

Quantifying the Performability of Cluster-Based Services

Kiran Nagaraja, *Student Member, IEEE*, Gustavo Gama, Ricardo Bianchini, *Member, IEEE*, Richard P. Martin, *Member, IEEE*, Wagner Meira Jr., and Thu D. Nguyen, *Member, IEEE Computer Society*

Abstract—In this paper, we propose a two-phase methodology for systematically evaluating the performability (performance and availability) of cluster-based Internet services. In the first phase, evaluators use a fault-injection infrastructure to characterize the service's behavior in the presence of faults. In the second phase, evaluators use an analytical model to combine an expected fault load with measurements from the first phase to assess the service's performability. Using this model, evaluators can study the service's sensitivity to different design decisions, fault rates, and other environmental factors. To demonstrate our methodology, we study the performability of a multitier Internet service. In particular, we evaluate the performance and availability of three soft state maintenance strategies for an online bookstore service in the presence of seven classes of faults. Among other interesting results, we clearly isolate the effect of different faults, showing that the tier of Web servers is responsible for an often dominant fraction of the service unavailability. Our results also demonstrate that storing the soft state in a database achieves better performability than storing it in main memory (even when the state is efficiently replicated) when we weight performance and availability equally. Based on our results, we conclude that service designers may want an unbalanced system in which they heavily load highly available components and leave more spare capacity for components that are likely to fail more often.

Index Terms—Performance, availability, fault tolerance, Internet services.



1 INTRODUCTION

POPULAR Internet services frequently rely on large clusters of commodity computers as their supporting infrastructure [6]. These services must exhibit several important characteristics, including high performance, scalability, and availability. The performance and scalability of cluster-based services have been studied extensively in the literature, e.g., [3], [6], [8]. In contrast, understanding designs for availability, behavior during component faults, and the relationship between performance and availability of these services has received much less attention.

While today's service designers are not oblivious to the importance of high availability [6], [13], [30], the design and evaluation of service availability is still an art based on the practitioner's experience and intuition, rather than a scientific methodology. In this paper, we advance the state-of-the-art by developing a methodology that combines fault-injection with modeling to quantify the performability—a metric combining *both* performance and availability—of cluster-based services. This methodology has two phases. The first phase involves the exploration of the actual impact of various isolated faults on the service; as a benchmark workload is

offered to the service, we inject a single fault into each subsystem in turn and observe the service'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 microbenchmark the service for both availability and performance in the presence of faults, rather than the more traditional approach of microbenchmarking performance in the absence of faults.

The second phase of the methodology uses an analytic model to combine an expected fault load [24], [34], measurements from the first phase, and parameters of the expected deployment environment to predict overall system performability. Designers can use this model to study the potential impact of different design decisions on the service's behavior. For example, designers can study the impact of different fault detection and recovery infrastructures on availability. As part of the model, we introduce a performability metric to enable designers to easily characterize and compare service designs in terms of both their performance and availability. In fact, our metric allows designers to easily weight performance and availability differently when they want to emphasize one metric over the other.

To show the practicality of our methodology and explore the design space for high performance and highly available services, we use it to study the performability of a complex multitier service. In particular, we quantify the performability of three maintenance strategies for *soft* state, i.e., state that can be reconstructed, either automatically or with user help, in the presence of an extensive fault load. No previous work that we are aware of has quantified the relationship

• K. Nagaraja, G. Gama, R. Bianchini, R.P. Martin, and T.D. Nguyen are with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019.

E-mail: {knagaraj, gmcgama, ricardob, rmartin, tdnguyen}@cs.rutgers.edu.
 • W. Meira Jr. is with the Departamento de Ciencia da Computacao, Universidade Federal de Minas Gerais, Belo Horizonte, MG Brazil.
 E-mail meira@dcc.ufmg.br.

Manuscript received 24 June 2003; revised 9 Apr. 2004; accepted 9 Sept. 2004; published online 22 Mar. 2005.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0096-0603.

between state maintenance strategies and the performance and availability of complex, multitier services. In fact, the studies that addressed state maintenance in Internet services (e.g., [14], [20]) focused solely on demonstrating performance effects resulting from a node crash and subsequent recovery.

The service that we study is an online bookstore implemented using three tiers of Web, application, and database servers. Its soft state is comprised mostly of the contents of the clients' shopping carts. Any error that causes the soft state to be lost or become unreachable typically forces each client to repeat several requests to recreate the state. The three strategies that we study include maintaining the soft state of each client session in a database server, at a single application server, and at a pair of application servers. The bookstore's hard (persistent) state, such as its current inventory of books, is always maintained at a database server, regardless of the soft state maintenance approach.

Our results show that keeping the soft state in a database causes significant performance degradation at high loads: up to 19 percent as the systems saturate. In terms of availability, all strategies achieve between 99.9 percent ("3 nines") and 99.99 percent ("4 nines"). Looking beyond the overall availability, our results clearly isolate the impact of a large set of faults on unavailability. Unexpectedly, faults in the Web server tier caused significant unavailability for all strategies. Overall, storing the soft state in a database compares favorably against storing it in main memory, even when the state is efficiently replicated, as the former strategy reduces unavailability by a larger factor than it degrades performance. As a result, we find that the database strategy achieves the best performability when performance and availability are weighted equally. This is also a surprising result in light of recent research suggesting (qualitatively) that storing state in main memory with efficient replication would lead to better performance with the same or better availability.

The key lesson from our study is that offloading the bottleneck tier improves performance, but may also hurt availability, if any other tier loses its ability to tolerate faults efficiently. This demonstrates the necessity of a methodology such as the one we introduce here, which allows service designers to consider both performance and availability in the design of a service.

In summary, our contributions include: 1) the introduction of a methodology that combines fault injection, experimentation, and modeling to systematically quantify the availability of cluster-based services as well as their performance, 2) using our methodology to compare the behavior of three soft state maintenance approaches in a complex multitier service, and 3) showing that service designers must consider availability and cost in addition to performance when provisioning and balancing load across the tiers of a multitier service. A smaller contribution is the development of a fault-injection and network-emulation infrastructure, called Mendosus, targeted specifically for cluster-based systems. Although not described here because of space constraints, Mendosus (or a similar tool) is critical to the practical use of our methodology. The Mendosus source code and documentation are available at <http://vivo.cs.rutgers.edu/mendosus.html>.

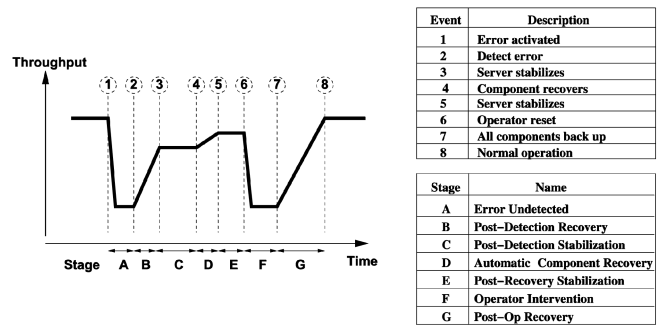


Fig. 1. The seven-stage piecewise linear template specified by our methodology along with events defining transitions between stages.

