Agile Dynamic Provisioning of Multi-Tier Internet Applications

BHUVAN URGAONKAR
Pennsylvania State University
PRASHANT SHENOY
University of Massachusetts
ABHISHEK CHANDRA
University of Minnesota
PAWAN GOYAL
Veritas Software India
and
TIMOTHY WOOD
University of Massachusetts

Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. In this article, we propose a novel dynamic provisioning technique for multi-tier Internet applications that employs (1) a flexible queuing model to determine how much of the resources to allocate to each tier of the application, and (2) a combination of predictive and reactive methods that determine when to provision these resources, both at large and small time scales. We propose a novel data center architecture based on virtual machine monitors to reduce provisioning overheads. Our experiments on a forty-machine Xen/Linux-based hosting platform demonstrate the responsiveness of our technique in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes, thus maintaining response-time targets. Our technique also reduced the overhead of switching servers across applications from several minutes to less than a second, while meeting the performance targets of residual sessions.

This article is an extended version of the paper "Dynamic Provisioning of Multi-Tier Internet Applications" that appeared in Proceedings of the 2nd IEEE Conference on Autonomic Computing (ICAC'05). © IEEE 2005.
Authors' addresses: B. Urgaonkar, Department of CSE, Pennsylvania State University, University Park, PA 16802; email: bhuvan@cse.psu.edu; P. Shenoy, Department of Computer Science, University of Massachusetts, Amherst, MA 01003; email: shenoy@cs.umass.edu; A. Chandra, Department of CSE, University of Minnesota, Minneapolis, MN 55455; email: chandra@cs.umn.edu; P. Goyal, email: pawan.goyal@veritas.com; T. Wood, timwood@cs.umass.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2008 ACM 1556-4665/2008/03-ART1 $5.00 DOI 10.1145/1342171.1342172 http://doi.acm.org/10.1145/1342171.1342172

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Internet application, dynamic provisioning

ACM Reference Format:

Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. 2008. Agile dynamic provisioning of multi-tier internet applications. ACM Trans. Autonom. Adapt. Syst. 3, 1, Article 1 (March 2008), 39 pages. DOI = 10.1145/1342171.1342172 http://doi.acm.org/ 10.1145/1342171.1342172

1. INTRODUCTION

1.1 Motivation

An Internet hosting platform is a server farm that runs a distributed application such as an online retail store or an online brokerage site. Typical Internet applications employ a multi-tier architecture, with each tier providing a certain functionality. Such applications tend to see dynamically varying workloads that contain long-term variations such as time-of-day effects as well as short-term fluctuations due to flash crowds. Predicting the peak workload of an Internet application, and capacity provisioning based on these worst case estimates, is notoriously difficult. There are numerous documented examples of Internet applications that faced an outage due to an unexpected overload. For instance, the normally well-provisioned Amazon.com site suffered a forty-minute downtime due to an overload during the popular holiday season in November 2000 [Amazon 2000].

Given the difficulties in predicting peak Internet workloads, an application needs to employ a combination of dynamic provisioning and request policing to handle workload variations. Dynamic provisioning enables additional resources—such as servers—to be allocated to an application on-the-fly, to handle workload increases, while policing enables the application to temporarily turn away excess requests while additional resources are being provisioned.

In this article, we focus on dynamic resource provisioning of Internet applications that employ a multi-tier architecture. We argue that (1) provisioning of multi-tier applications raises new challenges not addressed by prior work on provisioning single-tier applications, and (2) agile, proactive provisioning techniques are necessary to handle both long-term and short-term workload fluctuations seen by Internet applications. To address these issues, we present a novel provisioning technique based on a combination of predictive and reactive mechanisms.

1.2 The Case for A New Provisioning Technique

Dynamic provisioning of resources—allocation and deallocation of servers to replicated applications—has been studied in the context of single-tier applications, of which clustered HTTP servers are the most common example. The notion of *hot spares* and their allocation to cluster-based applications on-demand was first proposed by Fox et al. [1997]. The Muse project proposed a utility-based

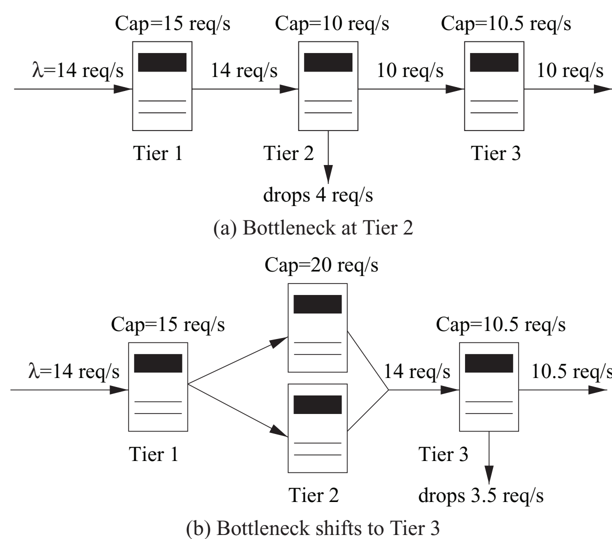


Fig. 1. Strawman approaches for provisioning a multi-tier application.

approach based on an economic theory for allocation and deallocation of servers to clustered Web servers [Chase and Doyle 2001]. A model-based approach for resource provisioning in single-tier Web servers was proposed by Doyle et al. [2003]. Kamra et al. [2004] model a multitier e-commerce application as a single M/GI/1 server and present a PI-controller-based admission control for maintaining response time targets. Whereas multi-tier Internet applications have been studied in the context of SEDA [Welsh and Culler 2003; Welsh et al. 2001], the effort focused on admission control issues to maintain target response times and did not explicitly consider provisioning issues.

It is nontrivial to extend provisioning mechanisms designed for single-tier applications to multi-tier scenarios. To understand why, we consider two strawman approaches that are simple extensions of the above single-tier methods and demonstrate their limitations for multi-tier applications.

Since many single-tier provisioning mechanisms have already been proposed, a straightforward extension is to employ such an approach at each tier of the application. This enables provisioning decisions to be made independently at each tier, based on local observations. Thus, our first strawman is to provision additional servers at a tier when the incoming request rate at that tier exceeds the currently provisioned capacity; this can be inferred by monitoring queue lengths, tier-specific response times, or request drop rates. We refer to this approach as *independent per-tier provisioning*.

Example 1. Consider the three-tier Internet application depicted in Figure 1(a). Initially, let us assume that one server each is allocated to the three tiers, and this enables the application to service 15, 10, and 10.5 requests/second at each tier (since a user request may impose different demands at different tiers, the provisioned capacity at each tier may be different). Let the incoming request rate be 14 requests/second. Given these capacities, all

requests are let in through the first tier, and 4 requests/second are dropped at the second tier. Due to these drops, the third tier sees a reduced request rate of 10 requests/second and is able to service them all. The effective good-put is therefore 10 requests/second. Since request drops are only seen at the second tier, this tier is perceived to be the bottleneck. The provisioning algorithm at that tier will allocate an additional server, doubling its effective capacity to 20 requests/second. At this point, the first two tiers are able to service all incoming requests and the third tier now sees a request rate of 14 requests/second (see Figure 1(b)). Since its capacity is only 10.5 requests/second, it drops 3.5 requests/second. Thus, the bottleneck shifts to the third tier, and the effective good-put only increases from 10 to 10.5 requests/second.

