus, we argue that these approaches are also not appropriate for improving the availability of networked services as they stand.

In this work, we present a novel strategy, *Fault Model Enforcement* (FME), for increasing system availability in the context of networked services. Our thesis is that it is impossible to model all possible faults and to build a system that is tolerant of all faults. Thus, the crux of the FME approach is to explicitly define an abstract fault model during design and to force the system to behave like this abstract fault model at runtime, possibly by actively causing components to fail. Instead of building ever more elaborate fault-tolerance mechanisms, FME reduces complexity by reducing the state space of a live system, allowing the designer to concentrate on tolerating a specific fault model.

We present an initial study of FME in the context of the PRESS distributed Web server [9], [10]. In particular, we use a combination of fault injection and modeling to quantify FME's potential for increasing PRESS's availability. At this point, we do not yet have an implementation of FME. Rather, our quantification is computed on a hypothetical FME infrastructure.

Our work is motivated by a recent effort to study the behavior of PRESS under faults [11]. We found that PRESS performed poorly even when subjected to only a modest range of common faults, including disk hangs, link and switch failures. We soon concluded that the designer's implicit fault model did not match the faults we were injecting. While it was certainly possible to improve PRESS's availability by expanding the fault model and adding code for tolerating the additional faults, we were discouraged that accounting for many plausible faults was an impossibly complex task. Further, accurately mapping runtime symptoms (e.g., TCP connection timeout) observable in a distributed system to the actual fault in a complex fault model is also extremely difficult, if not impossible. Thus, after many more fault-injection experiments, we decided that reducing the

possible state space using FME was a necessary and viable approach for dealing with this complexity.

We use a methodology that we recently introduced [11] to evaluate the performability of several versions of PRESS when running with and without FME. This methodology combines measured responses to fault injection and analytic modeling. The results of the study show that a PRESS server following an FME approach has the potential to decrease the unavailability by over 50%. This increase in availability was insensitive to the stability of the underlying system. It was, however, quite sensitive to the type of fault detection used; a poor fault detector lessens the effectiveness of the FME approach.

The rest of the paper is organized as follows. We present background on fault models in Section II. We then describe FME in more detail in Section III. Section IV briefly covers our fault-injection and modeling methodology. We present the results of our fault injection experiments in Section VI. Section VII models the availability and performability improvements for different versions of PRESS using FME. In Section VIII, we speculate on the strength and limitations of FME, and describe future research directions. Finally, section IX draws our conclusions.

## II. BACKGROUND

When building a fault-tolerant system, all designers rely on an *abstract fault model* of the system. The fault model may not be explicit or well-defined, but at least it always lurks in the designer's mind. We argue that the disconnect between the faults' impact on the system and the designer's fault model is the source of many failures in computer systems. More specifically, we define an abstract fault model as:

- The set of component types.
- The interfaces for each component.
- The interactions between components.
- The set of possible faults.
- A set of symptoms, or observable fault-detection events.
- The component behavior during faults.

An often overlooked property of fault models is the connection between model accuracy and the delivered level of fault tolerance. There are several dimensions of accuracy. Some are straightforward to reason about. For example, assigning an MTTF and MTTR of various components as well as mapping probability distributions to them is subject to error, and we can use well-known statistical techniques to reason about the impact of possible errors.

However, in complex systems more subtle forms of inaccuracy creep in. In our example PRESS system, we never modeled SCSI cables, although these are a common source of error [12]. Many components are not modeled. Many interfaces and interconnections are ignored, e.g. the return codes from the operating system [13]. The set of all possible faults must be restricted to something tractable, typically a few tens at best. Many assumed symptoms are just plain wrong; e.g., fail-stop behavior is typically assumed but often not true [14]. Symptoms are often attributed to the wrong failure; e.g. TCP assumes that packet loss is due to congestion, not link failure. TCP thus probes the network for minutes looking for a new bandwidth point, while in a LAN, fail-stop would be a better choice. Human errors are

never taken into consideration, even though they are a common problem [15], [3]. In general, many fault states are simply not accounted for in a fault-tolerant design because there are just too many of them.

In other fields, the situation is not nearly so grim. For example, in the context of circuit design, the set of components are elements such as resistors, capacitors, and power supplies. The interfaces to the components are the sources and drains. The circuit defines the interconnections, and the component behavior is to deviate from a specified resistance or capacitance according to a normal distribution. Such an analytic, implicit fault model can be found in the Six-Sigma approach as applied to circuit design [16].

In the context of complex computer systems, such as Web servers and databases, the fault modeling situation is much more nebulous. What is an accurate, yet workable fault model is ill-defined. For example, the set of components might be the network switches, network cables, workstations, and disk drives. This definition still omits a large number of components, e.g. power supplies, SCSI cables, and the operating system. The interfaces for these components might be the messaging interconnect for the switches and links, and the read/write interface between the disks and workstations. Moving to the set of possible faults, one must further abstract away from reality. To reason about a real system, we usually model all components as fail-stop. We also hope that failure rates and recovery occur with exponential distributions, even though there is strong empirical evidence against this, at least for workstations [17], [18].