2 QUANTIFYING PERFORMANCE, AVAILABILITY, AND PERFORMABILITY

As already mentioned, our methodology for evaluating the performability of cluster-based services is comprised of two phases: a measurement phase based on fault injection and a modeling phase that combines an expected fault load together with measurements from the first phase to quantify service performability. We now describe these two phases in detail. Then, we introduce a performability metric that combines performance and availability. With respect to the underlying metrics, we equate performance with throughput (requests served per second) and define availability as the percentage of offered requests served successfully. Note that both these metrics view all requests as being equal. That is, any successfully served request is counted equally for service throughput and availability, regardless of the type of the request. Similarly, any failed request counts equally toward service unavailability.

2.1 Phase 1: Characterizing Service Behavior under Single-Fault Fault Loads

In this phase, the evaluator first defines the set of fault types that will drive the availability study. Then, the evaluator describes the service's response to a single fault of each type by injecting the fault into a live execution, measuring the system's behavior during the fault, and fitting the results to a general seven-stage piecewise linear template. The live execution is driven by a constant benchmark workload that is offered to the service. The measurement of the live system is a critical part of our methodology as it quantifies the system's response to injected faults, including the impact of any existing mechanisms for fault detection, tolerance, and/or recovery.

Fig. 1 illustrates our seven-stage template. Time is shown on the X-axis and throughput is shown on the Y-axis. The template starts with the activation of an error when the system is running error-free. Stage A, the *Error Undetected* stage, models the degraded throughput delivered by the system from the time when an error is activated because of a component fault to when the system detects the error. The *Post-Detection Recovery* stage, or Stage B, models the transient throughput delivered as the system reconfigures to account for the error. We model the throughput during this transient period as the average throughput for the period. After the system stabilizes, throughput will likely remain at a degraded level because the faulty component has not yet recovered, been repaired, or replaced. Stage C, *Post-Detection Stabilization*, models this degraded perfor-

mance regime. Stage D, or *Automatic Component Recovery*, models the transient performance after the component recovers. After D, while the system is now back up with all of its components, the application may still not be able to achieve its peak performance (e.g., it was not able to fully reintegrate the newly repaired component). The *Post-Recovery Stabilization* stage (E) models this period of stable but suboptimal performance. Finally, stage F, or *Operator Intervention*, represents throughput delivered while the service is reset by the operator, whereas stage G, or *Post-Op Recovery*, represents the transient throughput immediately after reset.

For each stage, we need: 1) the average throughput delivered during that stage and 2) the length of time that the system will remain in that stage. The first is always a measured quantity. The second parameter may be an assumed environmental value that is supplied by the evaluators. For example, the time that a service will remain in the *Post-Detection Recovery* stage is typically measured; the time for the *Post-Detection Stabilization* and the *Post-Recovery Stabilization* stages are typically supplied as parameters (see Section 2.2). In phase 1, the evaluator measures the times of stages that are not dependent on assumed environmental values, leaving the remaining times as parameters for phase 2. In essence, in phase 1, the evaluator derives all measurements necessary for the most general instantiation of the seven-stage template for the error caused by each fault type, assuming that all faults of a particular type always lead to the same error and performance impact.

With respect to the fault injection, the service must have completely recovered from a fault (or have been restarted) before the next one is injected. Further, each fault should last long enough to actually trigger an error and cause the service to exhibit all stages in the seven-stage template unless the service does not exhibit some of the stages for the particular fault. For example, if there are no warming effects, then stages *Post-Detection Recovery*, *Automatic Component Recovery*, and *Post-Op Recovery* would not exist. In these cases, the evaluator must use his/her understanding of the service to correctly determine which stages are missing and set their times to 0.

2.2 Phase 2: Modeling Performance and Availability under Expected Fault Loads

In the second phase of the methodology, the evaluator uses an analytical model to compute the expected average performance and availability, combining the service's behavior under normal operation, the behavior during component faults, and the rates of fault (mean time to failure, MTTF) and repair (mean time to repair, MTTR¹) of each component. To simplify the analysis, we assume that errors triggered by different faults do not overlap (although the faults themselves can overlap) so that only a single error is in effect at any point in time. This assumption is realistic when faults are independent since MTTFs are typically

quite long compared to the time each error is in effect. The assumption may not be realistic when faults are correlated. We describe how our methodology deals with correlated faults in Section 2.3. Nevertheless, the assumption allows us to add together the various fractions of time spent in degraded modes given a seven-stage template for each fault type gathered in phase 1.

Our analytical model is thus comprised of the following equations. (To simplify the discussion, we assume that each fault immediately activates an error, although this does not have to be true in general.) If *Normal_Thruput* is the service's performance under normal operation, c is a fault type (such as node crash), $MTTF_c$ the mean time to failure for c , $Thruput_c^s$ the average performance of each stage s in Fig. 1 for a fault of type c , $Duration_c^s$ is the duration of each stage, $Sessions_Affected_c$ the average number of user sessions affected by the fault of c , and $Recreate_State_c$ the average number of requests required to recreate the soft state of each session, our model leads to the following equations for average performance (*Avg_Thruput*) and average delivered availability (*Avg_Avail*) for a given fault load:

$$Avg_Thruput = \left(1 - \sum_c Fraction_Degraded_c\right) \times Normal_Thruput + \sum_c \left(\sum_{s=A}^G \left(\frac{Duration_c^s}{MTTF_c} \times Thruput_c^s \right) - \frac{Sessions_Affected_c \times Recreate_State_c}{MTTF_c} \right) \quad (1)$$

$$Avg_Avail = \frac{Avg_Thruput}{Normal_Thruput}, \quad (2)$$

where $Fraction_Degraded_c = (\sum_{s=A}^G Duration_c^s) / MTTF_c$. Intuitively, $Fraction_Degraded_c$ is the expected fraction of the time during which the system operates in the presence of fault type c .² Thus, the $(1 - \sum_c Fraction_Degraded_c) \times Normal_Thruput$ factor above computes the expected performance when the system is free of any fault, whereas the $\sum_{s=A}^G (Thruput_c^s \times Duration_c^s / MTTF_c)$ factor computes the expected performance when the system is operating with a single fault of type c . The $Sessions_Affected_c \times Recreate_State_c / MTTF_c$ factor represents the sets of requests that will need to be reissued as a result of this type of fault. (Recall that a fault may cause some soft state to be lost, forcing the affected clients to reconstruct their soft states by reissuing the requests that were completed prior to the occurrence of the fault.) $Sessions_Affected_c$ and $Recreate_State_c$ can be computed from the workload that drives phase 1. The average delivered availability, $Avg_Thruput / Normal_Thruput$, computes the fraction of the fault-free performance that is delivered under the fault load. The average delivered unavailability, $Avg_Unavail$, can then be computed as $1 - Avg_Avail$.

1. Note that *Error Undetected* + *Post-Detection Recovery* + *Post-Detection Stabilization* should equal MTTR. Typically, the lengths of the *Error Undetected* and *Post-Detection Recovery* stages are measured, whereas the length of the *Post-Detection Stabilization* stage is set to $MTTR - Post-Detection Stabilization - Post-Detection Recovery$.

2. The denominator of $Fraction_Degraded_c$ is just $MTTF_c$ instead of $MTTF_c + \sum_{s=A}^G Duration_c^s$ because multiple faults of the same type can occur "concurrently" and be "queued" by the system according to our basic nonoverlapping errors assumption. In practice, because $MTTF \gg \sum_{s=A}^G Duration_c^s$, the numerical impact of this difference in assumptions is minimal.

Note that we can use the above equations to study “what if” scenarios by changing any of the $Throughput_c^s$, $Duration_c^s$, $Sessions_Affected_c$, and $Recreate_State_c$ parameters. Of course, we can also study the impact of different fault loads by changing MTTFs and MTTRs.