This simple example demonstrates that increasing the number of servers allocated to the bottleneck tier does not necessarily increase the effective good-put of the application. Instead, it may merely shift the bottleneck to a downstream tier. Although the provisioning mechanism at this downstream tier will subsequently increase its capacity, such shifting bottlenecks may require a number of independent provisioning steps at various tiers before the effective application capacity is actually increased. In the worst case, upto k provisioning steps, one at each tier, may be necessary in a k-tier application. Since allocation of servers to a tier entails overheads of several minutes or more [Chase and Doyle 2001], and since Internet workloads may spike suddenly, independent per-tier provisioning may be simply too slow to effectively respond to such workload dynamics.

Our second strawman models the multi-tier application as a black box and allocates additional servers whenever the observed response time exceeds a threshold.

Example 2. Consider the three-tier application from Example 1 with tier-specific capacities of 15, 20, and 10.5 requests/second as depicted in Figure 1(b). We ignore admission control issues in this example. Since the incoming request rate is 14 requests/second, the first two tiers are able to serve all requests, while the third saturates, causing request queues to buildup at this tier. This queue buildup increases the end-to-end response time of the application beyond the threshold. Thus, as in the single-tier case, a black box approach can successfully detect when additional servers need to be provisioned for the multi-tier application.

However, determining how many servers to provision, and where, is far more complex for multi-tier applications. First, since the application is treated as a black box, the provisioning mechanism can only detect an increase in end-to-end response times but cannot determine which tier is responsible for this increase. Second, for single-tier applications, an application model is used to determine how many servers are necessary to service all incoming requests with a certain response time threshold [Doyle et al. 2003]. Extending such models to multi-tier applications is nontrivial, since each tier has different characteristics. In a typical e-commerce application, for instance, this implies collectively modeling the effects of HTTP servers, Java application servers, and database

servers—a complex task. Third, not all tiers of the application may be replicable. For instance, the database tier is typically difficult to replicate on-the-fly. In the previous example, if the third tier is a nonreplicable database, the black box approach, which has no knowledge of individual tiers, will incorrectly signal the need to provision additional servers, when the correct action is to trigger request policing and let no more than 10.5 requests/second into the "black box."

This example demonstrates that due to the very nature of multi-tier applications, it is not possible to treat them as a black box for provisioning purposes. Knowledge of the number of tiers, their current capacities, and constraints on the degree of replication at each tier is essential for making proper provisioning decisions.

Both examples expose the limitations of using variants of single-tier provisioning methods for multi-tier applications. This article presents a multi-tier provisioning technique that overcomes these limitations.

1.3 Research Contributions

This article addresses the problem of dynamically provisioning capacity to a multi-tier application so that it can service its peak workload demand while meeting contracted response-time guarantees. The provisioning technique proposed in this article is tier-aware, agile, and able to take any tier-specific replication constraints into account. Our work has led to the following research contributions.

Predictive and reactive provisioning: Our provisioning technique employs two methods that operate at two different time scales—predictive provisioning, which allocates capacity at the time-scale of hours or days, and reactive provisioning, which operates at time scales of minutes to respond to flash crowds or deviations from expected long-term behavior. The combination of predictive and reactive provisioning is a novel approach for dealing with the multi-time-scale variations in Internet workloads.

Analytical modeling and incorporating tails of workload distributions: We present a flexible analytical model based on queuing theory to capture the behavior of applications with an arbitrary number of tiers. Our model determines the number of servers to be allocated to each tier based on the estimated workload. A novel aspect of our model-based provisioning is that it is based on the tail of the workload distribution—since capacity is usually engineered for the worst-case load, we use tails of probability distributions to estimate peak demand.

Fast server switching: Agile provisioning in a hosting platform requires the ability to quickly reallocate servers from one application to another. Doing so allows overloaded applications to be quickly allocated additional capacity on underloaded servers. To enable agile provisioning, we propose a novel technique that exploits the capabilities of virtual machines to significantly reduce server switching overheads. Our technique enables the system to be extremely agile to load spikes, with reaction times of tens of milliseconds.

Handling session-based workloads: Modern Internet workloads are predominantly session-based. Consequently, our techniques are inherently designed to

handle session-based workloads—they can account for multiple requests that comprise a session and the stateful nature of session-based Internet applications.

Implementation and experimentation: We implement our techniques on a forty-machine Linux-based hosting platform and use our prototype to conduct a detailed experimental evaluation using two open-source multi-tier applications. Our results show: (1) our model effectively captures key characteristics of multi-tier applications and overcomes the shortcomings inherent in existing provisioning techniques based on single-tier models, and (2) the combination of predictive and reactive mechanisms allows us to deal with predictable workload variations as well as unexpected spikes (during a flash crowd, our data center could double the capacity of a three-tier application within 5 minutes).

1.4 Relevance of Our Research to Adaptive and Autonomic Computing

The techniques presented in this article can be classified as adaptive or semi-autonomous, in the sense that they are designed to adapt to changing environmental conditions (the workload) with limited human intervention. While there are certain aspects of the system that require offline analysis, such as a subset of the application model parameterization, most other aspects of the system are completely online and purely observation-based, such as the construction of the predictive model, as well as reaction to recent workload behavior. Moreover, the provisioning mechanisms used in our system, such as agile VM-based server switching, are completely automated, and do not require any human intervention. We believe our work is a step towards a completely "autonomic" system that can employ more sophisticated learning techniques within the framework presented in this article to infer application behavior and workload characteristics on its own.

The remainder of this article is structured as follows. Sections 2 and 3 present an overview of the proposed system. Sections 4 and 5 present our provisioning algorithms, while Section 6 presents our fast server switching algorithm. We present our prototype implementation in 7 and our experimental evaluation in Section 8. We discuss related work in Section 9 and present our conclusions in Section 10.

2. SYSTEM OVERVIEW

This section presents an overview of Internet applications and the hosting platform assumed in our work.

2.1 Multi-Tier Internet Applications

Modern Internet applications are designed using multiple tiers. A multi-tier architecture provides a flexible, modular approach for designing such applications. Each tier provides a certain functionality, and the various tiers form a processing pipeline. Each tier receives partially processed requests from the previous tier and feeds these requests into the next tier, after local processing (see Figure 2). For example, an online bookstore can be designed using three tiers—a front-end Web server responsible for HTTP processing, a middle-tier

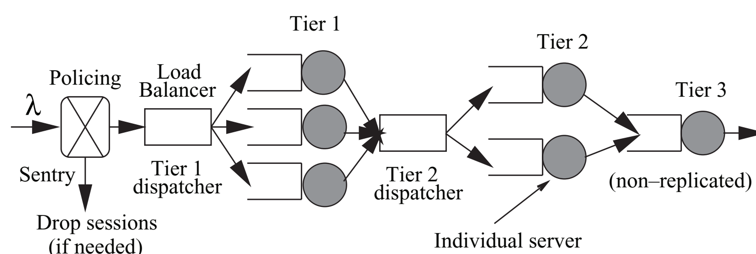


Fig. 2. Architecture of a 3-tier Internet application. In this example, tiers 1 and 2 are clustered, while tier 3 is not.