## III. FAULT MODEL ENFORCEMENT

Given the previous description of complex computer systems, creating reasonably accurate abstractions of them seems to be an impossibly complex task. There are too many different subsystems, none of which any one person fully understands, connected by bewildering protocols, and each one could fail in unexpected ways. It seems as if designs will always be plagued with a high degree of uncertainty, and so only extremely expensive, even paranoid, over-engineering can deliver something like five-nine availability.

However, there is a property of complex system which works in the direction of reducing complexity: the subsystems are designed around simplified abstractions. For example, a disk interface can be thought of as a linear array of fixed-sized blocks; a network interface as a variable size packet-delivery service. Fault model enforcement attempts to make the real system conform to these simpler models. It does this by enforcing the abstract fault model in a fault-tolerant design when possible. Even if the component in question did not actually fail, sometimes the system causes it to stop working to better fit what the designer intended. In particular, we are concerned with faults which although detectable, may lead us to mis-diagnose the fault. Often, this is because it is impossible to discern among multiple faults because they map to the same symptom. Distinguishing between them would require excessive monitoring hardware or software. In addition, often the system does not have a clear recovery action for these faults, because the original designers never considered it. Under these circumstances, an FME approach selects a particular fault and forces the system to act that

way.

We have found two FME strategies applicable for a wide range of situations. In a nested system, one FME approach has a failed sub-component cause the failure of an outer component. For example, if a disk fails, an FME approach may crash the entire workstation. Less drastic measures are also possible, e.g., if a packet is lost, an FME approach may only close the channel. This approach is somewhat reminiscent of fail-stop and extended fail-stop approaches; in essence, FME is enforcing a fail-stop model on the outer component. FME differs from these approaches because it selectively causes the active failure of another component according to an overall design.

A second FME strategy is to make sure that after a given symptom is observed, the expected fault behavior happens. This is useful when multiple faults may be the cause of a symptom observable at runtime. If not all of these faults were included in the fault model, then the system may incorrectly diagnose the fault. Thus, in FME, when a symptom occurs, we would ensure that one of the expected fault occurs, even if it requires the active failing of a non-faulty component. For example, if the abstract fault model of a system only includes node failures, not link and switch failures, then the system might incorrectly diagnose the loss of communication to a node as a node failure when the actual fault is a link or switch failure. In this case, the system's recovery action may not be effective to tolerate the fault and recover full functionality once the faulty component has been repaired. In this case, FME might actively create the expected failure by cycling the power to the workstation to force a reboot. Of course, if the link failure is not transient, someone has to actually detect the fault and correct it. FME only ensures that the fault is mapped to one expected by the software system so that it can initiate recovery actions to regain full functionality.

#### IV. METHODOLOGY

We will use the methodology that we recently proposed for evaluating performability [11] to demonstrate the potential of FME to increase the availability, and therefore the performability of cluster-based systems such as PRESS. This methodology consists of two phases. In the first phase, the evaluator defines the set of all expected faults, then uses a fault-injection system to inject them (and the subsequent recovery) one at a time into the system being evaluated while it is running live. 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 requests served per second and availability is the percentage of requests served successfully.

To ensure consistency, the evaluator must use a benchmark that drives the system to deliver a relatively stable throughput throughout the observation period. This is necessary to decouple the measured performance from the injection time of a fault. Also, the system should be exercised close to its peak performance as defined by the bottleneck component. For example, for PRESS, the bottleneck component is the CPU. Thus, we will be running at 90% CPU utilization for all of our experiments.

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

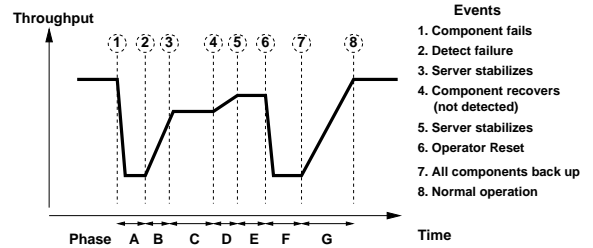


Fig. 1. The 7-stage piece-wise linear model specified by our methodology for evaluating the performability of cluster-based servers. Time is shown on the X-axis and throughput is shown on the Y-axis.

of each component. Figure 1 illustrates our 7-stage model of service performance in the presence of a fault. 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 often occurs 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.

To parameterize the model, we need two parameters for each stage: (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.

Finally, to combine the effects of the faults, 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. 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  $W_c = (\sum_{s=A}^G D_c^s) / MTTF_c^1$ .

## V. 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 [9], [10]. Like other locality-conscious servers [19], [20], [21], [22], 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, it decides, based in the content requested, 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.

**Communication architecture.** PRESS 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.

PRESS can use either TCP or VIA for intra-cluster communication. The TCP version basically has the same structure of its VIA counterpart; the main differences are the replacement of the VI end-points by TCP sockets and the elimination of flow control messages, which are implemented transparently to the server by TCP itself.