2.3 Correlated Faults

Our availability quantification methodology as presented thus far assumes that errors do not overlap. The methodology can easily be extended to consider correlated faults with overlapping errors, however. The key observation is that a set of correlated faults can be treated as a single fault in terms of the resulting service behavior. Thus, in phase 1 of the methodology, the evaluator needs to define the sets of correlated faults that can be expected and inject those sets one at a time to observe the system’s response. It is possible that the response will deviate from our seven-stage template. However, it is enough to determine the entire loss in performance as a result of the correlated faults; all we lose is the ability to evaluate “what-if” scenarios where we change the duration or performance of different stages independently. In phase 2, we can then operate with the performance losses (from individual *and* correlated faults) and MTTFs using a model similar to that described in the previous subsection.

We do not consider correlated faults further in this paper due to the limited amount of publicly available data on such faults.

2.4 Performability Metrics

Having presented our model for computing the average throughput and availability in the presence of faults, we now propose a performability metric that combines these measures into a single number. Hitherto, performability has mostly been defined as a measure of the expected average performance in the presence of faults; a measure similar to our average throughput ($Avg.Throughput$). We believe that this measure does not adequately capture the importance of unavailability. For example, two different systems can exhibit similar average performance in the presence of faults, but substantially different unavailabilities. The root cause of this effect is that similar availabilities hide significant differences in unavailability. For example, 99 percent and 99.9 percent availabilities differ by less than 1 percent, whereas the corresponding unavailabilities (1 percent and 0.1 percent, respectively) differ by an order of magnitude. This effect is clearly undesirable when we want to highlight the availability properties of different systems. Thus, we propose a new *performability* metric that can emphasize unavailability more strongly. In its most general form, our approach is to multiply (a power of) the throughput under normal operation by (a power of) an unavailability penalty factor as follows:

$$Perform = Normal.Throughput^{pw} \times \min\left(1, \frac{Target.Unavail}{Avg.Unavail}\right)^{uw}, \quad (3)$$

where $Perform$ is the performability of the system, pw and uw are performance and unavailability weights, respectively, and $Target.Unavail$ is the target average unavailability. The

target unavailability is defined by the service designer. It serves as a scaling constant so that $Perform$ approaches a function of the normal throughput as the achieved availability approaches the target. An example unavailability target might be 0.00001, as we use in our case study.

Our metric is general in that it can easily assign different importance to performance and unavailability. However, we typically favor setting $pw = uw = 1$, which makes performability scale linearly to both performance and unavailability; two systems are equal if and only if they trade off equal percentages of performance and unavailability. We use these unit-weight settings in our case study as well.

3 CASE STUDY

We now show how our methodology can be used to evaluate the impact of different designs on the performability of cluster-based services. In particular, we use our methodology to evaluate the impact of three different strategies for managing soft state in a multitier bookstore service. (We have also successfully used our methodology to study the performability trade offs of different designs and environmental factors, including the impact of different communication protocols, error recovery strategies, hardware reliability, and operator response time, on a cluster-based Web server [27], [25], [26].) In this section, we first give some background on multitier services, discuss the state maintenance strategies that we will study, and present some details of the bookstore service. We then describe our experimental environment and present results derived from the two phases of our methodology. Finally, we discuss the implications of our results.

3.1 Background: Multitier Internet Services

The first Internet services were supported by clusters of Web servers that were mostly responsible for serving static content (HTML files and images) and a relatively small amount of dynamic content (mostly generated by CGI scripts). In essence, they were single-tier systems placed behind a load-balancing switch or a round-robin DNS server. Current Internet services are much more complex. They are now organized in multiple tiers of clustered servers that cooperate to serve an increasing amount of dynamic content. These multitier services now support a multitude of e-commerce applications, ranging from simple online stores to stock trading to business-to-business commerce services.

Perhaps the most common multitier service architecture is the three-tier organization, comprised of Web, application, and database servers. The Web servers provide a front-end to the service, possibly providing an authentication and security layer in addition to serving HTML pages. The application servers implement the application logic, whereas the database servers store the core content of the service and provide access to it with ACID semantics. Some more complex three-tier architectures also include image servers and/or server-side caches in the same tier as the Web servers. Also, the server cluster is typically placed behind two or more devices (one device is used for fail-over purposes) that balance the load across the Web servers. For

simplicity, we only focus on the basic three-tier architecture from now on.

Client requests may flow from the first to the last tier of the architecture (and back). In more detail, a client sends an HTTP request to the service containing the appropriate URL and possibly some parameters. The request is initially processed by the least loaded Web server. If the request is for a static file, the Web server can service it immediately. If the request requires access to dynamic content, the Web server passes it to one of the application servers. Typically, this application server will issue a number of queries to the database servers and will format the results as an HTML page. This page is passed back to the original Web server, which sends it to the client. Subsequent dynamic requests from the same client are typically served by the same application server as discussed below.

3.2 State and State Maintenance Strategies

In the context of client/server systems, the notion of “state” is usually defined as any data that can be affected by a client request. Modern Internet services deal with two types of state: hard and soft. Hard state cannot be reconstructed easily or at all, so it has to be persistent and durable. Examples of hard state are: stock information about the available books in an online book store, the e-mail messages received by a user of an e-mail service, and the highest bid for each item available in an auction service.

Accesses to hard state may or may not require full ACID semantics. The first and third examples above clearly do, but the second may not. For an e-mail service, strong consistency might not be required: For example, it may be fine to deliver messages to a mailbox slightly out of order. Nevertheless, all hard state is typically stored in databases to guarantee persistence and durability.

Soft state is state that can be easily reconstructed, either automatically or with user help, and, so, does not need to be persistent. Internet services deal with a variety of soft state, such as user profiles, navigation tracking information, the contents of HTML forms, and client session information. Session state is particularly interesting in that it contains information about a series of sequential requests (known as a “session”) from a single user. If the user does not access the service for some time (e.g., 30 minutes), the session is assumed over and the session state is discarded. Shopping carts are the most common example of session state.

In addition to being reconstructable, soft state typically does not require full ACID semantics. Thus, soft state is usually stored entirely at the application servers or divided between the application servers and the clients themselves.

In this paper, we study the performability impact of three different state maintenance strategies, which we call Standard, DB State, and second-tier Replication. Next, we describe these strategies.

3.2.1 Standard

As already mentioned, in the Standard approach, each client’s soft state is stored at an application server as a memory object that is tied to the server’s session control mechanism. The major advantages of Standard are scalability and implementation simplicity and flexibility. The main drawback of this approach is its potentially low

availability. When an application server fails, all sessions maintained by the failed server and their accompanying states are lost. This can impact both service performance and availability. Performance may suffer because the affected clients may need to recreate their soft state by resubmitting requests that have already been processed. Availability is affected because the loss of state is a visible failure that may in fact degrade user satisfaction.

3.2.2 DB State

DB State uses a conservative approach to storing soft state, treating it as regular database records that are maintained with full ACID properties. DB State should provide the best availability since state is stored on what is typically the most reliable/available (and expensive) component of the service.

The disadvantages of DB State, on the other hand, include the potential loss of performance and scalability. Maintaining soft state in the database means that the service is maintaining stronger consistency semantics than are needed for the data, increasing the load on the service and decreasing parallelism. More critically, this migration of load from the application servers to the database may make the database the performance bottleneck; this is undesirable because scaling the database is often difficult and human-intensive.