Java application server that implements the application logic, and a back-end database that stores catalogs and user orders.

The various tiers of an application are assumed to be distributed across different servers. Depending on the desired capacity, a tier may also be clustered. In an online bookstore, for example, the front-end tier can be a clustered Apache server that runs on multiple machines. If a tier is both clustered and replicable on-demand, it is assumed that the number of servers allocated to it, and thus the provisioned capacity, can be varied dynamically. Not all tiers may be replicable. For instance, if the back-end tier of the bookstore employs a database with *shared-nothing* architecture, it cannot be replicated on-demand. Database servers with a *shared-everything* architecture [oracle9i 2005], in contrast, can be clustered and replicated on-demand, but with certain constraints. We assume that each tier specifies its degree of replication, which is the limit on the maximum number of servers that can be allocated to it.[1]

Each clustered tier is also assumed to employ a load balancing element that is responsible for distributing requests to servers in that tier [Pai et al. 1998]. The workload of an Internet application is assumed to be session-based, where a session consists of a succession of requests issued by a client with think times in between. If a session is stateful, successive requests will need to be serviced by the same server at each tier, and the load balancing element will need to account for this server state when redirecting requests.

Every application also runs a special component called a sentry. The sentry polices incoming sessions to an application's server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads (see Figure 2). Observe that, unlike systems that use per-tier admission control [Welsh and Culler 2003], we assume a policer that makes a one-time admission decision when a session arrives. Once a session has been admitted, none of its requests can be dropped at any intermediate tier. Thus, sufficient capacity needs to be provisioned at various tiers to service all admitted sessions. Such a one-time policer avoids resource wastage resulting from partially serviced requests that may be dropped at later tiers.

[1]The degree of replication of a tier can vary from one to infinity, depending on whether the tier is partially, infinitely, or not, replicable.

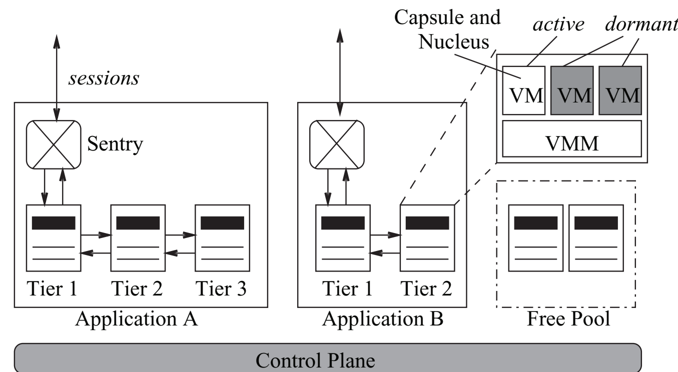


Fig. 3. Hosting platform architecture.

Finally, we assume that each application desires a performance bound from the hosting platform that is specified in the form of a service-level agreement (SLA). Our work assumes that the SLA is specified either in terms of the average response time or a suitable high percentile of the response time distribution (e.g., a SLA may specify that 95% of the requests should incur an end-to-end response time of no more than 1 second).

2.2 Hosting Platform Architecture

Our hosting platform is a data center that consists of a cluster of commodity servers interconnected by gigabit Ethernet. One or more high bandwidth links connect this cluster to the Internet. Each server in the hosting platform can take on one of the following roles: run an application component, run the control plane, or be part of the free pool (see Figure 3). The free pool contains all unallocated servers.

Servers Hosting Application Components: The hosting platform runs multiple third-party applications concurrently, in return for revenues [Chase and Doyle 2001; Shen et al. 2002; Urgaonkar et al. 2002]. This work assumes a dedicated hosting model, where each application runs on a subset of the servers and a server is allocated to at most one application at any given time.[2] The dedicated model is useful for running large clustered applications such as online mail [Saito et al. 1999], retail and brokerage sites, where server sharing is infeasible due to the client workload—the server pool is partitioned among applications running on the platform.

The component of an application that runs on a server is referred to as a *capsule*. Each server also runs a *nucleus*—a software component that performs online measurements of the capsule workload, its performance and resource usage; these statistics are periodically conveyed to the control plane.

Control Plane: The control plane is responsible for dynamic provisioning of servers to individual applications. It tracks the resource usage on servers, as

[2]A dedicated hosting model is different from shared hosting [Urgaonkar et al. 2002], where the number of applications exceeds the number of servers, and each server may run multiple applications concurrently.

reported by the nuclei, and determines the number of servers to be allocated to each application.

3. PROVISIONING ALGORITHM OVERVIEW

The goal of our provisioning algorithm is to allocate sufficient capacity to the tiers of an application so that its SLA can be met even in the presence of the peak workload. At the heart of any provisioning algorithm lies two issues: how much to provision, and when? We provide an overview of our provisioning algorithm from this perspective.

How much to provision. To address the issue of how many servers to allocate to each tier and each application, we construct an analytical model of an Internet application. Our model takes as input the incoming request rate and service demand of an individual request, and computes the number of servers needed at each tier to handle the aggregate demand.

We model a multi-tier application as a network of queues where each queue represents an application tier (more precisely, a server at an application tier), and the queues from a tier feed into the next tier. We model a server at a tier as a G/G/1 system, since it is sufficiently general to capture arbitrary arrival distributions and service time distributions.

By using this building block, which we describe in Section 4, we determine the number of servers necessary at each tier to handle a peak session arrival rate of λ, and provision resources accordingly. Our approach overcomes the drawbacks of independent per-tier provisioning and the black box approaches: (1) While the capacity needed at each tier is determined separately using our queuing model, the desired capacities are allocated to the various tiers all at once. This ensures that each provisioning decision immediately results in an increase in effective capacity of the application. (2) The use of a G/G/1 building block for a server at each tier enables us to break down the complex task of modeling an arbitrary multitier application into more manageable units. Our approach retains the ability to model each tier separately, while being able to reason about the needs of the application as a whole.

When to Provision. The decision of when to provision depends on the dynamics of Internet workloads. Internet workloads exhibit long-term variations such as time-of-day or seasonal effects, as well as short-term fluctuations such as flash crowds. While long-term variations can be predicted by observing past variations, short-term fluctuations are less predictable, or in some cases, not at all predictable. Our techniques employ two different methods to handle variations observed at different time scales. We use predictive provisioning to estimate the workload for the next few hours and provision for it accordingly. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. Whereas predictive provisioning attempts to stay ahead of the anticipated workload fluctuations, reactive provisioning enables the hosting platform to be agile to deviations from the expected workload.

