State Maintenance and its Impact on the Performability of Multi-tiered Internet Services *

G. Gama[†‡], K. Nagaraja[†], R. Bianchini[†], R. Martin[†], W. Meira Jr.[‡], and T. Nguyen[†]

[†]Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019
{knagaraj,ricardob,rmartin,tdnguyen}@cs.rutgers.edu

[‡]Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
{gmcgama,meira}@dcc.ufmg.br

Abstract

In this paper, we evaluate the performance, availability, and combined performability of four soft state maintenance strategies in two multi-tier Internet services, an on-line book store and an auction service. To take soft state and service latency into account, we propose an extension of our previous quantification methodology, and novel availability and performability metrics. 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. Overall, we find that the services achieve between 99.9% and 99.99% availability. 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. Based on our results, we conclude that service designers need to provision the cluster and balance the load with availability and cost, as well as performance, in mind.

1 Introduction

Popular Internet services frequently rely on clusters of commodity computers as their supporting infrastructure [6]. These services must exhibit several characteristics, including high performance, scalability, and availability. The performance and scalability of cluster-based servers have been studied extensively in the literature, e.g. [3, 6, 8]. In contrast, understanding designs for availability, behavior during component faults, and the relationship between performance and availability of these servers have received much less attention. In particular, no previous work that we are aware of has quantified the relationship between state maintenance strategies and the performance and availability of complex, multi-tier services. In fact, the studies that addressed state maintenance in Internet services (e.g., [13, 19]) focused solely on demonstrating performance effects resulting from a node crash and subsequent recovery.

Given the limitations of the previous work, in this paper we quantify the performance, availability, and combined *performability* of four 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. Three of the maintenance strategies are *stateful* in that the servers maintain the soft state of all clients. In the other strategy, the soft state associated with each client is kept encrypted at the client machine and is transferred between the client and the server cluster along with each request (and, possibly, reply). For this reason, we refer to this strategy as *soft-stateless* or simply *stateless*.

We study these strategies in the context of two multi-tier services, an on-line book store and an auction service. These services are organized into three tiers of servers: Web, application, and database servers. Their soft state is comprised mostly by the contents of shopping carts (book store) and the auctions of interest to the clients (auction). Any error that causes the soft state to be lost or become unreachable typically forces each client to repeat several requests to re-create the state. The hard/persistent state of these applications, such as the number of stocked copies of a book or the highest bid for an item, is always maintained at a database server, regardless of the soft state maintenance approach.

To quantify the performance, availability, and performability of these services, we use an extension of the 2-phased quantification methodology that we proposed in [24]. In the first phase of our methodology, the service performance is benchmarked in the presence and absence of (injected) faults. In the second phase, an analytical model is used to combine an expected fault load, measurements from the first phase, and parameters of the surrounding environment to predict availability. Finally, service performance and availability are combined into two performability metrics, one based on service throughput and the other on service latency.

Our performance results show that keeping the soft state in a database causes significant performance degradation compared to other stateful strategies and the stateless system for high server loads. For the auction service, the throughput degradation is highest at 27%. In terms of availability, all strategies achieve only 99.9% ("3 nines") or 99.99% ("4 nines") for both services, depending on whether throughput or latency is the metric of interest. In fact, 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 in terms of both throughput and latency. Overall, storing the soft state in a database compares favorably against storing it in main memory, even when the state is

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

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 for both services. 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 experiments is that offloading the bottleneck tier improves performance but may also hurt availability, if any other tier loses its ability to tolerate faults efficiently. The common practice of over-provisioning all tiers achieves high performance and availability but at high hardware and maintenance costs. Based on this lesson, we conclude that service designers need to provision and balance the load across the cluster tiers with availability, cost, as well as performance, in mind. Thus, in future work, we plan to develop a framework for *availability-aware provisioning and load balancing* of server clusters.

In summary, we make the following contributions:

- *We extend our quantification methodology and propose new availability and performability metrics.* In its first incarnation, our methodology assumed that service requests were independent, i.e. the loss of a request had no impact on future requests seen by the server. Also, our availability and performability metrics only reflected performance and availability from the viewpoint of service throughput. Here, we extend our availability model to consider client sessions, where sets of requests are inter-related and so the failure of a request impacts the future requests coming from the same client. Furthermore, we propose additional availability and performability metrics to reflect service behavior from a latency viewpoint. Finally, and perhaps most importantly, we show that our extended methodology is appropriate for evaluating today's complex multi-tiered services.
- *We compare the behavior of four soft state maintenance approaches in two three-tier services.* Previous state maintenance studies focused on services that operate correctly in response to a stateful node failure and recovery. However, the spectrum of possible failures is much broader, including service application crashes and hangs in multiple tiers, for example; we assess the effect of several other failures on the behavior of complex services. Furthermore, our quantification of availability and performability allows for more precise comparisons between different service designs than was previously possible.

The remainder of the paper is organized as follows. The next section presents some background information on multi-tier services, current approaches for maintaining soft state, and performability analysis. Section 3 describes the state maintenance strategies we study. Section 4 describes our quantification methodology. Section 5 describes our experiments, including the two sample services we study, our server infrastructure, and fault loads. Section 6 presents our performance, availability, and performability results. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 Background

Multi-tier 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 than these first-generation services. They are now organized in multiple tiers of clustered servers that cooperate to serve an increasing amount of dynamic content. In fact, the processing of each dynamic request may now have significant computing and data access requirements that can vary greatly depending on the particular type of request. These multi-tier services now support a multitude of e-commerce applications, ranging from simple on-line stores to stock trading to business-to-business commerce services.