3.2.3 Second-Tier Replication

To explore a midpoint between the Standard and DB State approaches, we implemented a replication scheme similar to [15], where the soft state of each session is replicated on two application server nodes. When a session is started on some application server *A*, it chooses a peer server *B* (in round-robin fashion) to hold a backup replica of the new session’s soft state. The requests associated with the session are only sent to *A*, which is responsible for updating the state on *B*. If *A* fails, the fault is detected by the Web servers, which start routing the requests that would be going to *A* to *B*. When *A* comes back up, it reestablishes its TCP connections with the other servers and then can take on new sessions. In addition, if any of the sessions that *A* was handling before it failed are still active, *A* resumes its role as primary server for them. In contrast, if *B* fails, the updates that would be sent to it are simply lost. When *B* comes back up, it reestablishes its connections with the other servers and can take on new sessions. It also resumes its role as backup replica for any sessions that remain from before it failed.

The details of the replication of the soft state are as follows: When *A* fields a request that causes the soft state to change, after all changes have been made but before *A* completes the request and replies to the Web server, *A* serializes its copy of the soft state and sends it to *B* over a persistent TCP connection. Server *A* assumes that *B* has received the soft state update after its TCP write operation completes, i.e., *A* does not wait for an acknowledgment of the update. This replication scheme is lightweight—it only requires the serialization and one message—and allows the service to tolerate many common faults, such as node or application crash. However, note that this scheme does not guarantee strong consistency. In the presence of certain sequences of faults that overlap (or occur close together),

TABLE 1
Summary of the Characteristics of the Different Strategies

Strategy	Location of State	Replication	Availability	Performance
Standard	Application servers	No	Loss of state with application server	Fast and scalable
DB State	Database servers	No	Database servers are most reliable	Performance of database
2nd-tier Repl.	Application servers	Yes	Loss of state is rare	Replica coherence

the state can be lost or revert to an older version. Thus, this scheme favors higher performance in the common case in exchange for weaker semantics in the presence of multiple concurrent faults.

We present a summary of the characteristics of each state maintenance strategy in Table 1.

3.3 Online Bookstore

The specific multitier service that we study is a bookstore modeled after the TPC-W standard benchmark for e-commerce systems [35]. The code for this application is publicly available from the DynaServer project [29] at Rice University. The bookstore implements the functionality in TPC-W that affects performance [1], including transactional support. All persistent data, except for images, are stored in the database. The database contains eight tables, including tables for customers, orders, items, and authors. The shopping cart is also stored in the database. The 14 interactions with the service can be either read-only or cause the database to be updated. The read-only interactions involve accesses to the home page, listing of new books and best sellers, requests for product detail, and searches. The read-write interactions include user registration, updates to the shopping cart, and purchases.

Thus, the soft state of this service is comprised essentially of the contents of shopping carts. Any error that causes the soft state to be lost or become unreachable may force each client to repeat several requests to recreate the state.

The bookstore is organized into three tiers of servers: Web, application, and database tiers. The first and second tiers are comprised of multiple nodes, whereas the third tier involves a single node running the database. (To be realistic, in our experiments, the database node is the fastest and most reliable machine in the cluster.) The service requests are received by the Web servers and may flow toward the second and third tiers. The replies flow through the same path in the reverse direction. Each Web server keeps track of the requests it sends to the application servers. It also tries to balance the load on these servers, according to its own outstanding-request information.

A client emulator (also from Rice) is used to exercise the services [1]. The workload consists of a number of concurrent clients that each repeatedly open sessions with the service being exercised. Each client repeatedly issues a request, receives and parses the reply, “thinks” for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. (For repeatability, we generated a trace from one run using this model and then used the same trace for all subsequent measurement runs.)

We made a few changes to the service to boost its performance. The two most significant changes were as follows: First, we modified the service to implement the state maintenance strategies that do not store soft state in the database. Specifically, we changed the original implementation to use the session management infrastructure provided by the Tomcat application servers to maintain the soft state. These new implementations affect the request forwarding scheme. On the first request of a session, the Web server selects an application server to handle the session in round-robin fashion. Other requests belonging to the session are sent to the same application server.

Second, we changed the node fault (or unreachability) detection mechanism. Originally, the detection was based on TCP timers, which take unnecessarily long to timeout for clusters, so we implemented a heartbeat service on the first and second tiers. Each node broadcasts one heartbeat per second and maintains a local cluster membership list according to the broadcasts it receives. A node A deems another node B to be down (or unreachable) if it fails to receive a heartbeat from B for 10 consecutive seconds. When B becomes available again, it is reintroduced into the service.

3.4 Server Setup and Workloads

Our experiments use a 1.9-GHz Pentium IV-based machine with a 15K rpm SCSI disk and a Gigabit Ethernet network interface as the database server. This server runs the MySQL relational database, version 4.12. The other two tiers are implemented by a set of eight 1.2-GHz Celeron-based PCs, each with 512 MBytes of RAM and a Gigabit Ethernet interface. Two of the eight machines comprise the first tier, each running the Apache server, version 1.3.27. The six remaining machines comprise the second tier, each running the Tomcat servlet server, version 4.1.18. All nodes run Linux.

The provisioning of the first and second tiers was defined so that each tier has the smallest number of nodes that still leaves the database as the performance bottleneck. In our fault-injection experiments, we set the offered load at 90 percent of the maximum throughput of the database server for the service, which translates to a request rate of 560 req/s for Standard and second Tier Replication and 480 req/s for DB State. With these settings, no node besides the database node is more than 80 percent utilized (in the absence of faults). These settings represent the common case of having highly utilized bottleneck resources and plenty of extra capacity elsewhere.

We run the clients on four other PCs connected to the same Gigabit Ethernet switch as the server cluster. Our bookstore’s database is initialized to contain 10,000 items.

TABLE 2
Faults and Their MTTFs and MTTRs per Fault Type

Fault	MTTF	MTTR	Fault	MTTF	MTTR
Node crash	2 weeks	3 minutes	Internal link down	6 months	3 minutes
Node freeze	2 weeks	3 minutes	Switch hang*	1 year	30 seconds
Application crash	2 months	3 minutes	Database crash*	2 years	30 seconds
Application hang	2 months	3 minutes			

*Application hang and crash together represent an MTTF of one month for application results. *Switch and database faults are not injected; they are captured in the modeling during phase 2.*

We exercise the bookstore with a “shopping mix” of requests, where 20 percent of the requests are read-write. We set the “think” time between requests to 0.5 second to reduce the number of client nodes required. A client request times out after 10 seconds without a reply. A request that times out is counted as a failed request and, so, counts toward the unavailability of the service, even if the service is able to eventually complete the request. This is appropriate because a request should not be counted as successful if the user is no longer interested in the request by the time that the service is able to complete it.

3.5 Fault Load

We use Mendosus [19] to inject the fault load shown in Table 2 into live service executions.³ The table lists the MTTF and the MTTR we use for each fault type. The MTTFs for node, link, and switch-related faults are derived from previous works that empirically observed the fault rates of many systems [4], [16], [18], [21], [24]. The MTTFs for application and database-related faults and all MTTRs are based on common sense and our own experience with Internet services, due to the lack of available data on these rates.

These generic faults can have many causes in a real system; for example, an operator accidentally pulling out the wrong network cable corresponds to a link fault. 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 faults as observed by the system, using an MTTF that covers most potential causes of a particular fault.