The following sections present our queuing model, and the predictive and reactive provisioning methods.

4. HOW MUCH TO PROVISION: MODELING MULTI-TIER APPLICATIONS

To determine how many servers to provision for an application, we present an analytical model of a multi-tier application. Consider an application that consists of k tiers, denoted by $T_1, T_2, \ldots T_k$. Let the desired end-to-end response time for the application be R; this value is specified by the application's contracted SLA. Assume that the end-to-end response time is broken down into per-tier response times,[3] denoted by $d_1, d_2, \ldots, d_k$, such that $\sum d_i = R$. Let the incoming session rate be λ. Since capacity is typically provisioned based on the worst-case demand, we assume that λ is some high percentile of the arrival rate distribution—an estimate of the peak session rate that will be seen by the application.

Given the peak session rate and per-tier response times, our objective is to determine how many servers to allocate such that each tier can service all incoming requests with a mean response time of d_i.

Our model is based on a network of queues. Each server allocated to the application is represented by a queue (see Figure 2). Queues (servers) representing one tier feed into those representing the next tier. The first step in solving our model is to determine the capacity of an individual server in terms of the request rate it can handle. Given the capacity of a server, the next step computes the number of servers required at a tier to service the peak session rate. We model each server as a G/G/1 queuing system. In a G/G/1 queuing system, requests arrive at a server such that their interarrival times are derived from a fixed, known distribution. Each request brings with it a certain amount of work for the server to do. The time it takes the server to finish this work for a request, when serving only that request, is called the *service time* of the request. In a G/G/1 system, service times are assumed to be drawn from a known, fixed distribution. Requests are serviced in a first-come-first-served (FCFS) order. The queue is assumed to be infinitely long, meaning any request that arrives when the server is busy waits in the queue behind all the requests that arrived before it and have not yet been serviced. Finally, the servicing of requests is nonpreemptive. A G/G/1 system can express useful system metrics like average request response time and throughput, in terms of the interarrival and service time distributions. Since a G/G/1 system can handle an arbitrary arrival distribution and arbitrary service times, it enables us to capture the behavior of a various tiers such as HTTP, J2EE, and database servers.

The behavior of a G/G/1 system can be captured using the following queuing theory result [Kleinrock 1976]:

$$\lambda_i \geq \left[s_i + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (d_i - s_i)} \right]^{-1}, \tag{1}$$

where d_i is the mean response time for tier i, s_i is the average service time for a request at that tier, and λ_i is the request arrival rate to tier i. σ_a^2 and σ_b^2 are the variance of interarrival time and the variance of service time, respectively.

[3] Offline profiling can be used to break down the end-to-end response time into tier-specific response times.

Observe that d_i is known, while the per-tier service time s_i as well as the variance of interarrival and service times σ_a^2 and σ_b^2, can be monitored online in our system. By substituting these values into Equation 1, a lower bound on request rate λ_i that can serviced by a single server, can be obtained.

Given an average session think-time of Z, a session issues requests at a rate of $\frac{1}{Z}$. Using Little's Law [Kleinrock 1976], we can translate the *session* arrival rate of λ to a *request* arrival rate of $\frac{\lambda\tau}{Z}$, where τ is the average session duration. Therefore, once the capacity of a single server λ_i has been computed, the number of servers η_i needed at tier i to service a peak request rate of $\frac{\lambda\tau}{Z}$ is simply computed as

$$\eta_i = \left\lceil \frac{\beta_i \lambda \tau}{\lambda_i Z} \right\rceil, \tag{2}$$

where β_i is a tier-specific constant. The quantities Z and τ are estimated using online measurements. Note that implicit in the above calculation, is the assumption of perfect load-balancing among the servers comprising a tier. In complementary research some of the coauthors have explored enhancements to the model to incorporate the presence of load imbalances [Urgaonkar et al. 2005].

Observe that a single incoming request might trigger more than one request (unit of work) at intermediate tiers. For instance, a single search request at an online superstore might trigger multiple queries at the back-end database, one in the book catalog, one in the music catalog, and so on. Consequently, our model assumes that $\frac{\lambda\tau}{Z}$ incoming requests imposes an aggregate demand of $\beta_1\frac{\lambda\tau}{Z}$ requests at tier 1, $\beta_2\frac{\lambda\tau}{Z}$ requests at tier 2, and so on. The parameters $\beta_1, \ldots \beta_k$ are derived using online measurements. The value of β_i may be greater than one if a request triggers multiple units of work at tier i, or it may be less than one if caching at prior tiers reduces the demand at this tier.

An additional enhancement to our model is to incorporate any limits that capsules may have on the number of requests they can service simultaneously (e.g. the Apache Web server has a configurable upper limit on the number of processes that it can spawn to handle requests). This simply involves setting the capacity of a server to be the minimum of the capacity given by our model (expressed as the average number of requests that it can process simultaneously) and any concurrency limit that the capsule hosted on it may have.

Observe that our model can handle applications with an arbitrary number of tiers, since the complex task of modeling a multitier application is reduced to modeling an individual server at each tier. Equation 2 assumes that servers are homogeneous and that servers in each tier are load-balanced. Both assumptions can be relaxed as we show in a complementary paper [Urgaonkar et al. 2005].

The output of the model is the number of servers $\eta_1, \ldots \eta_k$ needed at the k tiers to handle a peak demand of λ. We then increase the capacity of all tiers to these values in a single step, resulting in an immediate increase in effective capacity. In the event η_i exceeds the degree of replication M_i of a tier, the actual allocation is reduced to this limit. Thus, each tier is allocated no more than $min(\eta_i, M_i)$ servers. To ensure that the SLA is not violated when the allocation is reduced to M_i, the excess requests must be turned away at the sentry.

5. WHEN TO PROVISION?

In this section, we present two methods—predictive and reactive—to provision resources over long and short time-scales, respectively.

5.1 Predictive Provisioning for the Long Term

The goal of predictive provisioning is to provision resources over time-scales of hours and days. The technique uses a workload predictor to predict the peak demand over the next several hours or a day, and then uses the model presented in Section 4 to determine the number of servers that are needed to meet this peak demand. Predictive provisioning is motivated by long-term variations such as time-of-day or seasonal effects exhibited by Internet workloads [Hellerstein et al. 1999]. For instance, the workload seen by an Internet application typically peaks around noon every day and is minimum in the middle of the night. Similarly, the workload seen by online retail Web sites is higher during the holiday shopping months of November and December than other months of the year. These cyclic patterns tend to repeat and can be predicted ahead of time by observing past variations. By employing a workload predictor that can predict these variations, our predictive provisioning technique can allocate servers to an application well ahead of the expected workload peak. This ensures that application performance does not suffer even under the peak demand.

The key to predictive provisioning is the workload predictor. In this section, we present a workload predictor that estimates the tail of the arrival rate distribution (the peak demand) for the next few hours. Other statistical workload predictive techniques proposed in the literature can also be used with our predictive provisioning technique [Hellerstein et al. 1999; Rolia et al. 2002].