Perhaps the most common multi-tier 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 server(s) stores the core content of the service and provides 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 server(s) 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.

State and State Maintenance in Services. 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 on-line book store, the email messages received by a user of an email 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 email 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 auto-

matically or with user help, and so does not need to be persistent. Modern Internet services deal with a variety of soft state, such as thread state (e.g., stack), user profiles, navigation tracking information, the contents of multi-page 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. More specifically, Internet services employ some combination of five basic soft state maintenance techniques [25]: client-side cookies, parameters in URLs, hidden fields of HTML forms, "session objects," or custom database solutions. Cookies are usually used to store session identification information and other small pieces of client-related soft state. URL parameters can only store a small amount of state since Web browsers limit the length of the URLs. Using the hidden fields of HTML forms requires browsing to be done solely through POST requests and complicates application development since the forms must contain hidden fields for all possible data. Most modern application server technologies (e.g., ASP, Servlets/JSP) also allow soft state to be stored in session objects, i.e., in-memory data structures specifically designed to store session soft state. When the service implements session objects, all dynamic requests belonging to the same session need to be processed by the same application server, thus restricting load balancing. Furthermore, session objects can threaten scalability due to memory constraints. The last option is to employ a customized state engine using the database for storage. However, this approach may degrade performance and scalability since the database can often be the bottleneck that is difficult to scale.

Performability Analysis. Most of the previous work on performability, which combines performance and availability, centers around models which capture expected performance in the presence of faults in a stochastic framework. For example, Smith *et al.* [31] 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 (1 - availability) to today's services [26]. In particular, two different systems can exhibit similar expected performance in the presence of faults despite having substantially different unavailabilities. The root cause of this effect is that similar availabilities hide significant differences in unavailability. For example, 99% and 99.9% availabilities differ by only 1%, whereas the corresponding unavailabilities (1% and 0.1%, respectively) differ by an order of magnitude.

The performability analysis we proposed in [24] and extend here departs from these traditional approaches in that it relies on actual fault-injection experiments and simple linear availability models, and introduces simple performability metrics that penalize designs heavily for their unavailability.

3 State Maintenance Strategies

We now describe and qualitatively compare the four state maintenance strategies, which we call Standard, DB State, 2nd-tier Replication, and Stateless. Standard and DB State are the commonly deployed approaches in today's services, taking the opposite approaches of storing soft state in the application servers vs. storing it in a database. The 2nd-tier Replication strategy takes an intermediate position: the soft state is stored in the application servers for performance but is replicated to increase availability. Finally, in Stateless each client's soft state is stored entirely at the client, off-loading the responsibility for state availability to the clients.

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. With respect to scalability, it is easy to scale the number of application servers to meet specific performance goals. With respect to implementation, Standard is easily implementable using today's dynamic-content technologies such as PHP or Servlets/JSP. These technologies provide the necessary infrastructure for handling sessions and allow service-specific state to be attached to sessions. Furthermore, it is often easier to evolve the soft state data structure and code in this strategy than it is to change the data model when the soft state is stored in a database.

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 re-create their soft state by re-submitting 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.

DB State. DB State uses a conservative approach to storing soft state, treating such state as regular database records that are maintained with full ACID properties.

The advantages of this approach are two-fold. First, it is not always easy to partition the state into soft and hard state. Conservatively treating all state as hard state can reduce the service design and implementation time. Second, this approach 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, on the other hand, include the potential loss of performance and scalability, as well as flexibility for evolving the service. 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 applica-

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
Stateless	Clients	No	Loss of state with client	Long msgs, en/decryption, flexible load balancing

Table 1: Summary of the characteristics of the different strategies.

tion servers to the database may make the database the performance bottleneck; this is undesirable because scaling the database is often difficult and human-intensive. Finally, any modification to the soft state structure now involves modifying the database data model, which can also be a difficult task.

2nd-Tier Replication. To explore a mid-point between the Standard and DB State approaches, we implemented a replication scheme similar to [14], where the soft state of each session is replicated on two application servers. 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 failure 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 re-establishes 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 its failure 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 re-establishes 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 its failure.

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 acknowledgement of the update. This replication scheme is light weight—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), 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.

Stateless. Finally, we evaluate a strategy where each client stores its soft state. Since the soft state is stored outside the server, we call this strategy *soft-stateless* or simply *stateless*.

Each client's soft state is stored in the form of one or more cookies, where each cookie can hold up to 4KB of data. All cookies are encrypted to ensure that only the server can modify the soft state. (Note that the two services studied here, as well as many other services, may not care whether the client can see/modify this state. We are taking the conservative stance of always encrypting to evaluate this approach under the most stringent requirement.) These cookies are transferred in the cookie part of the header of each HTTP request and reply. When the server receives a request, it decrypts the attached cookies and uses them to service the request. If the server modifies the state, then it encrypts the new state and send back the new cookies with the reply. Otherwise, the cookies do not have to be sent back and the state is simply discarded.