**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 continuously 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 can be connection or heartbeat-based. In the heartbeat-based approach, PRESS 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. In the connection-based approach, a node assumes that another node has failed if the TCP connection between them is

broken. In this implementation, nodes are also organized in a directed ring, but only for recovery purposes.

The fault detection when VIA is used for intra-cluster communication is also connection-based. PRESS relies on our VIA implementation to maintain node and communication up/down information. Any detected faults cause the corresponding connections to be terminated/broken. Again, nodes are organized in a directed ring 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 our experiments, we eliminated the IP take-over part of the recovery process, so we do not discuss IP take-over further.) 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. 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 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 [10]. Table I 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 throughputs of the various versions of PRESS will be compared against throughput when various faults are injected into the cluster.

All PRESS versions cooperate in caching files, but differ in terms of their approach to detecting failed nodes, and the performance of their intra-cluster messaging. TCP-PRESS and VIA-PRESS-0 use connection breaks to detect node failures, whereas TCP-PRESS-HB uses heartbeat messages. The heartbeats were added to overcome the same shortcomings of TCP's implicit fault model as were recognized by [23]. However, TCP-PRESS-HB's heartbeat system is much less elaborate; for example, it does not include authentication checks.

In TCP-PRESS and TCP-PRESS-HB, all messages involve data copies on both sides and interrupt-driven message reception. VIA-PRESS-0 also utilizes regular messages, but they are sent directly from user-space.

## VI. PRESS BEHAVIOR UNDER SINGLE-FAULT LOADS

We now look at the behavior of PRESS when faults are injected in isolation. Recently, we have studied PRESS under a comprehensive set of faults including: failure of network components, node crashes and hangs, disk faults, operating system

<sup>1</sup>We refer the interested reader to [11] for an explanation of why the denominator is just  $MTTF_c$  instead of  $MTTF_c + MTTR_c$ .

Version	Main Features	Expected Behavior	Throughput
TCP-PRESS	TCP used for intra-cluster communication; connection breaks are used as trigger for reconfiguration	Performance may suffer in the presence of faults because TCP connection timeouts are typically lengthy	4965 reqs/sec
TCP-PRESS-HB	TCP used for intra-cluster communication; loss of heartbeat messages are used as trigger for reconfiguration	Faster response to faults but may give false positives if communication of heartbeats is delayed	4965 reqs/sec
VIA-PRESS-0	VIA used for intra-cluster communication; connection breaks are used as trigger for reconfiguration	Better throughput than the TCP versions, but does not use heartbeats	6031 reqs/sec

TABLE I

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

resource exhaustion, and application crashes and hangs [11], [24]. This set is comprehensive with respect to PRESS in that it covers most of the resources that PRESS uses in providing its service. In this section, we will discuss a few of these fault experiments; in particular, we will focus on those that expose the mismatch between the fault model assumed by PRESS and the actual faults. Data for all fault experiments can be found at <http://www.panic-lab.rutgers.edu/Research/mendosus/>.

#### A. Experiment Design

In all experiments, we run a four-node version of PRESS on four 800 MHz PIII PCs, each of which is equipped with 206 MBytes of memory and 2 10,000 RPM SCSI disks. Nodes are interconnected by 1 Gb/s lines to a cLAN VIA switch. We can communicate with TCP or VIA over this network. PRESS was allocated 128 MBytes on each node for its file cache; the remainder of the memory was sufficient for the operating system and the server code, such that no swapping took place during the experiments.

The workload for all experiments is generated by a set of clients running on separate machines. To stress the communication aspect of PRESS, our experiments only involve static content and the entire set of documents is replicated at each node. The client machines are connected to PRESS by the same cLAN network that connects the nodes of the cluster. Using a single network for communication is not at all a problem. The total network traffic does not saturate any of the cLAN network interfaces, links, or the switch, and so the interference between the two classes of traffic is minimal in our setup. Furthermore, our fault-injection infrastructure 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.

#### B. Internal Link Failure

In this section we discuss the behavior of PRESS under a single transient link failure. Figure 2 shows the effect of the fault on the 3 versions of PRESS. Observe that in Figure 2, TCP-PRESS stalls for the entire period of the fault. The TCP protocol on the cooperating nodes keeps trying to re-send packets across the faulty component, causing the filling up of communication

queues on all nodes, thus dropping the throughput to zero until slightly after the component recovers and the messages start flowing again. Note that the fault does not last long enough for TCP timeouts to occur. These timeouts tend to be very long, on order of 10-15 minutes.

In contrast, TCP-PRESS-HB detects the fault in a very short time and reconfigures. The reason is that the heartbeat messages lost over the faulty link cause the other nodes to assume that the unreachable node is down. This splinters the cluster into 3 cooperating nodes and 1 independent node. The detection and recovery durations correspond to a failure detection threshold of 15 seconds (3 heartbeats) used by the heartbeat code.