Our workload predictor is based on a technique proposed by Rolia et al. [2002] and uses past observations of the workload to predict peak demand that will be seen over a period of T time units. For simplicity of exposition, assume that $T = 1$ hour. In that case, the predictor estimates the peak demand that will be seen over the next one hour, at the beginning of each hour. To do so, it maintains a history of the session arrival rate seen during each hour of the day over the past several days. A histogram is then generated for each hour using observations for that hour from the past several days (see Figure 4). Each histogram yields a probability distribution of the arrival rate for that hour. The peak workload for a particular hour is estimated as a high percentile of the arrival rate distribution for that hour (see Figure 4). Thus, by using the tail of the arrival rate distribution to predict peak demand, the predictive provisioning technique can allocate sufficient capacity to handle the worst-case load, should it arrive. Further, monitoring the demand for each hour of the day enables the predictor to capture time-of-day effects. Since workloads of Internet applications exhibit occasional overloads, the peak values of their resource needs often far exceed the resources that they need most of the time. Using an appropriate high percentile of the resource need distribution allows our system to prevent the wastage of resources that would result from provisioning based on the peak values. The reactive component of our scheme, described in the next section, is

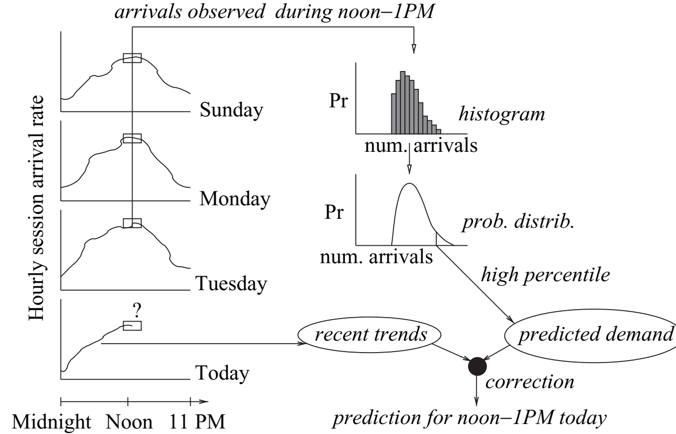


Fig. 4. The workload prediction algorithm.

designed to add capacity during such overloads when the provisioned capacity falls short of the application needs.

In addition to using observations from prior days, the workload seen in the past few hours of the current day can be used to further improve prediction accuracy. Suppose that $\lambda_{pred}(t)$ denotes the predicted arrival rate during a particular hour denoted by t. Further let $\lambda_{obs}(t)$ denote the actual arrival rate seen during this hour. The prediction error is simply $\lambda_{obs}(t) - \lambda_{pred}(t)$. In the event of a consistently *positive* prediction error over the past few hours, indicating that the predictor is consistently underestimating peak demand, the predicted value for the next hour is corrected using the observed error:

$$\lambda_{pred}(t) = \lambda_{pred}(t) + \sum_{i=t-h}^{t-1} \frac{max(0, \lambda_{obs}(i) - \lambda_{pred}(i))}{h},$$

where the second expression denotes the mean prediction error over the past h hours. We only consider positive errors in order to correct underestimates of the predicted peak demand—negative errors indicate that the observed workload is *less* than the peak demand, which only means that the worst-case workload did not arrive in that hour and is not necessarily a prediction error.

Using the predicted peak arrival rate for each application, the predictive provisioning technique uses the model to determine the number of servers that should be allocated to each tier of an application. An increase in allocation must be met by borrowing servers from the free pool, or underloaded applications—underloaded applications are those whose new allocations are less than their current allocations. If the total number of required servers is less than the servers available in the free pool and those released by underloaded applications, then a utility-based approach [Chase and Doyle 2001] can be used to arbitrate the allocation of available servers to needy applications—servers are allocated to applications that benefit most from it as defined by their utility functions.

5.2 Reactive Provisioning: Handling Prediction Errors and Flash Crowds

The workload predictor outlined in the previous section is not perfect—it may incur prediction errors if the workload on a given day deviates from its behavior on previous days. Further, sudden load spikes or flash crowds are inherently unpredictable phenomena. Finally, errors in the online measurements of the model parameters can translate into errors in the allocations computed by the model. Reactive provisioning is used to swiftly react to such unforeseen events. Reactive provisioning operates on short time scales—on the order of minutes—checking for workload anomalies. If any such anomalies are detected, then it allocates additional capacity to various tiers to handle the workload increase.

Reactive provisioning is invoked once every few minutes. It can also be invoked on-demand by the application sentry if the observed request drop rate increases beyond a threshold. In either case, it compares the currently observed session arrival rate $\lambda_{obs}(t)$ over the past few minutes, to the predicted rate $\lambda_{pred}(t)$. If the two differ by more than a threshold, corrective action is necessary. Specifically if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} > \tau_1$ or drop rate $> \tau_2$, where τ_1 and τ_2 are application-defined thresholds, then it computes a new allocation of servers. This can be achieved in one of two ways. One approach is to use the observed arrival rate $\lambda_{obs}(t)$ in Equation 2 of the model to compute a new allocation of servers for the various tiers. The second approach is to increase the allocation of all tiers that are at or near saturation by a constant amount (e.g., 10%). The new allocation needs to ensure that the bottleneck does not shift to another downstream tier; the capacity of any such tiers may also need to be increased proportionately. The advantage of using the model to compute the new allocation is that it yields the new capacity in a single step, as opposed to the latter approach that increases capacity by a fixed amount. The advantage of the latter approach is that it is independent of the model and can handle any errors in the measurements used to parameterize the model. In either case, the effective capacity of the application is raised to handle the increased workload.

The additional servers are borrowed from the free pool if available. If the free pool is empty or has insufficient servers, then these servers need to be borrowed from other underloaded applications running on the hosting platform. An application is said to be underloaded if its observed workload is significantly lower than its provisioned capacity: if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} < \tau_{low}$, where τ_{low} is a low watermark threshold.

Since a single invocation of reactive provisioning may be insufficient to bring sufficient capacity online during a large load spike, repeated invocations may be necessary in quick succession to handle the workload increase.

Together, predictive and reactive provisioning can handle long-term predictable workload variations as well as short term fluctuations that are less predictable. Predictive provisioning allocates capacity ahead of time in anticipation of a certain peak workload, while reactive provisioning takes corrective action *after* an anomalous workload increase has been observed. Put another way, predictive provisioning attempts to stay ahead of the workload

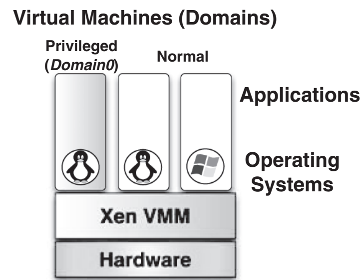


Fig. 5. The Xen VMM hosting multiple VMs.

fluctuations, while reactive provisioning follows workload fluctuations, correcting for errors.