Current technology imposes a few constraints on this approach: (1) the encrypted state has to be coded to not contain characters that are illegal in the HTTP header; (2) browsers have to be configured to accept cookies; and (3) the size of the encrypted and encoded state is limited to 80 KBytes (20 cookies of 4 KBytes each). This size should be large enough for most practical purposes. For example, one cookie is enough to store a user's soft state in the two services we study in this paper.

Encryption, encoding, and their inverse operations can either be performed by the Web servers or by the application servers. In fact, the best approach is probably a hybrid one, in which Web servers take the responsibility for these operations when they are less utilized than application servers and vice-versa. For simplicity, our implementations always assign these operations to the Web servers.

The Stateless strategy has the same advantage of the Standard and 2nd-tier Replication approaches over DB State: operations involving the soft state do not involve the database server. In addition, it has two advantages over Standard and 2nd-tier Replication: (1) it provides better memory scalability since stateless application servers do not have to store soft state for all active clients; and (2) it provides more flexible load balancing because a dynamic request can be forwarded to any stateless application server, rather than the application server that holds the state for the corresponding client.

The most important disadvantage of Stateless is that it does not allow any soft state that spans multiple clients to be stored at the clients. Other disadvantages include potentially higher response time and loss of availability when clients move between multiple machines. Response time may increase due to the overhead of larger messages and, if necessary, en/decryption and en/decoding. Loss of availability occurs if a client starts the session on one computer and then moves to another (possibly due to a failure of the initial machine); the state has to be explicitly re-created or copied from one machine to the other. This might suggest that, from the point of view of an individual user, service availability would be lower with the stateless architecture. However, note that in stateful architectures the soft state is usually discarded after a period of user inactivity (e.g., 30 minutes for session state). This means that the user changing machines would also require state to be re-created, unless the change were done "quickly enough".

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

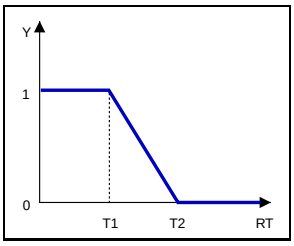


Figure 1: Yield function applied to requests' response times; for this paper, we used T1=1 and T2=10.

4 Quantifying Performance, Availability, and Performability

In this section we describe our extended quantification methodology. First, we describe the baseline performance metrics we use. Next, we describe our availability metrics. Finally, we describe our performability metrics.

4.1 Performance Metrics

From an Internet service perspective, characterizing the system in terms of both throughput and response time (latency) is necessary. From a service provider's perspective, throughput (T) is important as it describes the amount of work that can be performed per unit of time. We define T_n to be the average throughput in the absence of faults. However, from the client's perspective, the service latency is a more relevant metric than throughput.

Using latency as a performance metric has two undesirable properties, however. First, its importance is not a linear function over the range of plausible latencies. Typically, continuing to lower latency beyond the point of human perception is not a useful improvement. At the other end of the spectrum, returning results past the point of human patience is not valuable either. Second, improving latency is a decreasing function, which can be viewed as counter-intuitive.

To address these concerns, instead of latency we use a related "yield" (or "utility") function. We define a yield function, Y, to be applied to each request, such that a reply that is produced quickly is considered highly useful and a reply that is produced after too long is useless. We assume the yield function described in Figure 1, which is similar to that of previous work [29], where a response time of less than 1 second has Y of 1, a response time of more than 10 seconds has a Y of 0, and a simple linear function describes Y when the response time is between 1 and 10 seconds. We define Y_n to be the average yield in the absence of faults.

4.2 Availability Metrics

Our approach for quantifying service availability is divided into two phases. The first phase assesses the service's response to one fault at a time. The second phase combines these responses as a function of the expected arrivals of each fault type, i.e. the fault load. Below, we discuss each of these phases in turn.

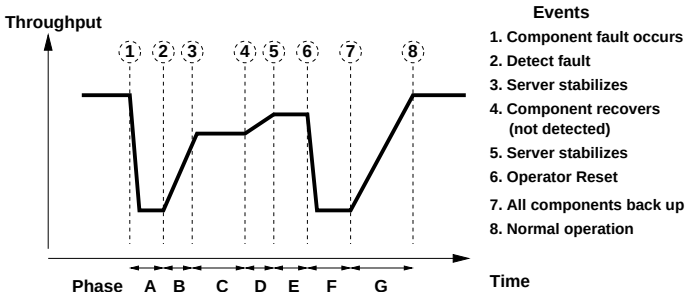


Figure 2: The 7-stage piece-wise linear template specified by our methodology.

4.2.1 Phase 1: Characterizing Service Behavior Under Single-Fault Fault Loads

In this phase, the evaluator first defines the set of all possible faults that can occur while the service is running. Then, the evaluator describes the system's response to a single fault of each type by injecting the fault, measuring the system's behavior during the fault, and fitting the results to a general 7-stage piece-wise linear template.

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

For each stage, we need: (i) the average throughput or yield delivered during that stage, and (ii) the length of time that the system will remain in that stage. The first is always a measured quantity. In the throughput case, care should be taken to disregard requests that are satisfied but take longer than the maximum response time the user would be willing to wait (10 seconds). 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 stage B is typically measured; the time for stages D + E is typically a supplied parameter. 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 7-stage template for each fault.

With respect to the fault injection, the service must have completely recovered from a fault (or have been restarted) before the next one is injected. Further, each fault should last long enough to actually trigger an error and cause the service to exhibit all stages in the 7-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 B, D, and G 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.