The above faults are injected into the first and second tiers of nodes. Our heartbeat mechanism and automatic reintegration of repaired components allow the bookstore service to automatically recover to normal performance once any failed component has been repaired, i.e., operator intervention is never required to regain normal functionality for the injected fault load. When a component fails, requests that are in progress and need to use the component may be delayed or lost altogether. The clients consider any request that takes more than 10 seconds to be serviced as a lost request.

Finally, because we do not have a spare switch and a replicated database, we cannot inject faults of these components. Instead, we only model these faults. We assume that switches fail with an MTTF of one year and it

takes 30 seconds to fail-over to a spare switch, as described in [9]. During these 30 seconds, the service is assumed to be completely unavailable. Similarly, we assume that the database server fails with an MTTF of two years and the fail-over to a hot spare takes 30 seconds. During these 30 seconds, the database server is completely unavailable, which in turn causes the entire service to quickly become unavailable as well. We assume that the overhead of keeping a hot spare database is negligible for our workloads, as demonstrated in [2].

3.6 Performance

We first consider service performance in the absence of faults: Fig. 2 plots service throughput for each of the three strategies as a function of request rate. These results confirm our intuition that Standard and Second-Tier Replication should outperform DB State because DB State maintains the soft state in the tier that is already the system bottleneck. Specifically, the maximum throughput of DB State is 19 percent lower than that of Standard. On other hand, there is little if any difference in the performance of Standard and Second-Tier Replication.

3.7 Fault Injection

We now focus on the behavior of the bookstore in the presence of faults. In particular, we present and discuss some representative results from phase 1 of our methodology.

3.7.1 Network Faults

Fig. 3a shows the effects of a transient fault in a link connecting one of the application servers to the rest of the system. This prevents both the front-end servers and the database from reaching the application server, stalling any flow of requests through it. We see that the Standard and

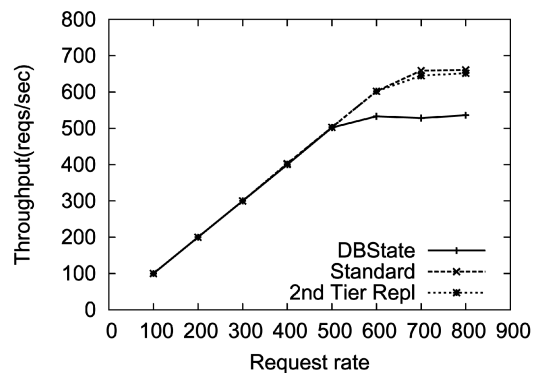


Fig. 2. Throughput versus request rate.

3. We also injected disk faults in the form of IDE timeouts at Web and application servers. But, due to the negligible number of disk accesses made by these servers, there is no measurable impact on the overall availability. Thus, we did not include disk faults in our fault load.

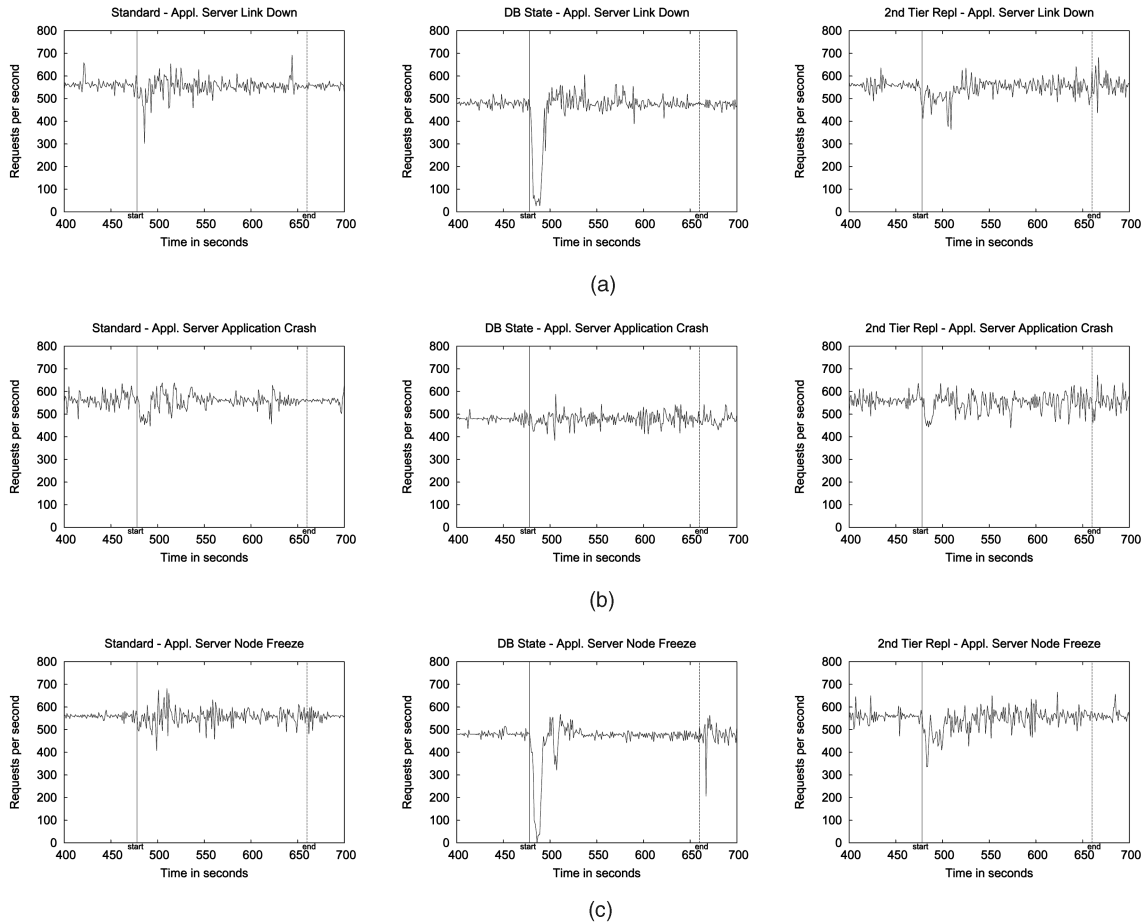


Fig. 3. Measured throughput when a fault is injected at an application server node for (left) Standard, (middle) DB State, and (right) Second-Tier Replication strategies. Vertical lines represent the start (solid) and end (dashed) times of injected faults. (a) Effects of a transient link, (b) effects of a process crash, and (c) effects of a node freeze.

the Second-Tier Replication versions behave as expected. All current sessions associated with the faulty server timeout and are either aborted, as in the case of Standard, or transferred over to the back-up server in the case of Second-Tier Replication. In the case of DB State, however, the outstanding transactions from the application server hold resources at the database, stalling it until the connections timeout and are aborted. This causes the extreme throughput degradation at the beginning of the fault in the case of DB State.

In all three versions, the throughput recovers to the normal value once the fault at the application server is detected by the front-end servers and the requests are redirected to other servers. Note that, since the database is the bottleneck resource in all versions, the loss of a single application server is easily handled by the remaining set of servers in the second tier. Once the link has recovered, the application server is once again included in the membership by the front-end servers and the requests are once again forwarded to it.

3.7.2 Application Faults

Fig. 3b shows the impact of an application crash fault injected at one of the application servers. The fault kills the application server, resulting in the termination of all open connections between the application server and both the

front-end and database servers. Unlike in the link fault described above, the resources held at the database and the front-end servers are recovered almost immediately and subsequent requests are directed away from the faulty application server. The application server recovery involves restarting the Tomcat process, which is then detected by the membership protocol and requests for new sessions are once again directed to the recovered server.