5.3 Request Policing

The predictor and reactor convey the peak session arrival rate for which they have allocated capacity to the application's sentry. This is done every time the allocation is changed. The sentry then ensures that the admission rate does not exceed this threshold—excess sessions are dropped at the sentry. Note that in our system, admission control decisions are made only in front of the first tier of the application. Once admitted, a request is not explicitly dropped at a subsequent tier within the application. This is in contrast to some related work [Welsh and Culler 2003], where each tier employs its own admission control. Dropping a request beyond the first tier results in wastage of resources at all the tiers that processed it. Any system with per-tier admission control can be converted into one where only the first tier performs admission control. By modeling all the tiers and their interactions, our multi-tier model allows us to integrate admission control decisions for various tiers into a single sentry.

6. AGILE SERVER SWITCHING USING VMMS

A Virtual Machine Monitor (VMM) is a software layer that virtualizes the resources of a physical server and supports the execution of multiple virtual machines (VMs) [Goldberg 1974]. Each VM runs a separate operating system and an application capsule within it. The VMM enables server resources, such as the CPU, memory, disk and network bandwidth, to be partitioned among the resident virtual machines. Figure 5 shows a hypothetical Xen VMM [Barham et al. 2003] hosting VMs running two different operating systems.

Traditionally VMMs have been employed in shared hosting environments to run multiple applications and their VMs on a single server; the VM provides isolation across applications while the VMM supports flexible partitioning of server resources across applications. In dedicated hosting, no more than one application can be active on a given physical server, and as a result, sharing of individual server resources across applications is moot in such environments. Instead, we employ VMMs for a novel purpose—fast server switching.

Traditionally, switching a server from one application to another for purposes of dynamic provisioning has entailed overheads of several minutes or more. Doing so involves some or all of the following steps: (1) wait for residual sessions

of the current application to terminate, (2) terminate the current application, (3) scrub and reformat the disk to wipe out sensitive data, (4) reinstall the OS, (5) install and configure the new application. Our hosting platform runs a VMM on each physical server. Doing so enables it to eliminate many of these steps and drastically reduces switching time.

6.1 Techniques for Agile Server Switching

We assume that each Elf server runs multiple virtual machines and capsules of different applications within it. Only one capsule and its virtual machine is active at any time—this is the capsule to which the server is currently allocated. Other virtual machines are dormant—they are allocated minimal server resources by the underlying VMM and most server resources are allocated to the active VM. If the server belongs to the free pool, all of its resident VMs are dormant.

In such a scenario, switching an Elf server from one application to another implies deactivating a VM by reducing its resource allocation to ϵ, and reactivating a dormant VM by increasing its allocation to (100-ϵ)% of the server resources.[4] This only involves adjusting the allocations in the underlying VMM and incurs overheads on the order of tens of milliseconds. Thus, in theory, our hosting platform can switch a server from one application to another in a few milliseconds. In practice, however, we need to consider the residual state of the application before it can be made dormant. Figure 6 illustrates the process of activating a dormant VM hosting a replica of tier-2 of a hypothetical 3-tier application.

To do so, we assume that once the predictor or the reactor decide to reassign a server from an underloaded to an overloaded application, they notify the load balancing element of the under-loaded application tier. The load balancing element stops forwarding new sessions to this server. However, the server retains existing sessions and new requests may arrive for those sessions until they terminate. Consequently, the underloaded application tier will continue to use some server resources and the amount of resources required will diminish over time as existing sessions terminate. As a result, the allocation of the currently active VM cannot be instantaneously ramped down; instead the allocation needs to be reduced gradually, while increasing the allocation of the VM belonging to the overloaded application. Two strategies for ramping down the allocation of the current VM are possible.

—Fixed rate ramp-down: In this approach, the resource allocation of the underloaded VM is reduced by a fixed amount δ every t time units until it reduces to ϵ; the allocation of the new VM is increased correspondingly. The advantage of this approach is that it switches the server from one application to another in a fixed amount of time, namely t/δ. The limitation is that long-lived residual sessions will be forced to terminate, or their performance guarantees will be violated if the allocation decreases beyond that necessary to service them.

[4] ϵ is a small value such that the VM consumes negligible server resources and its capsule is idle and swapped out to disk.

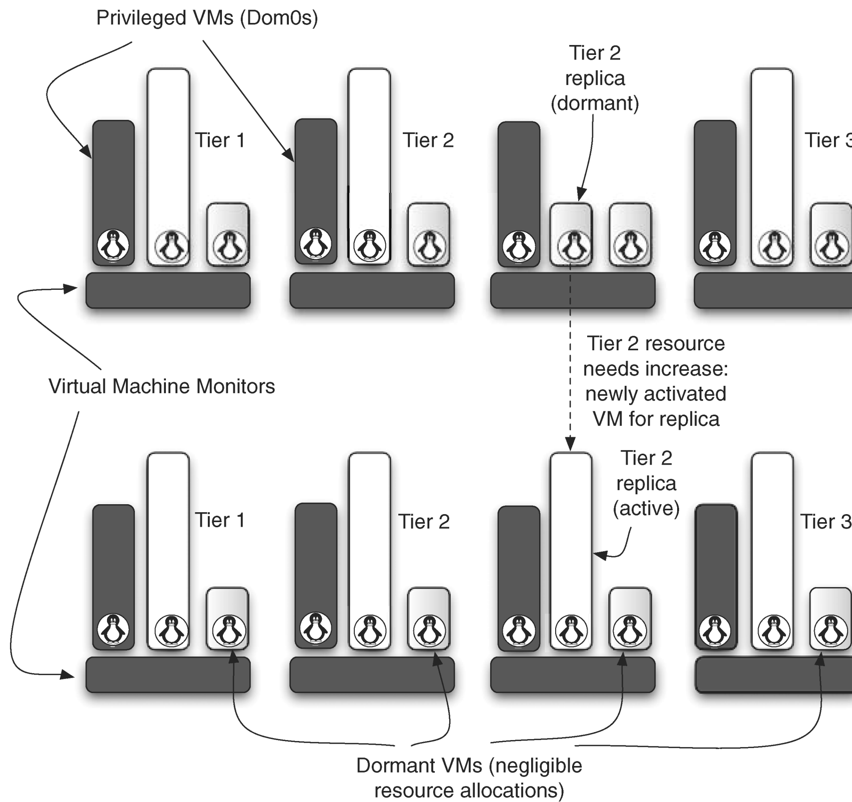


Fig. 6. Illustration of agile server switching. A 3-tier application is shown here. Each tier actively uses one server in the first configuration. Due to increased resource needs for tier-2, its dormant replica is activated by increasing its resource allocations.

—Measurement-based ramp-down: In this approach, the actual resource usage of the underloaded VM is monitored online. As the resource usage decreases with terminating sessions, the underlying allocation in the VMM is also reduced. This approach requires monitoring of the CPU, memory, network, and disk usage so that the allocation can match the falling usage. The advantage of this approach is that the ramp-down is more conservative and less likely to violate performance guarantees of existing sessions. The drawback is that long-lived sessions may continue to use server resources, which increases the server switching time.