4.2.2 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, MTTR) of each component. To simplify the analysis, we assume that faults of different components are not correlated, faults trigger the corresponding errors immediately, and faults "queue at the system" so that only a single fault is in effect at any point in time. These assumptions allow us to add together the various fractions of time spent in degraded modes given a 7-stage template for each fault type gathered in phase 1.

If P_n is the service's performance under normal operation, (i.e., T_n or Y_n), c the faulty component, $MTTF_c$ the mean time to failure of component c, P_c^s the average performance of each stage s in Figure 2 when this fault occurs, D_c^s is the duration of each stage, F_c the average number of user sessions affected by the fault of c, and R_c the average number of requests required to re-create the soft state of each session, our model leads to the following equations for average performance in the presence of faults (P_d) and average delivered availability (A_d).

For throughput performance:

$$P_d = (1 - \sum_c W_c)P_n + \sum_c (\sum_{s=A}^{G} (\frac{D_c^s}{MTTF_c} P_c^s) - \frac{F_c R_c}{MTTF_c})$$

For yield performance:

$$P_d = (1 - \sum_c W_c)P_n + \sum_c (\sum_{s=A}^{G} (\frac{D_c^s}{MTTF_c} P_c^s))$$

$$A_d = \frac{P_d}{P_n}$$

where $W_c = (\sum_{s=A}^{G} D_c^s)/MTTF_c$. Intuitively, W_c is the expected fraction of the time during which the system operates in the presence of fault c. (The denominator of W_c is just $MTTF_c$ instead of $MTTF_c + MTTR_c$ because, when a component fails, another fault could arrive and "queue" at the component. The impact of this assumption is that we compute the fraction of downtime as $MTTR/MTTF$, not as the more typical $MTTR/(MTTF + MTTR)$. In practice, because $MTTF >> MTTR$, the numerical impact of this difference in assumptions is minimal.) Thus, the $(1 - \sum_c W_c)P_n$ factor above computes the expected performance when the system is free of any fault, whereas the $\sum_{s=A}^{G} (P_c^s D_c^s / MTTF_c)$ factor computes the expected performance when the system is operating with a single fault of type c. The $F_c R_c / MTTF_c$ factor represents the sets of requests that will need to be re-issued as a result of this type of fault. The average delivered availability, P_d/P_n, computes the fraction of the fault-free performance that is delivered in the presence of faults. The average delivered unavailability, U_d, can then be computed as $1 - A_d$.

Using this model, we can quantify availability and unavailability assuming either throughput or yield as the performance metric. In fact, these computations typically lead to different absolute results. Intuitively, the throughput-based unavailability is the percentage of requests that are not serviced successfully as a result of faults, whereas the yield-based unavailability is the percentage deviation in yield due to faults. The key is that the service designer or evaluator can select the metric to consider first depending on the aspect of the system that he/she wants to emphasize more strongly.

4.2.3 Correlated Faults

Our availability quantification methodology as presented thus far assumes that faults do not overlap. We believe that the likelihood of overlapping faults for realistic fault rates is negligible, unless faults are correlated.

The methodology can easily be extended to consider correlated faults. 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 7-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.

4.3 Performability Metrics

Having presented our approach for computing performance and availability, we now present a performability metric that combines these measures into a single number. The performability number can be used as a scoring function to rank different service designs.

Again, the problem with standard performability metrics or P_d is that they may hide substantial differences in unavailability. Thus, we define performability as the baseline performance (i.e., the service performance in the absence of faults) scaled by an unavailability penalty factor between 0 and 1.

$$Performability = P_n \times Penalty$$

The penalty factor is adjustable, since different systems may place different importance on unavailability. Specifically, we adjust the delivered unavailability U_d that we computed above by incorporating an *ideal unavailability* factor, such that there is no penalty if the system meets this target:

$$Penalty = \frac{Ideal\ Unavailability}{Delivered\ Unavailability} = \frac{U_i}{U_d}$$

The ideal unavailability is selected by the service designer or evaluator and should be different than 0. A reasonable choice of ideal throughput-based unavailability may be 0.00001 (0.001%), which means that the ideal throughput-based *availability* is 5 nines. A reasonable choice of ideal yield-based unavailability may be 0.01 (1%), which means that we are willing to accept an increase in latency of up to roughly 10%.

Further, the delivered unavailability should not be lower than the ideal unavailability. For yield, the combination of these two requirements can be a problem. The reason is that response time degradation due to faults may not generate a change in yield. For example, it may happen that all response times of a service are very short under normal operation, leading to an yield of 1 as in Figure 1. Under the fault load of interest, these times may be degraded but remain classified as having an yield of 1, simply because they remain less than T1 seconds. This would mean that the system has 0 yield-based unavailability, which is unlikely but not impossible. To eliminate this corner case, we can simply assume the penalty factor to be 1 when the yield-based unavailability is 0.

Overall, we argue that our metrics are intuitive and simple measures for performability because they scale linearly to both performance (be it throughput or yield) and unavailability. Two systems are equal if and only if they trade off equal percentages of performance and unavailability.

5 Methodology

5.1 Services

We study two multi-tier services, an on-line bookstore and an auction service. The bookstore is modeled after the TPC-W standard benchmark for e-commerce systems [32], whereas the auction is modeled after EBay. The code for both services is publicly available from the DynaServer project [27] at Rice University. In the next few paragraphs we describe these services and the modifications we made to improve them in more detail.