3.7.3 Node Faults

Fig. 3c shows the effects of a node freeze fault at the application tier. The behavior of the three versions is similar to those under the link fault described above. The fault is detected by the front-end nodes and all future requests are directed to other servers in the second tier. The Standard and Second-Tier Replication versions recover quickly to normal throughput. The DB State shows a more severe degradation as the resources held by the faulty application server are not cleanly released until the timeout and subsequent abort of all outstanding connections.

3.7.4 First-Tier Faults

The Web servers were subjected to the same fault load as the application servers. As a sample of these results, Fig. 4 shows the impact of a node freeze injected at one of the Web

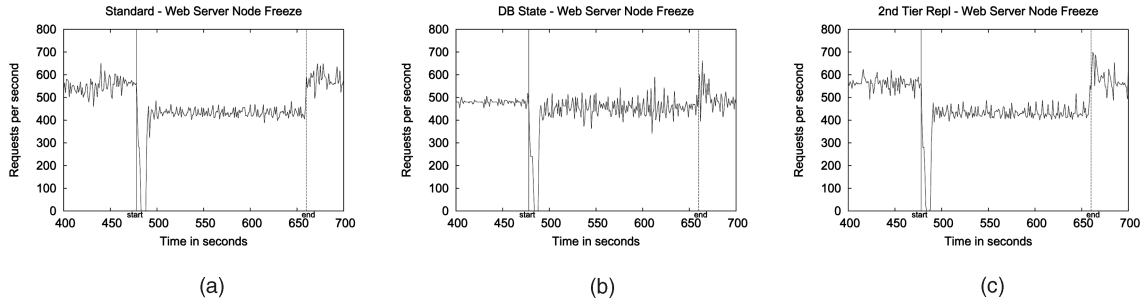


Fig. 4. Measured throughput when a node freeze is injected at a Web server node for (a) Standard, (b) DB State, and (c) Second-Tier Replication strategies. Vertical lines represent the start (solid) and end (dashed) times of injected faults.

server nodes for each of the systems. Note that the impact of these faults on throughput is more pronounced than we see in Fig. 3c. The reason is that the failure of a Web server reduces the capacity of the first tier by 50 percent. The throughput of DB State suffers the least since DB State cannot serve as high a load as the other systems.

3.8 Availability

Fig. 5 provides a flavor for the data we collected for the various stages of our seven-stage template in phase 1 of the methodology. The figure shows the average throughput and duration of each stage during a node freeze at an application server node of DB State (middle graph of Fig. 3c). Recall that, for some types of faults, some stages collapse into a single stage or are not used. These data, along with the MTTF and MTTR values listed in Table 2, are used as inputs for phase 2 to compute availability, as explained in Section 2.

Fig. 6 shows the unavailability of the three strategies. These results show that, overall, availability is somewhat better than 99.9 percent, or 3 nines, with DB State achieving close to 99.99 percent, or 4 nines. As expected, Standard exhibits the highest unavailability. Also, node and application faults are the largest contributors to unavailability since they occur most frequently.

Surprisingly, faults injected into the Web servers accounted for a large fraction of the unavailability, especially

under the Second-Tier Replication strategy. The reason is that there are only two nodes in the first tier so that, when one of them becomes unavailable, the service loses 50 percent of its processing capacity in this tier. In contrast, when a node fails in the second tier, only 17 percent of the processing capacity is lost. Of course, switch and database faults are catastrophic in that 100 percent of the processing capacity is lost; however, these failures have a lower impact on unavailability because they occur much less frequently and their fail-over is fast.

The high unavailability of the Standard version compared to the other two strategies is explained by the requests that need to be reissued to recreate the soft state lost due to faults in Standard. Recall from (1) that we account for reissued requests in the average number of failed sessions (*Sessions_Affected*) and the average number of requests reissued per session (*Recreate_State*). Our experimentally derived *Sessions_Affected* is 1,120 sessions and *Recreate_State* is 7.193 requests for each fault that causes the loss of soft state in an application server in Standard, resulting in a total reissued request count of 8,056.86 per fault. This is a substantial request load; assuming a normal throughput of 650 requests/second, this number is equivalent to about 12 seconds of fault-free operation.

3.9 Performability

Finally, we consider the performability achieved by the three strategies using our performability metric. Fig. 7

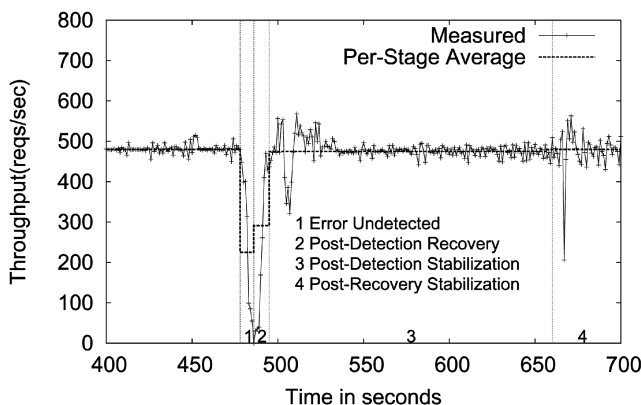


Fig. 5. Measured throughput and the corresponding mapping to our seven-stage template upon a node freeze fault injected at a DB State application server node. Vertical lines indicate the separation of service behavior into the stages of our template.

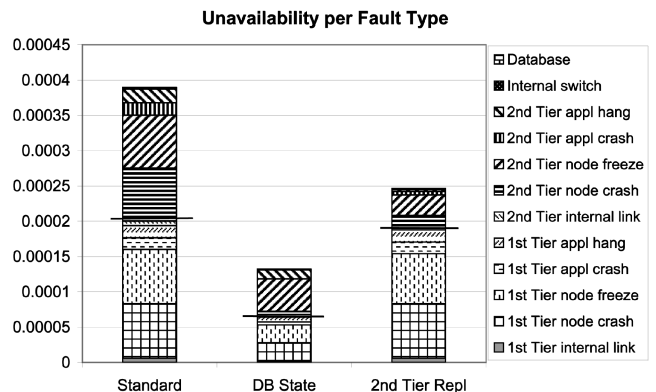


Fig. 6. Unavailability of each strategy. Each bar shows the contribution of each fault type when injected at different tiers. The contributions are stacked in the same order as the legends; the horizontal line across each bar separates the first and second-tier faults.

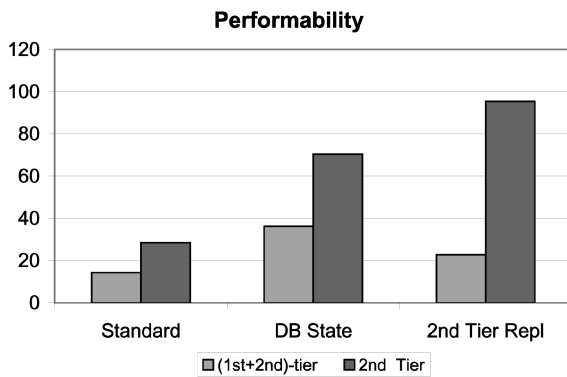


Fig. 7. Performability of each strategy. In each pair of bars, the one on the left quantifies performability when faults can occur on both the first and second tiers, whereas the bar on the right only includes faults on the second tier. Switch and database faults are considered in all cases.

shows service performability for the three strategies, assuming a target unavailability of 0.00001 (5 nines) and unit weights for performance and unavailability. For each strategy, we show the performability when faults are injected into both the first and second tiers, as compared to when faults are injected only into the second tier. Note, however, that internal switch and database faults are considered in all cases.