In either case, use of VMMs enables our hosting platform to reduce system switching overheads. The switching time is solely dominated by application idiosyncrasies. If the application has short-lived sessions or the application tier is stateless, the switching overhead is small. Even when sessions are long-lived, the overloaded application immediately gets some resources on the server, which increases its effective capacity; more resources become available as the current VM ramps down.

As a final detail, observe that we have assumed that sufficient dormant VMs are always available for various tiers of an overloaded application to arbitrarily increase its capacity. The hosting platform needs to ensure that there is always a prespawned pool of dormant VMs for each application in the system. As dormant VMs of an application are activated during an overload, and the number of dormant VMs falls below a low limit, additional dormant VMs need to be spawned on other Elf servers, so that there is always a ready pool of VMs that can be tapped.

6.2 Do VMMs Render Predictive Provisioning Unnecessary?

Given the agile switching of servers enabled by the use of VMMs, it is tempting to argue in favor of a purely reactive provisioning mechanism: such a provisioning mechanism might be able to match the server allocations for various applications with their workloads by quickly moving servers where they are needed. However, such a purely reactive scheme has the following shortcomings:

(1) The agility of reactive provisioning is crucially dependent on when it is conducted. For example, in our system, adding a replica to a tier to deal with an increased workload may involve transferring a large image over the network to activate a dormant VM. This transfer time would depend on the network utilization—the higher the network utilization, the higher the transfer time. Predictive mechanisms can identify the most opportune times for conducting such transfers, thereby assisting in fast reactive provisioning. We will demonstrate this phenomenon using an experiment in Section 8.3.2.

(2) Due to the presence of residual sessions on a server being moved from one application (call it *lender*) to another (call it *borrower*), it may take up to several minutes for the server to become fully available to *borrower*. Therefore, despite using VMMs, a reactive provisioning scheme may not yield the best possible switching times. The situation worsens if most servers in the hosting platform are being utilized (because this increases the probability of a server having residual sessions when the reactor decides to move it from one application to another). Additionally, long-lasting and resource-intensive sessions will further exacerbate this problem. With a predictor in addition to the reactor, the provisioning algorithm can start moving servers with residual sessions, and stop admitting new sessions of *lender*, well in time for the server to be available to *borrower* when it needs it.

(3) If the workload of an application changes quickly, the actions of a purely reactive provisioning mechanism may lag the workload. In such cases, the application may experience degraded performance untill the reactor has pulled in enough servers to meet the application's needs.

7. IMPLEMENTATION CONSIDERATIONS

We implemented a prototype data center on a cluster of 40 Pentium servers connected via a 1 Gbps Ethernet switch and running Linux 2.4.20. Each machine

in the cluster ran one of the following entities: (1) an application capsule and its nucleus, or load balancer, (2) the control plane, (3) a sentry, (4) a workload generator for an application. The applications used in our evaluation (described in detail in Section 8.1) had two replicable tiers—front tier based on the Apache Web server and a middle tier based on Java servlets hosted on the Tomcat servlets container. The third tier was a nonreplicable Mysql database server.

Virtual Machine Monitor. We use Xen 1.2 [Barham et al. 2003] as the virtual machine monitor in our prototype. The Xen VMM has a special virtual machine called domain0 (virtual machines are called domains in the Xen terminology) that gets created as soon as Xen boots and remains throughout the VMM's existence. Xen provides a management interface that can be manipulated from domain0 to create new domains, control their CPU, network, and memory resource allocations, allocate IP addresses, grant access to disk partitions, and suspend/resume domains to files, and so forth. The management interface is implemented as a set of library functions implemented in C, for which there are Python language bindings. We use a subset of this interface—`xc_dom_create.py` and `xc_dom_control.py` to provide ways to start a new domain or stop an existing one; the control plane implements a script that remotely logs on to domain0 and invokes these scripts. The control plane also implements scripts that can remotely log onto any existing domain to start a capsule and its nucleus or stop them. `xc_dom_control.py` provides an option that can be used to set the CPU share of an existing domain. The control plane uses this feature for VM ramp-up and ramp-down.

Nucleus. The nucleus was implemented as a user-space daemon that periodically (once every 15 minutes in our prototype) extracts information about tier-specific requests needed by the provisioning algorithms and conveys it to the control plane. Our nuclei use a combination of (1) online measurements of resource usages and request performance, (2) real-time processing of logs provided by the application software components, and (3) offline measurements to determine various quantities needed by the control plane. To enable low overhead recording of online measurements, the logs are written to named pipes that are read by the nuclei. We made simple modifications to Apache and Tomcat to record the average service time, s_i, of a request at these tiers. For Mysql, s_i was determined using offline profiling [Urgaonkar et al. 2002]. The variance of service time, σ_b^2, was determined from observations of individual service times. We configured Apache and Tomcat by turning on the appropriate options in their configuration files, to have them record the arrival and residence times of individual requests into their logs. The logs were written to named pipes and processed in real-time by the nuclei to determine, σ_a^2, the variance of the request interarrival time. The parameter β_i for tier i was estimated by the control plane as the ratio of the number of requests reported by the nuclei at that tier and the number of requests admitted by the sentry during the last period. Finally, the nuclei used the sysstat package [SAR 2005] for online measurements of resource usages of capsules used by the reactive provisioning and by the measurement-based strategy for ramping down the allocation of a VM.

Sentry and Load Balancer. We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [KTCPVS 2005] to implement the policing mechanisms described in Section 5.3. ktcpvs is an open-source, layer-7 request dispatcher implemented as a Linux module. A round-robin load balancer implemented in ktcpvs was used for Apache. Load balancing for the Tomcat tier was performed by *mod_jk*, an Apache module that implements a variant of round robin request distribution while taking into account session affinity. The sentry keeps records of arrival and finish times of admitted sessions as well as each request within a session. These observations are used to estimate the average session duration τ and the average think time Z.

Control Plane. The control plane is implemented as a daemon running on a dedicated machine. It implements the predictive and reactive provisioning techniques described in Section 5. The control plane invokes the predictive provisioning algorithm periodically and conveys the new server allocations or deallocations to the affected sentries and load balancers. It communicates with the concerned virtual machine monitors to start or stop capsules and nuclei. Reactive provisioning is invoked by the sentries once every 5 minutes.

8. EXPERIMENTAL EVALUATION

In this section we present the experimental setup followed by the results of our experimental evaluation.

8.1 Experimental Setup

The control plane was run on a dual-processor 450 MHz machine with 1 GB RAM. Elf and Ent servers had 2.8 GHz processors and 512 MB RAM. The sentries were run on dual-processor 1GHz machines with 1 GB RAM. Finally, the workload generators were run on uniprocessor machines with 1 GHz processors. Elves and Ents ran the Xen 1.2 VMM with Linux; all other machines ran Linux 2.4.20. All machines were interconnected by gigabit Ethernet.