Original Services. The bookstore implements the functionality in TPC-W that affects performance [1], including transactional support. All persistent data, except for images, is 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.

The auction service implements selling, browsing, and bidding [1]. There are also three kinds of user sessions: visitor, buyer, and seller. The database stores seven tables, including tables for users, items, bids, and categories. As an optimization, the items table is split into new and old items. The vast majority of requests access new items. We created an extra table to store the list of auctioned items a user tracks during his/her navigation. There can be 26 interactions with the service, the most important of which involve browsing items, bidding, and buying or selling items.

Thus, the soft state of these services is comprised essentially by the contents of shopping carts (bookstore) and the auctions of interest to users (auction). Any error that causes the soft state to be lost or become unreachable may force each client to repeat several requests to re-create the state.

The services are organized into three tiers of servers: Web, application, and database tiers. For each tier, they use the Apache Web server (version 1.3.27), the Tomcat servlet server (version 4.1.18), and the MySQL relational database (version 4.12), respectively. 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 towards 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.

Modifications. We made a few modifications to the services, in order to boost their performance and adjust their design to more standardized practices.

First, we used a different table type for our databases. The original implementations used a performance-optimized table type (MyISAM) that does not provide full ACID properties and relies on single-query atomic operations to guarantee correctness. We switched to the InnoDB table type, which provides all these properties, at the cost of lower performance. To counter the potential degradation in performance, we profiled the queries of both services and tuned the database accordingly. By adding extra indexes for specific queries, inverted lists to handle pattern matching, and caching frequently accessed data, we were able to improve the performance of the database by about 75% compared to the original implementations, even though we now use the ACID-compliant table type.

Second, we modified the services to implement the state maintenance strategies that do not store soft state in the database. For the stateless strategy, we implemented en/decryption and en/decoding at the Web server tier using the DES and Base64 algorithms, respectively. For Standard and 2nd-tier replication, we changed the original implementations to use the session management infrastructure provided by the Tomcat application servers to maintain the services' soft states. (The hard/persistent state of the services is always maintained in the database server, regardless of the soft state maintenance approach.) These new implementations affect the request forwarding scheme in the services. On the first request of a session, the Web server selects the application server to handle the session in round-robin fashion. Other requests belonging to the same session are sent to the same application server. In the other strategies,

the Web server always selects an application server in round-robin fashion.

Third, we changed the node failure (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. Nodes broadcast one heartbeat per second. 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 re-introduced into the service.

Finally, we changed the client emulator in two major ways: (1) we force the emulator to generate a constant workload, even when the service takes long to respond to requests; and (2) to make experiments more repeatable, we now drive the emulator with traces. The requests in the traces are produced by the old version of the emulator. A few other modifications were made to implement timeouts and "cookie" support for the stateless strategy.

5.2 Server Setup and Workloads

Our experiments with the services 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. The other two tiers are implemented by a set of 1.2-GHz Celeron-based PCs, each with 512 MBytes of RAM and a Gigabit Ethernet interface. All nodes run Linux.

We run the clients on 4 other PCs connected to the same Gigabit Ethernet switch as the server cluster. In our experiments with the bookstore, the database contains 10000 items. We exercise the bookstore with a "shopping mix" of requests, where 20% of the requests are read-write. The auction service involves 65784 new items and 1M old items with an average 10 bids per item. For the auction, we use a "bidding mix" of requests, which contains 15% read-write requests. 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.

For the bookstore, we use 8 PC nodes: 2 nodes to run Web servers and 6 nodes to run application servers. For the auction, we use 7 PC nodes: 2 Web-server nodes and 5 application-server nodes. 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 to each service at 90% of the maximum throughput of the database server for the service. With these settings, no node besides the database node is more than 80% utilized (in the absence of faults) for the workloads we consider. Thus, *these settings represent the common case of having highly utilized bottleneck resources and plenty of extra capacity elsewhere.*

5.3 Fault Loads

We use Mendosus [18] to inject the fault load shown in Table 2 into live service executions. The table lists the Mean Time to Failure (MTTF) and the Mean Time to Repair (MTTR) of each component. These rates are derived from previous works that empirically observed the fault rates of many systems [4, 15, 17, 20, 23]. These generic faults can be caused by a wide variety of reasons for a real system; for example, an operator accidentally pulling out the wrong network cable would lead to a link 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 all potential causes of a particular fault. The services can recover automatically to normal performance after the faults are repaired, i.e. operator intervention in never required.

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

Table 2: Faults and their MTTFs and MTTRs per component. Application hang and crash together represent an MTTF of 1 month for application faults. *Switch and database faults are not injected; they are captured in the modeling during phase 2.

These faults are injected into the first and second tiers of nodes. Because we do not have a spare network switch and a replicated database, we cannot inject faults of these components. Instead, we only model these faults. Specifically, we assume that switches fail with an MTTF of 1 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 2 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 performance overhead of keeping a hot spare database is negligible for our workloads, as demonstrated in [2].

6 Experimental Results

In this section, we first compare the performance of the four state maintenance strategies. Then, we compare their unavailability with respect to our injected fault load. Finally, we compare their combined performability.