The figure shows that DB State achieves the best performability when injecting faults into both the first and second tiers. This is an interesting result since, if we were only considering performance, we would be highly motivated to move the maintenance of soft state out of the database (Section 3.6). However, our results show that leaving excess capacity in the first two tiers, particularly in the first tier, is quite important in the presence of faults. Thus, optimizing the system by moving functionality out of the database has a cost in availability, and, therefore, performability because, when serving a higher throughput, the system has less spare capacity to tolerate faults.

It is also interesting to note that, if we only consider faults in the second tier, Second-Tier Replication exhibits higher performability than DB State. This is consistent with intuition since the second-tier replication strategy is specifically designed to tolerate such faults through its replication.

3.10 Discussion

Generalizing our results, we find that service provisioning and load balancing have to consider their associated availability implications. For example, an availability-oblivious load distribution approach may cause lower unavailability and performability. Performance optimizations, such as offloading of bottleneck tiers, need to consider two key issues: the availability properties of the tiers that will receive the load and their ability to handle faults under the higher load. In particular, designers may want an unbalanced system in which they heavily load highly available components and leave more spare capacity for components that are likely to fail more often. Thus, service designers must consider these two opposing forces to achieve the best performability. By considering these forces explicitly, designers will be able to avoid today's common practice of achieving high performance and availability

through substantial overprovisioning and its associated hardware and maintenance costs.

Extrapolation of our results into scaling rules for larger clusters is difficult because our experiments were performed on a small cluster of eight nodes. In particular, scaling the database tier in a systematic manner has been an area of recent active research [2]. We thus leave as an open question how the performability would change in response to scaling the cluster. However, we would still expect our general result to hold, that is, unbalanced tiers may be a desirable property used to increase availability. The reason is two-fold: First, the database tier would still be the performance bottleneck and, second, the resource demand for the other tiers would be lowest for DB State.

4 RELATED WORK

4.1 Availability and Performability

Our work also extends the previous research on the evaluation and modeling of availability and performability. In particular, the quantification methodology we present here is a refinement and extension of the approach we proposed in [27].

Our quantification study differs from other previous studies in that we focus on multitiered, cluster-based services and use a completely different set of metrics to evaluate availability and performability. Perhaps, most similar to our work is [7], which outlines a methodology for benchmarking systems' availability. Other works have proposed robustness [32] and reliability benchmarks [36] that quantify the degradation of system performance under faults.

Our work provides a formal but simple methodology to evaluate service performability in multitier systems. The previous work on performability analysis [11], [22], [33] involved a rich set of stochastic process models that describe system dependencies, fault likelihoods over time, and performance. For example, Smith et al. [33] construct a Markovian reward model representing the evolution of a multiprocessor system through states with different sets of operational components. Each state is associated with a reward, i.e., a performance measure such as throughput or latency. In this context, performability is defined as the distribution of the accumulated reward over time. This distribution allows system designers to explore characteristics of the system, such as the probability of completing a certain amount of work within a period of time. These analyses are usually performed analytically.

Unfortunately, such stochastic modeling approaches are often extremely difficult to apply because they require a detailed understanding of the system and the parameterization of numerous low-level probabilistic state transitions and the rewards associated with individual states. Furthermore, current performability metrics do not adequately capture the very high cost of unavailability to today's services [28].

Compared to these complex stochastic models, our models are much simpler and, thus, more accessible to practitioners. Furthermore, our performability analysis relies on actual fault-injection experiments and introduces simple performability metrics that penalize service designs heavily for their unavailability.

4.2 State Maintenance

Modern Internet services require high performance, scalability, and availability, among other important characteristics. To achieve these characteristics, several research projects and commercial products have moved away from keeping all service state using transactions and databases with ACID properties [12], which typically favor strong consistency over performance. For example, a few systems [10], [14], [20] have proposed in-core data management layers that provide flexible and efficient access to (soft and/or hard) state with weaker consistency semantics. High concurrency and availability are ensured by managing multiple replicas of data. The Porcupine [30] and Neptune [31] systems pursued similar goals.

Industrial Internet service platforms, such as the IBM Websphere [17] and the BEA WebLogic [5], support various state maintenance schemes for both hard and soft state, either through EJB interfaces or other proprietary mechanisms. The soft state is usually replicated using the primary-secondary scheme, and copies are kept coherent using strict two-phase commit. ASP.NET [23] also allows on-demand backing up of state to an independent node called the StateServer. All these platforms also support database persistence and client-side cookies for minimal session state.

Unfortunately, these projects and products never really quantified the availability aspects of their proposed state maintenance strategies. Their focus was always on demonstrating high performance, tolerance to faults, and correct behavior on recoveries. Furthermore, their experiments were always limited to the tier where the state was kept.

Our work extends these previous works by quantifying the performance, availability, and performability of several soft state maintenance strategies that share the same concepts. This quantification shows, for example, that the major source of unavailability may not really be the tier where the state is kept. Moreover, it shows that keeping the soft state in the database actually leads to higher availability when the database has a hot spare [2]. In terms of performance, we indeed observe that storing the soft state in the database produces lower throughput under high load. However, in terms of performability, we demonstrate that the performance degradation of the database scheme is actually outweighed by its high availability when one weights performance and unavailability equally.

5 CONCLUSIONS

The need for appropriate methodologies and infrastructures for the design and evaluation of highly available services 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 due to design changes.

To demonstrate our methodology, we have used it to quantify the performance, availability, and performability of three soft state maintenance strategies in the context of a multitier bookstore service. Our study demonstrated that, surprisingly, most of the unavailability of the service is due to faults in the first tier of nodes. When comparing the

different soft state maintenance strategies, we found that storing the soft state in a database achieves better performability than storing it in main memory (even when the state is efficiently replicated) when performance and availability are equally important.

The key lesson from our experiments is that offloading the bottleneck tier certainly improves performance, but it may hurt availability by a larger factor. Service designers have to evaluate the impact of these offloading decisions on all other tiers. In particular, it is important to guarantee that the other tiers will be capable of tolerating faults efficiently, especially when they are more prone to faults. Unfortunately, the common practice of overprovisioning all tiers achieves high performance and availability, but at high hardware and maintenance costs. Thus, we conclude that service provisioning and load balancing needs to consider availability, cost, and performance explicitly.

ACKNOWLEDGMENTS

This research was partially supported by US National Science Foundation grants #EIA-0103722, #EIA-9986046, and #CCR-0100798, and CNPq/Brazil under grants #680.024/01-8 and #380.134/97-7.