Similar to TCP-PRESS-HB, VIA-PRESS-0 detects the failure almost instantaneously because all connections to the unreachable node break. This splinters the cluster into sub-clusters, as in the case of TCP-PRESS-HB.

TCP-PRESS-HB and VIA-PRESS-0 do not reconfigure back into a single cluster once the link returns to normal operation. This surprising behavior arises from a mismatch between the fault model assumed by PRESS and the actual fault. PRESS assumes that nodes fail but links and switches do not. Thus, reconfiguration only occurs at startup and on loss of 3 heartbeats; nodes do not merge again after partitions. This presents a typical case where FME would be applicable. A possible enforcement action would be to restart the faulty node, so that it can re-integrate with the rest of the servers.

Additionally, in this failure, we observe two contrasting approaches to fault detection and recovery. TCP, by default being more conservative, interprets the failure as transient congestion, and waits long enough so that, the component actually recovers before timeout. Introducing a more aggressive fault-detection protocol, such as in the TCP-PRESS-HB and VIA-PRESS-0 versions, proves to be detrimental. Under short transient failures, it splinters the cluster into sub-clusters, quite unnecessarily. Return to normal operation thus requires the intervention of an administrator to restart all but one of the sub-clusters. This, in effect, makes these versions *less* available than the basic TCP-PRESS in the face of relatively short transient faults. We believe, that the choice of protocol and parameter values for detection and recovery should follow reliability characteristics of the associated environment, and also the cost associated with recovering from such incorrect fault detections.

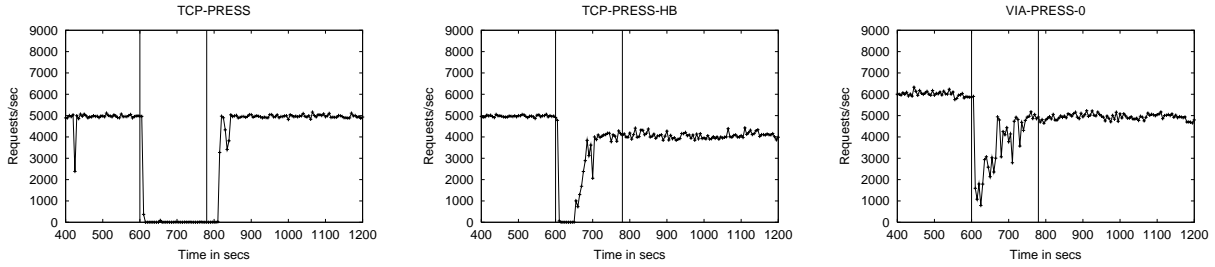


Fig. 2. Throughput of PRESS under a single transient link failure. The vertical lines show the fault injection and recovery points. Incorrect fault detection splinters the TCP-PRESS-HB and VIA-PRESS-0 into sub-clusters, delivering a degraded throughput. TCP-PRESS merely stalls during the fault period.

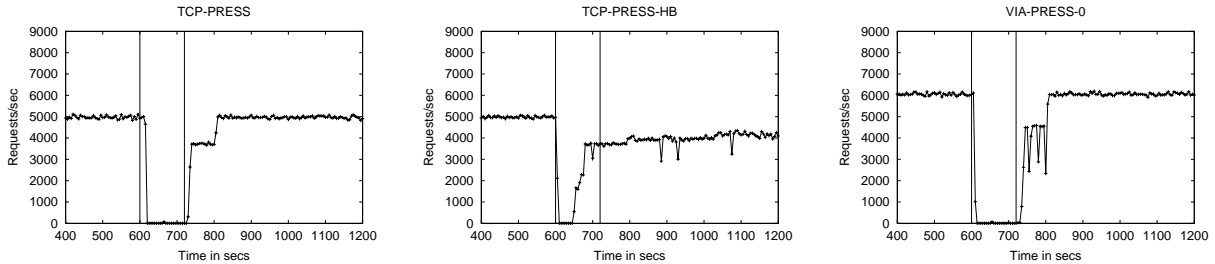


Fig. 3. Throughput of PRESS under SCSI time-out faults. The vertical lines show the fault injection and recovery points. Only TCP-PRESS-HB detects the transient fault, but assumes the affected node to be down, once again splintering the cluster.

### C. Disk Timeouts

Figure 3 shows the behavior of PRESS under SCSI timeouts. TCP-PRESS and VIA-PRESS-0 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.

TCP-PRESS-HB, 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. The main thread in PRESS is also responsible for posting the periodic heartbeats, so its blocking (as explained above) prevents heartbeats from going out. This leads the other nodes to assume the node is down, resulting in splintering. This observation once again emphasizes the need for either a more advanced fault-detection and recovery mechanism or a way to enforce the fault model assumed by the application.