6.1 Performance

Figures 3 and 4 plot service throughput and yield, respectively, for each of the strategies, as a function of request rate. These results show that there is relatively little difference in performance between the strategies, except that DB State saturates much earlier in both services. The maximum throughputs of DB State are 27% and 18% lower than those of Standard for the auction and bookstore services, respectively. In addition, DB State achieves yields that are up to 30% lower than those of Standard (auction). Clearly, DB State is outperformed by the other strategies because it maintains the soft state in the tier that is already the system bottleneck. The reason the differences in performance are smaller for the bookstore than the auction is that the percentage of accesses to the soft

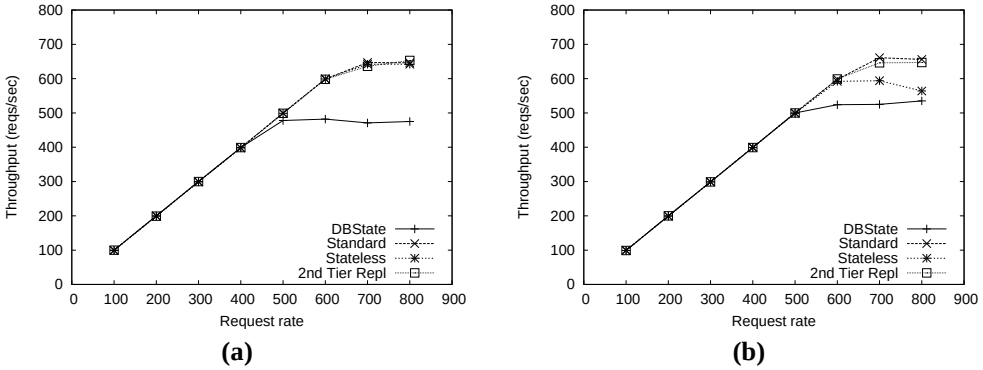


Figure 3: Throughput vs. request rate for the (a) auction and (b) bookstore.

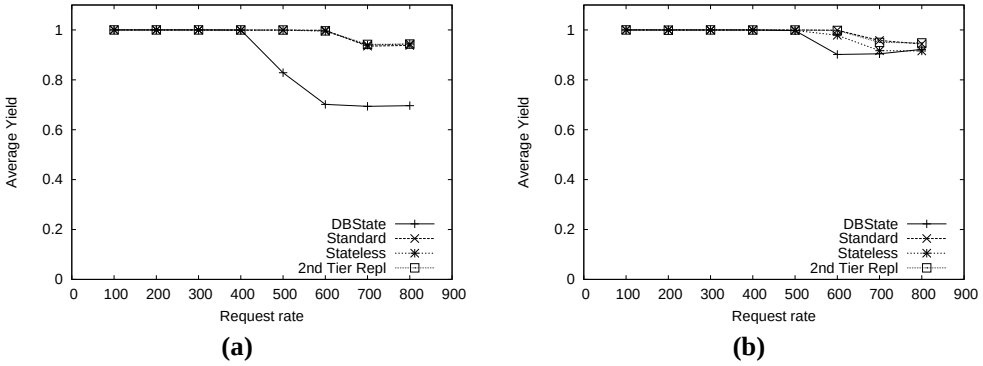


Figure 4: Average yield vs. request rate for the (a) auction and (b) bookstore.

state is also smaller; in the bookstore only 27% of the requests access the soft state, whereas almost 100% of the request access the soft state in the auction.

6.2 Availability

Figures 5 and 6 show the unavailability of the four strategies with respect to throughput and yield, respectively. Each bar includes the contributions of the different fault types injected at different tiers. The unavailabilities associated with each fault type are stacked in the same order as the legends.

These figures show that, for both throughput and yield, overall availability is somewhat better than 99.9%, or three nines, with DB State achieving close to 99.99%, or four nines. Either Stateless (auction) or Standard (bookstore) exhibit the highest unavailability in terms of both throughput and yield.

These results also demonstrate that node and application faults are the largest contributors to unavailability, as one would expect, since these faults occur most frequently. Interestingly, in all strategies other than DB State, faults injected into the Web servers accounted for more than 50% of the unavailability. The reason is that there are only 2 nodes in the first tier so that when one of them becomes unavailable, the service loses 50% of its processing capacity in this tier. In contrast, when a node fails in the second tier, only 17-20% of the processing capacity is lost. (Of course, switch and database failures are catastrophic faults in that 100% 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.)

Finally, note that node and application faults (especially node crash and freeze) in both tiers degrade the availability of Stateless consistently. The reason is that Stateless heavily utilizes both tiers, with en/decryption and en/decoding on the first tier and state object serialization and update on the second tier.

6.3 Performability

Figures 7 and 8 show service performability for the four strategies with respect to throughput and yield, respectively. 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.