REFERENCES

- [1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and Implementation of Dynamic Web Site Benchmarks," *Proc. Fifth Ann. Workshop Workload Characterization*, Nov. 2002.
- [2] C. Amza, A. Cox, and W. Zwaenepoel, "Conflict-Aware Scheduling for Dynamic Content Applications," *Proc. Fourth USENIX Symp. Internet Technologies and Systems*, Mar. 2003.
- [3] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable Content-Aware Request Distribution in Cluster-Based Network Servers," *Proc. USENIX 2000 Technical Conf.*, June 2000.
- [4] S. Asami, "Reducing the Cost of System Administration of a Disk Storage System Built from Commodity Components," Technical Report CSD-00-1100, Univ. of California, Berkeley, June 2000.
- [5] BEA, BEA WebLogic, <http://www.bea.com/products/weblogic>, Sept. 2003.
- [6] E. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, July/Aug. 2001.
- [7] A. Brown and D.A. Patterson, "Towards Availability Benchmarks: A Case Study of Software RAID Systems," *Proc. 2000 USENIX Ann. Technical Conf.*, June 2000.
- [8] E.V. Carrera and R. Bianchini, "Efficiency vs. Portability in Cluster-Based Network Servers," *Proc. Eighth Symp. Principles and Practice of Parallel Programming*, June 2001.
- [9] Cisco, Failover Configuration for LocalDirector, 2003. http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm.
- [10] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, "Cluster-Based Scalable Network Services," *Proc. 16th ACM Symp. Operating Systems Principles*, Oct. 1997.
- [11] S. Garg, A. Puliafito, M. Telek, and K.S. Trivedi, "Analysis of Preventive Maintenance in Transactions-Based Software Systems," *IEEE Trans. Computers*, vol. 47, no. 1, pp. 96-107, Jan. 1998.
- [12] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [13] S.D. Gribble, E.A. Brewer, J.M. Hellerstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction," *Proc. Fourth USENIX Symp. Operating Systems Design and Implementation*, pp. 319-332, Oct. 2000.
- [14] S.D. Gribble, M. Welsh, R. von Behren, E.A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services," *J. Computer Networks*, vol. 35, no. 4, Mar. 2001.

- [15] F. Hanik, In Memory Session Replication in Tomcat4, <http://www.theserverside.com/articles/article.tss?l=Tomcat>, Apr. 2002.
- [16] T. Heath, R. Martin, and T.D. Nguyen, "Improving Cluster Availability Using Workstation Validation," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, June 2002.
- [17] IBM, IBM WebSphere, <http://www.ibm.com/websphere>, Sept. 2003.
- [18] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers," *Proc. 18th Symp. Reliable and Distributed Systems*, Oct. 1999.
- [19] X. Li, R.P. Martin, K. Nagaraja, T.D. Nguyen, and B. Zhang, "Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services," *Proc. First Workshop Novel Uses of System Area Networks*, Jan. 2002.
- [20] B. Ling and A. Fox, "A Self-Tuning, Self-Protecting, Self-Healing Session State Management Layer," *Proc. Fifth Ann. Workshop Active Middleware Services*, June 2003.
- [21] D.D.E. Long, J.L. Carroll, and C.J. Park, "A Study of the Reliability of Internet Sites," *Proc. 10th Symp. Reliable Distributed Systems*, pp. 177-186, Sept. 1991.
- [22] J.F. Meyer, "Performability Evaluation: Where It Is and What Lies Ahead," *Proc. IEEE Int'l Computer Performance and Dependability Symp.*, pp. 334-343, Apr. 1995.
- [23] Microsoft, ASP.NET, <http://www.asp.net/>, Sept. 2003.
- [24] B. Murphy and B. Levidow, "Windows 2000 Dependability," Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.
- [25] K. Nagaraja, R. Bianchini, R. Martin, and T.D. Nguyen, "Using Fault Model Enforcement to Improve Availability," *Proc. Second Workshop Evaluating and Architecting System Dependability*, Oct. 2002.
- [26] K. Nagaraja, N. Krishnan, R. Bianchini, R. Martin, and T.D. Nguyen, "Evaluating the Impact of Communication Architecture on the Performability of Cluster-Based Services," *Proc. Ninth Symp. High Performance Computer Architecture*, Feb. 2003.
- [27] K. Nagaraja, N. Krishnan, R. Bianchini, R.P. Martin, and T.D. Nguyen, "Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services," *Proc. Fourth USENIX Symp. Internet Technologies and Systems*, Mar. 2003.
- [28] D.A. Patterson et al., "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report CSD-02-1175, Univ. of California, Berkeley, Mar. 2002.
- [29] Rice Univ., DynaServer Project, <http://www.cs.rice.edu/CS/Systems/DynaServer>, 2003.
- [30] Y. Saito, B.N. Bershad, and H.M. Levy, "Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service," *Proc. 17th ACM Symp. Operating Systems Principles*, pp. 1-15, Dec. 1999.
- [31] K. Shen, H. Tang, T. Yang, and L. Chu, "Integrated Resource Management for Cluster-Based Internet Services," *Proc. Fifth USENIX Symp. Operating Systems Design and Implementation*, Dec. 2002.
- [32] D. Siewiorek, J. Hudakund, B. Suh, and Z. Segall, "Development of a Benchmark to Measure System Robustness," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing*, pp. 88-97, June 1993.
- [33] R.M. Smith, K.S. Trivedi, and A.V. Ramesh, "Performability Analysis: Measures, an Algorithm, and a Case Study," *IEEE Trans. Computers*, vol. 37, no. 4, Apr. 1998.
- [34] N. Talagala and D. Patterson, "An Analysis of Error Behaviour in a Large Storage System," *Proc. 1999 Workshop Fault-Tolerant Parallel and Distributed Systems*, Apr. 1999.
- [35] Transaction Processing Performance Council, TPC-W, <http://www.tpc.org/>, 2003.
- [36] T.K. Tsai, R.K. Iyer, and D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems," *Proc. Symp. Fault-Tolerant Computing*, pp. 314-323, June 1996.



Kiran Nagaraja is currently pursuing the PhD degree in the Department of Computer Science at Rutgers University. He received the Bachelor's degree in computer engineering in 1999 from the National Institute of Technology Karnataka, India. His research interests include performance, availability, and fault tolerance aspects of cluster-based servers. He is a student member of the IEEE and a member of the ACM.



Gustavo Gama received the MSc degree in computer science from the Federal University of Minas Gerais (Brazil) in 2003. From 2003 until 2004, he was a research assistant at Rutgers University. He now works as a software engineer at Vetta Technologies.



Ricardo Bianchini received the PhD degree in computer science from the University of Rochester in 1995. From 1995 until 1999, he was an assistant professor at the Federal University of Rio de Janeiro, Brazil. He is now an associate professor in the Department of Computer Science at Rutgers University. His current research interests include the performance, availability, and energy consumption of cluster-based network servers. He has received several awards, including the US National Science Foundation CAREER award. He is a member of the IEEE, the IEEE Computer Society, and the ACM.



Richard P. Martin received the PhD degree in computer science from the University of California at Berkeley in 1999. He is currently an assistant professor in the Department of Computer Science at Rutgers University. His current research interests include availability of large scale computer systems and sensor networks. He is a member of the IEEE and the ACM.



Wagner Meira Jr. received the PhD degree in computer science from the University of Rochester. He is an associate professor in the Department of Computer Science at the Federal University of Minas Gerais, Brazil. His current interests are in performance analysis and modeling of parallel and distributed systems and data mining algorithms, as well as their parallelization.



Thu D. Nguyen received the PhD degree in computer science from the University of Washington in 1999. From 1986 to 1991, he was a member of the technical staff at AT&T Bell Laboratories. He received the MS degree in computer science from the Massachusetts Institute of Technology in 1988 under the sponsorship of AT&T. Since 1999, he has been an assistant professor in the Department of Computer Science at Rutgers University. His current research interests include the performance, availability, and security of distributed systems, ranging from cluster-based Internet services to peer-to-peer systems. He is a member of the IEEE Computer Society.