### D. Node Freeze

Figure 4 shows the effects of a transient node freeze fault. Similar to the effect seen in the disk timeout faults, the node freeze stalls all communication to the faulty node. The result is that the communication queues on the other nodes fill up. Since the nodes in basic TCP-PRESS do not detect the fault, the entire server is blocked for the duration of the fault. In contrast, the nodes in TCP-PRESS-HB detect the fault, since heartbeats from the faulty node fail to arrive within the threshold interval of 3 heartbeat periods. TCP-PRESS-HB reconfigures into 3 cooper-

ating nodes and 1 singleton. When the faulty node recovers, it functions as an independent server. This splintering effect, once again, is the result of mismatch in the fault assumed by PRESS (i.e. node is down) and the actual fault.

VIA-PRESS-0 behaves similar to TCP-PRESS-HB, splintering into two sub-clusters. This is attributed to the fault detection mechanism used by the VIA protocol driver, which is similar to the heartbeat mechanism used in TCP-PRESS-HB.

### E. Application Hang

Figure 5 shows the effects of a transient server process hang. The freezing of one of the servers stalls all other servers trying to communicate with it. TCP-PRESS behaves as expected, stalling for the entire duration of the transient fault. The reaction of TCP-PRESS-HB, however, is to splinter into sub-clusters of 3 and 1 nodes.

The behavior of VIA-PRESS-0 is quite different from TCP-PRESS-HB. It does not splinter, behaving identically to the basic TCP-PRESS version. This is because VIA-PRESS-0 relies on the network driver to monitor connectivity and detect failures. Since the VIA driver is decoupled from the application (i.e. the PRESS server process), the other nodes detect no anomaly, thus avoiding the splintering effect. This leads us to believe that decoupling the fault-detection and recovery mechanisms from the main application can aid in reasoning better about the actual state of the system. However, a designer might choose a functionality similar to TCP-PRESS-HB, since splintering prevents the throughput of the entire cluster from going down to zero, and recovers the system to a reasonable state. Additionally, steps such as merging of sub-clusters after fault recovery would have to be performed to restore the cluster to its normal operating state.

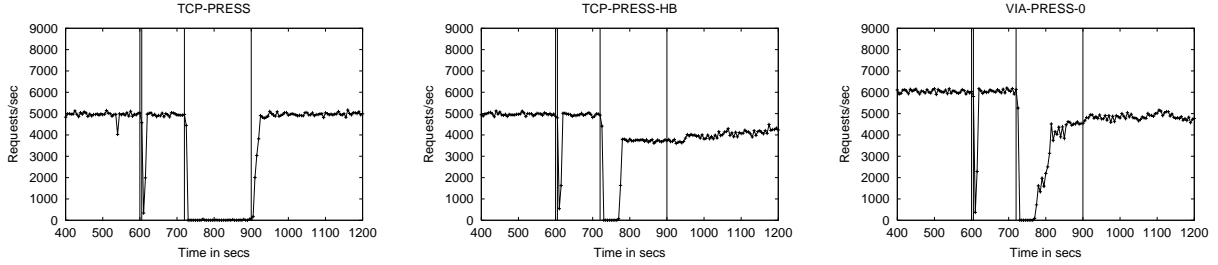


Fig. 4. Throughput of PRESS when a node freeze is injected. Both TCP-PRESS-HB and VIA-V0 detect the fault, and assuming the node to be down, remove it from the set of cooperating servers.

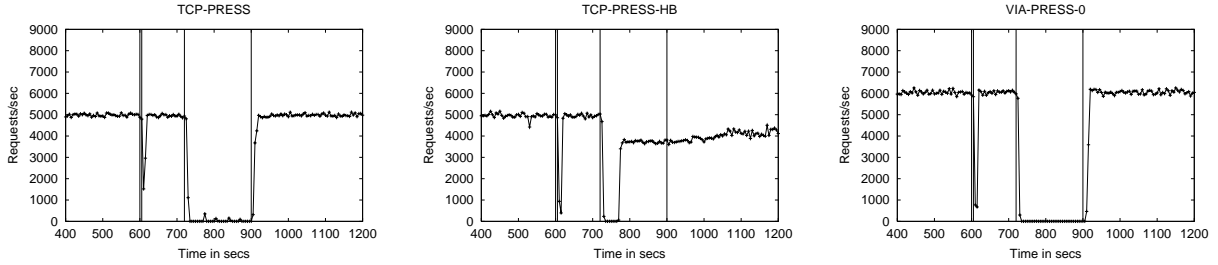


Fig. 5. Throughput of PRESS under transient application hang. Vertical lines indicate the period of fault activity. Both a long and short period are shown. TCP-PRESS-HB detects the failure and splinters into sub-clusters, but not before the fault duration exceeds the threshold of 15 seconds. A shorter fault of 5 seconds, momentarily dropped the throughput of all the 3 versions of PRESS.

## VII. PERFORMABILITY OF THE PRESS VERSIONS

We now proceed to the second phase of our methodology to evaluate the performability of the different PRESS versions. We first examine performability assuming the same fault load for all versions of PRESS. After that, we also consider what happens to availability and performability if we assume that the different versions use an FME approach. Finally, we examine the sensitivity of the FME versions to increased fault rates for application and node hangs.