These figures show that DB State achieves the best performability for both services when injecting faults into both first and second tiers. This is a very 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 as shown in Section 6.1. However, our results show that leaving excess capacity in the first two tiers, particularly in the first tier, where there may be

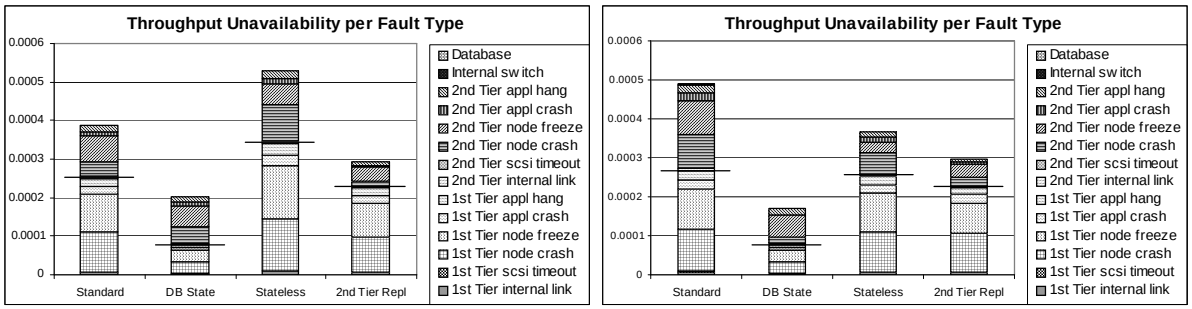


Figure 5: Throughput-based unavailability for the (a) auction and (b) bookstore. Each bar shows the contribution of each fault type when injected at different tiers. The contributions are stacked in the same order as the legends; the horizontal line across each bar separates the 1st and 2nd-tier faults.

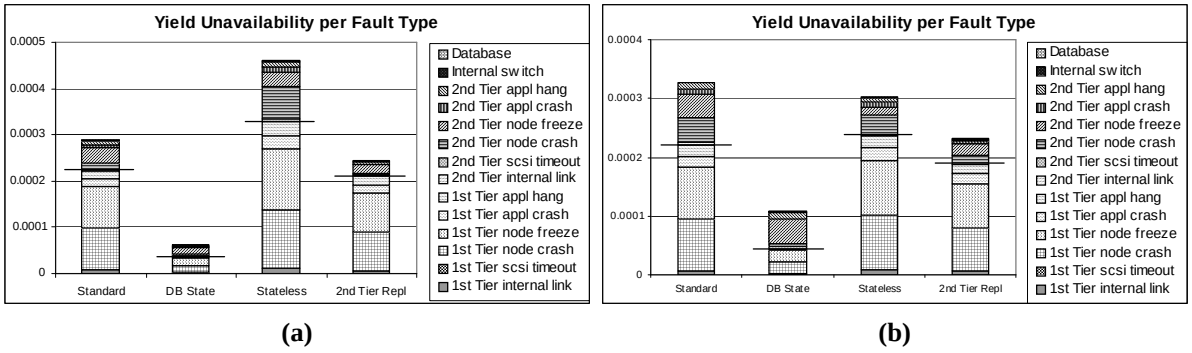


Figure 6: Yield-based unavailability for the (a) auction and (b) bookstore. Each bar shows the contribution of each fault type when injected at different tiers. The contributions are stacked in the same order as the legends; the horizontal line across each bar separates the 1st and 2nd-tier faults.

the least amount of redundancy to tolerate faults, 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, 2nd-Tier Replication exhibits higher performability than DB State in all but one case, yield-based performability for the auction. (The reason for this exception is that the fault-free yield of DB State is already lower at the load we offer to this strategy.) This is consistent with intuition, since the 2nd-tier replication strategy is specifically designed to tolerate such faults through its replication.

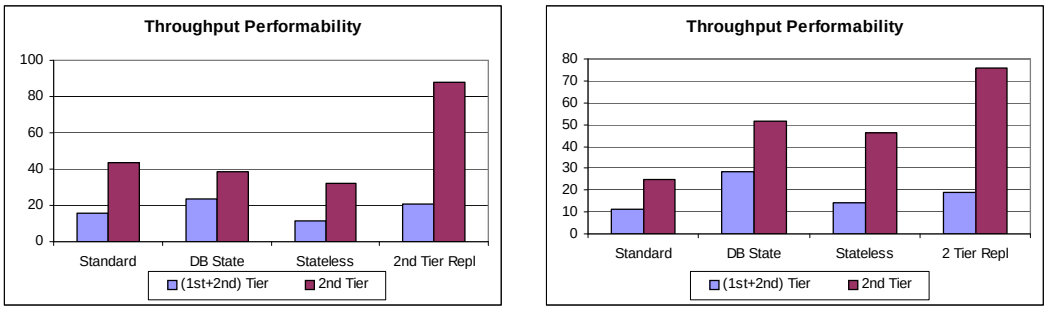
6.4 Discussion

Our experiments were performed on small clusters of 8 (bookstore) and 7 (auction) nodes. Scaling the systems to handle heavier loads would require database replication as in [2]. Under the common provisioning scheme we assume (section 5.2), we expect to see similar trends for large clusters with replicated databases as those reported here. The reason is two-fold: the database tier would still be the performance bottleneck and the resource demand for the other tiers would be lowest for DB State.

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 over-provisioning and its associated hardware and maintenance costs.