### A. Fault Load

Table II gives the initial fault load used to compare the performability of the different versions of PRESS. Recall that to make the modeling tractable, we assume that faults in different components are not correlated and all fault arrivals are exponentially distributed. We have done our best to derive meaningful parameters from the available data [25], [26], [27], [18], [12], [28], [17]. A duration of 5 minutes was assumed for the operator intervention stage E and restart stage F.

### B. Evaluation Metrics

Our model computes two metrics to evaluate each server. The first is the unavailability, which is the average fraction of requests dropped. We use unavailability instead of availability because it is easier to reason about changes in unavailability compared to availability. For example, it is quite natural to call a system with an unavailability of 1% as “twice as good” as one which has an unavailability of 2%. However, the relationship between systems with a 98% and 99% availability is not so intuitive.

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

TABLE II

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

Further, we propose a combined *performability* metric that allows direct comparison of systems using both performance and availability 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 terms of “the number of nines” achieved, we believe that a log-scaled ratio of how each server compares 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.

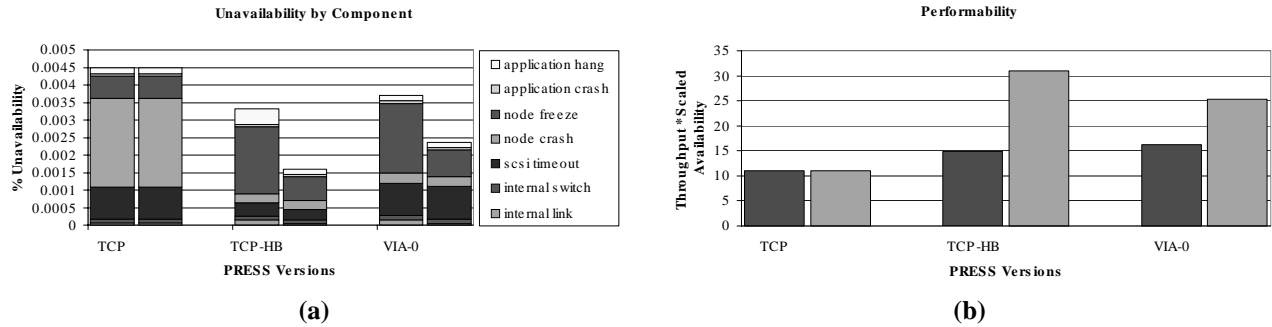


Fig. 6. The modeled (a) unavailability and (b) performability of the 3 versions of PRESS. For each group of 2 bars, the left is normal run, and the right is for the run with FME active. The combined application fault arrival rates (incl. hangs and crashes) is 1 per month, modeling a reasonably stable software.

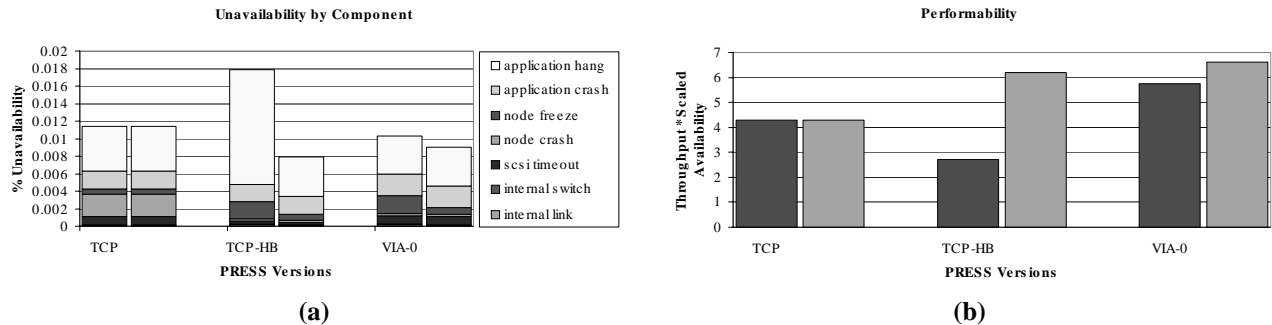


Fig. 7. The modeled (a) unavailability and (b) performability of the 3 versions of PRESS. For each group of 2 bars, the left is normal run, and the right is for the run with FME active. The combined application fault arrival rates (incl. hangs and crashes) is 1 per day, modeling a very unstable software 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. The intuition behind this relationship between unavailability ( $u$ ) and performability is that we can approximate  $\log(1 - u)$  as  $-u$  when  $u$  is small. Thus,  $\frac{\log(A_T)}{\log(1 - \frac{1}{2}u)} \approx 2 \frac{\log(A_T)}{\log(1 - u)}$ .