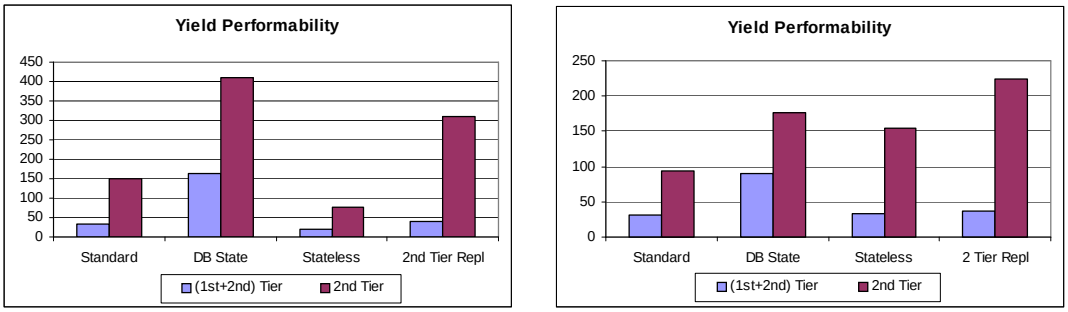
7 Related Work

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 ser-



(a) **(b)**

Figure 7: Throughput-based performability for the (a) auction and (b) bookstore. 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.



(a) **(b)**

Figure 8: Yield-based performability for the (a) auction and (b) bookstore. In each pair of bars, the one on the left quantifyies 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.

vice state using transactions and databases with ACID properties [12], which typically favor strong consistency over performance and availability. For example, a few systems [10, 13, 19] have proposed in-core data management layers that provide a 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 [28] and Neptune [29] systems pursued similar goals.

Industrial Internet service platforms, such as the IBM Websphere [16] 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 [22] 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 failures, and correct behavior on recoveries. Furthermore, their experiments 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 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 and yield under high load. However, we demonstrate that the performance degradation of the database scheme is actually outweighed by its high availability, when performability is considered.

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 an extension of the approach we proposed in [24]. The extension involved an additional performance metric, a new model for quantifying availability, and new availability and performability metrics.

Our quantification approach differs from other previous studies in that we focus on multi-tiered, cluster-based services, and use a completely different set of metrics to evaluate availability and per-

formability. Perhaps most similar to our work is [7], which outlines a methodology for benchmarking systems' availability. Other works have proposed robustness [30] and reliability benchmarks [33] that quantify the degradation of system performance under faults.

Our work provides a formal but simple methodology to evaluate service performability in multi-tier systems. The previous work on performability analysis [11, 21, 31] involved a rich set of stochastic process models that describe system dependencies, fault likelihoods over time, and performance. Compared to these complex stochastic models, our models are much simpler, and thus more accessible to practitioners. Furthermore, as we mentioned in section 2, our performability analysis relies on actual fault-injection experiments and introduces simple performability metrics that penalize service designs heavily for their unavailability.

8 Conclusions

In this paper, we quantified the performance, availability, and performability of four soft state maintenance strategies in two complex cluster-based Internet services. Using a fault injection and modeling methodology, we isolated the effect of different faults on the availability of the two services. This isolation demonstrated that most of the unavailability of the services 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.

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 failures efficiently, especially when they are more prone to failures. Unfortunately, the common practice of over-provisioning all tiers achieves high performance and availability but at high hardware and maintenance costs. Thus, we conclude the provisioning and load balancing of server clusters needs to consider availability, cost, and performance explicitly. In future work, we plan to develop a framework for availability-aware provisioning and load balancing of server clusters.

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. In *Proceedings of the 5th Annual Workshop on Workload Characterization*, Nov. 2002.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the 4th USENIX Symposium on 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. In *Proceedings of USENIX'2000 Technical Conference*, San Diego, CA, 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, University of California, Berkeley, June 2000.

[5] BEA. BEA WebLogic. Available at http://www.bea.com/products/weblogic, Sept. 2003.

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

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

[8] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Snowbird, UT, June 2001.

[9] Cisco. Failover Configuration for LocalDirector. http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/locdf_wp.htm, 2003.

[10] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on 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 Transactions on Computers*, 47(1):96–107, Jan. 1998.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] 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. *To appear in a Special Issue of Computer Networks on Pervasive Computing*.

[14] F. Hanik. In memory session replication in tomcat4. http://www.theserverside.com/resources/article.jsp?l=Tomcat, 2002.

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

[16] IBM. IBM WebSphere. Available at http://www.ibm.com/websphere, Sept. 2003.

[17] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th Symposium on Reliable and Distributed Systems (SRDS '99)*, 1999.

[18] X. Li, R. P. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: A san-based fault-injection test-bed for the construction of highly available network services. In *In Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, Jan. 2002.

[19] B. Ling and A. Fox. A Self-tuning, Self-protecting, Self-healing Session State Management Layer. In *Proceedings of the 5th Annual Workshop on Active Middleware Services*, June 2003.

[20] D. D. E. Long, J. L. Carroll, and C. J. Park. A Study of the Reliability of Internet Sites. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 177–186, Pisa, Italy, Sept. 1991.

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

[22] Microsoft. ASP.NET. Available at http://www.asp.net/, Sept. 2003.

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

[24] 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. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[25] B. Olges. Simplify State Maintenance with ASP.NET. http://www.ftponline.com/wss/2002_10/online/bolges/, Oct. 2002.

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

[27] Rice University. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer/index.html, 2003.

[28] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Charlston, SC, Dec. 1999.

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

[30] D. Siewiorek, J. Hudakund, B. Suh, and Z. Segall. Development of a benchmark to measure system robustness. In *In Proceedings 23rd International Symposium Fault-Tolerant Computing*, pages 88–97, 1993.

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

[32] Transaction Processing Performance Council. TPC-W. http://www.tpc.org/, 2003.

[33] T. K. Tsai, R. K. Iyer, and D. Jewitt. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.