### C. Performability of PRESS under FME

In this section, we study the performability of the different versions of PRESS, subject to identical fault loads, both under a normal run and when the assumed fault model is enforced under failures. We model an FME approach that uses both strategies discussed in Section III. In particular, our hypothetical FME infrastructure would do the following:

- **Link down:** Reboots the node that was cut off from the main cluster.
- **Switch down:** Reboots all nodes.
- **SCSI timeout:** Reboots node with faulty disk.
- **Node crash:** Nothing. This fault was included in the abstract model.
- **Node freeze:** Reboots faulty node.
- **Application crash:** Nothing. This fault was included in the abstract model.
- **Application hang:** Reboots node on which the faulty process was running.

Note that all FME node reboots take place *after* the faulty component has recovered. Thus, at this point, our FME approach is

purely designed to help PRESS automatically re-integrate into a single cluster if the failure led to a splintering of the cluster.

Figure 6 compares (a) the modeled unavailability and (b) performability of the different versions of PRESS without (left bar in each group) and with (right bar) the enforcement of the fault model. These results show that the unavailability of TCP-PRESS-HB is cut in half under FME, while VIA-PRESS-0 also exhibits significant improvement. The basic TCP-PRESS shows no improvement, since this version did not detect short-lived failures, hence did not require enforcement.

Figure 6(a) also shows the contribution of each fault type to overall unavailability. Observe that our FME approach is successful because it was able to significantly reduce the impact of node freezes on the unavailability of TCP-PRESS-HB and VIA-PRESS-0. Recall that node freezes splinters the cluster into 2 sub-clusters which do not rejoin correctly, even after the frozen node recovers. Thus, the enforcement, which restarts the faulty node, enables the node to rejoin the cluster, hence avoiding the stages D, E, and F, where the servers are delivering degraded performance levels. For other faults discussed in our experiment, i.e., disk errors, application hangs, and link failures, our FME approach also reduced unavailability. However, the effect was less pronounced compared to node freezes.

The one exception in which our simple FME approach did not help is reducing the impact of disk errors on the VIA-PRESS-0 server. This is because FME does not reboot the node with the faulty disk until *after* the disk recovers. This helps TCP-PRESS-HB because this fault splinters this server. It does nothing to help VIA-PRESS-0, which does not splinter but degrades to 0

throughput during the fault. If we had assumed that FME was sophisticated enough to remove the node from operation during the disk failure, this would further significantly reduce VIA-PRESS-0's unavailability. This difference is what leads TCP-PRESS-HB to achieve a higher overall performability score than VIA-PRESS-0 when operating under FME.

Finally, we also looked at the benefits of FME when the application is much less stable, and susceptible to more application-induced errors. Figure 7 shows the same comparison as above, but with an increased application fault arrival rate of 1 fault per day. The benefits of FME are more obvious for the TCP-PRESS-HB, mainly due to the reduced contribution of application hangs towards unavailability, bringing the TCP version much closer to its VIA counterpart.

### VIII. DISCUSSION AND FUTURE DIRECTIONS

Our initial study of FME leaves many questions open. A few sample ones include when to enforce the model, what kind of infrastructure is needed, and how to keep the model under control. We discuss each of these questions in turn.

When to enforce the model is tricky. In the PRESS servers, enforcing the model after the symptom is detected is a workable approach in all cases, even if it is not always the best solution. However, it is far from clear if such an approach is always desirable. Applying FME more generally, if the system performs some recovery action based on a component failure, it is advisable that the system at least check through a second channel that the component did actually fail.

The second open question we pose is the kind of fault-detection organization one could use to implement FME. We considered three cases: a distributed ring embedded into the nodes themselves (similar to TCP-PRESS-HB), a separate centralized monitor, and a separate process-pair design. In all cases, having a simple, separate control path into the component is desirable, because often the main communication channel is subject to many faults types.

The first approach seems problematic. For example, we could embed the logic to reboot machines on a fault inside the PRESS server itself. This approach seems fraught with peril, because it relies on the correct operation of the FME system while it is exposed to faults in the system it is trying to enforce. For example, if a disk error hung a node, the FME system would be unable to perform actions on that node.

At the opposite end of the spectrum, the second approach places the FME logic in a separate process, on a separate computer, with access to the separate control network. In the PRESS system, this might be a separate computer running an FME program with access to the power grid. This approach is attractive because it is easy to judge the stability of the FME program, and reason about the impact of possible faults on this centralized system. However, because it is a centralized system, many single faults may bring the FME system down. The rate of these faults will, of course, determine if such a system delivers acceptable availability.

An intermediate approach might use a process-pair idea, in which dual FME processes not only monitor and enforce the fault model on PRESS nodes and servers, but also on each other. Such a system is more difficult to reason about than the central-

ized approach, but has the added advantage of being robust to single errors, as well as being easier to reason about compared with a fully distributed approach.

The last open question we address is how to prevent FME from making the system less available. If the system purposely fails components, we are opening up the door to making the system less available because bugs in the enforcement can cause needless enforcement. In general, it seems that enforcing the model can cause unnecessary downtime if the system is designed from the start to work around a particular fault. However, in our experience so far, it is better to take an FME approach and get the system back into a known state than risk error states that require human intervention to correct.

Finally, FME leaves open an important issue: when to declare a symptom a fault. For example, in PRESS, no communication to a node implied a node was down. The open question here is when do we decide to map the symptom, i.e., no communication, to a fault, i.e., the node is down. Once the system crosses the boundary of declaring a fault, FME takes the position that the system should enforce the fault. However, FME says nothing about when is the proper time to decide a fault has occurred.

### IX. CONCLUSION

In this paper we argued that designing highly available network services with comprehensive fault models is an extremely complex and error-prone proposition. Based on this, we argued for a novel strategy, called Fault Model Enforcement (FME), under which an abstract fault model that necessarily only includes a subset of all possible faults is defined at design time and enforced at runtime. To enforce the model, FME may actively fail non-faulty components to help the application initiate the correct and necessary recovery actions. We performed an initial evaluation of our proposed strategy when applied to a sophisticated cluster-based Web server. Our results demonstrated that FME has the potential to reduce unavailability by over 50%. Finally, we discussed several open questions in the design of FME-based cluster-based services.

### REFERENCES

- [1] Jim Gray, "Why do Computers Stop and What Can Be Done About It?," in *Proceedings Fifth Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986.
- [2] B. Murphy and T. Gent, "Measuring System and Software Reliability using an Automated Data Collection Process," *Quality and Reliability Engineering International*, pp. 341–353, 1995.
- [3] B. Murphy and B. Levidow, "Windows 2000 Dependability," Tech. Rep. MSR-TR-2000-56, Microsoft Research, June 2000.
- [4] Eric Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, July/August 2001.
- [5] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.
- [6] Kenneth P. Birman, "Replication and Fault-Tolerance in the ISIS System," in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Dec. 1985.
- [7] Miguel Castro and Barbara 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.
- [8] Daniel P. Siewiorek and Robert S. Swarz, *Reliable Computer Systems: Design and Evaluation (third edition)*, A. K. Peters, Ltd., 1997.
- [9] 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.
- [10] 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.

- [11] Kiran Nagaraja, Xiaoyan Li, Bin Zhang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen, "Using Fault Injection to Evaluate the Performability of Cluster-Based Services," Tech. Rep. DCS-TR-491, Department of Computer Science, Rutgers University, May 2002, Revised August, 2002.
- [12] Nisha Talagala and David Patterson, "An Analysis of Error Behavior in a Large Storage System," in *The 1999 Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.
- [13] J. Traupman P. Broadwell, N. Sastry, "FIG: A prototype Tool for On-line Verification of Recovery Mechanism," in *Proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, June 2002.
- [14] Subhachandra Chandra and Peter M. Chen, "How Fail-Stop are Faulty Programs," in *28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, June 1998.
- [15] Aaron B. Brown and David A. Patterson, "To Err is Human," in *Proceedings of the First Workshop on Evaluating and Architecting System Dependability (EASY '01)*, Gothenburg, Sweden, July 2001.
- [16] Robert V. White, "An Introduction to Six Sigma with a Design Example," in *Eleventh Annual Applied Power Electronics Conference and Exposition (APEC '92)*, Feb. 1992.
- [17] Taliver Heath, Richard Martin, and Thu D. Nguyen, "Improving Cluster Availability Using Workstation Validation," in *to appear in Proceedings of the ACM SIGMETRICS 2002*, Marina Del Rey, CA, June 2002.
- [18] M. Kalyanakrishnam, Zbigniew Kalbarczyk, and Ravishanka 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] 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*, San Jose, CA, October 1998, pp. 205–216.
- [20] 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.
- [21] E. V. Carrera and R. Bianchini, "Evaluating Cluster-Based Network Servers," in *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, August 2000, pp. 63–70.
- [22] R. Bianchini and E. V. Carrera, "Analytical and Experimental Evaluation of Cluster-Based WWW Servers," *World Wide Web Journal*, vol. 3, no. 4, pp. 215–229, December 2000.
- [23] Alan Robertson, "Linux-ha heartbeat system design," in *Proceedings of the 4th Annual Linux Showcase and Conference (ALS 2000)*, Atlanta, GA, October 2000.
- [24] Kiran Nagaraja, Neeraj Krishnan, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen, "Evaluating the Impact of Communication Architecture on the Performability of Cluster-Based Services," Tech. Rep. DCS-TR-496, Department of Computer Science, Rutgers University, Aug. 2002.
- [25] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems," *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pp. 2–9, 1991.
- [26] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, pp. 424–433, June 1995.
- [27] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. S. Trivedi., "A Methodology for Detection and Estimation of Software Aging," in *International Symposium on Software Reliability Engineering (ISSRE 1998)*, Nov. 1998.
- [28] K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and Analysis of Software Aging and Rejuvenation," in *Proceedings of the IEEE Annual Simulation Symposium*, Apr. 2000.