We used two open-source multi-tier applications in our experimental study. *Rubis* implements the core functionality of an eBay-like auction site: selling, browsing, and bidding. It implements three types of user sessions, has nine tables in the database and defines 26 interactions that can be accessed from the clients' Web browsers. *Rubbos* is a bulletin-board application modeled after an online news forum like Slashdot. Users have two different levels of access: regular user and moderator. The main tables in the database are the users, stories, comments, and submissions tables. Rubbos provides 24 Web interactions. Both applications were developed by the DynaServer group at Rice University [DYNASERVER 2005]. Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate workloads and take measurements needed by our experiments. Rubis and Rubbos sessions had an average duration of 15 minutes and 5 minutes, respectively. For both applications, the average think time was 5 seconds.

We used 3-tier versions of these applications. The front tier was based on the Apache 2.0.48 Web server. The middle tier was based on Java servlets that

Table I. Per-Tier Workload Characteristics for Rubis

Parameter	Apache	Tomcat	Mysql
d_i	80 msec	400 msec	320 msec
s_i	20 msec	294 msec	254 msec
σ_a^2	848	2304	1876
σ_b^2	0	3428	4024
Degree of replication	inf	inf	1
Concurrency limit per replica	256	256	2000

Table II. Per-Tier Workload Characteristics for Rubbos

Parameter	Apache	Tomcat	Mysql
d_i	80 msec	400 msec	320 msec
s_i	20 msec	320 msec	278 msec
σ_a^2	656	2018	1486
σ_b^2	0	4324	2564
Degree of replication	inf	inf	1
Concurrency limit per replica	256	256	2000

Table III. Session Characteristics for Rubis and Rubbos

Parameter	Rubis	Rubbos
Z	5 sec	5 sec
τ	15 min	5 min

implement the application logic. We employed Tomcat 4.1.29 as the servlets container. Finally, the database tier was based on the Mysql 4.0.18 database.

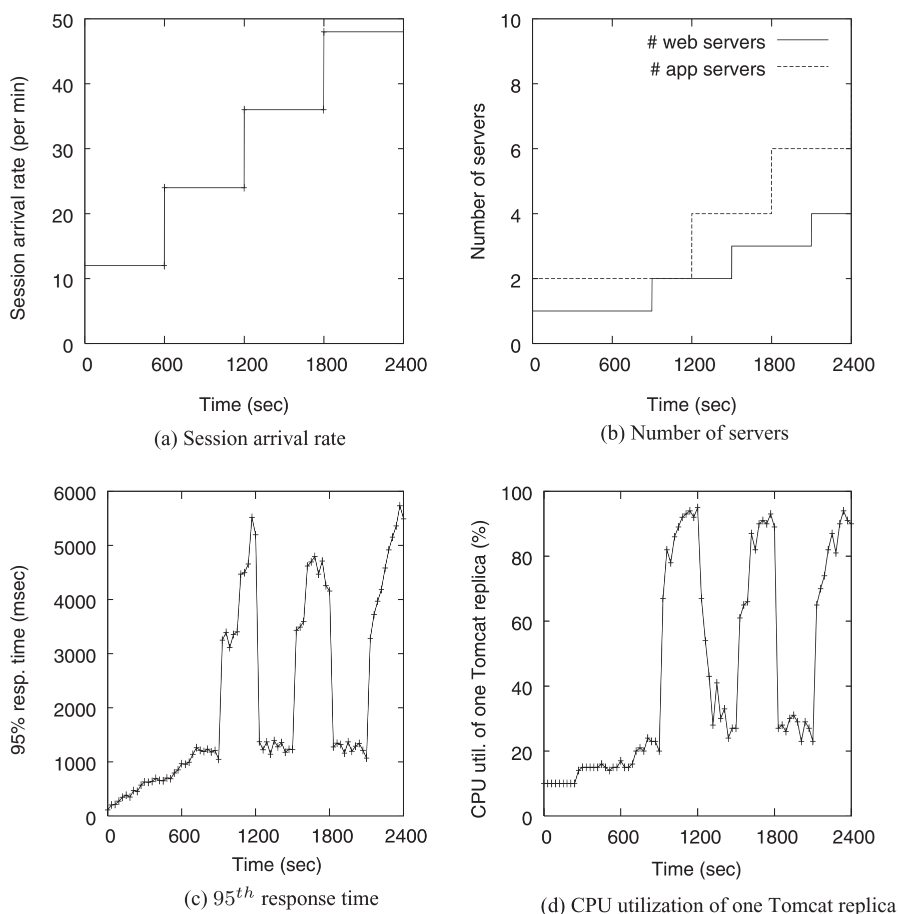
Both applications are assumed to require an SLA where the 95th percentile of the response time is no greater than 2 seconds. We use a simple heuristic to translate this SLA into an equivalent SLA specified using the average response time—since the model in Section 4 uses mean response times, such a translation is necessary. We use application profiling [Urgaonkar et al. 2002] to determine a distribution whose 95th percentile is 2 seconds and use the mean of that distribution for the new SLA. The per-tier average delay targets d_1, d_2, and d_3 were then set to be 10, 50, and 40% of the mean response time, for Apache, Tomcat, and Mysql respectively.

We present the values for various parameters used by our multi-tier models for our applications, in Tables I and II. Since these values are either, (1) updated once every 15 minutes based on online measurements for Apache and Tomcat tiers or, (2) recorded over 15 minute periods based on offline measurements for Mysql, for each parameter, we sort the observed values in an increasing order and report the 95th percentile.

Finally, we present the parameters of our model that capture session characteristics for these applications in Table III.

8.2 Effectiveness of Multi-Tier Model

This section demonstrates the effectiveness of our multi-tier provisioning technique over variants of single-tier methods.



(a) Session arrival rate

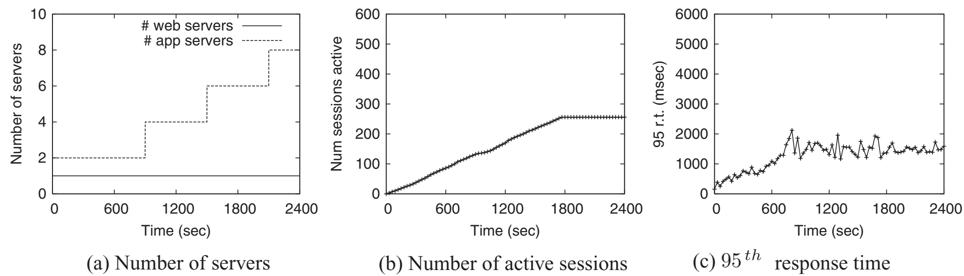
(b) Number of servers

(c) 95th response time

(d) CPU utilization of one Tomcat replica

Fig. 7. Rubbos: Independent per-tier provisioning.

8.2.1 *Independent Per-Tier Provisioning.* Our first experiment uses the Rubbos application. We use the first strawman described in Example 1 of Section 1 for provisioning Rubbos. Here, each tier employs its own provisioning technique. Rubbos was subjected to a workload that increases in steps, once every ten minutes (see Figure 7(a)). The first workload increase occurs at $t = 600$ seconds and saturates the tier-1 Web server. This triggers the provisioning technique, and an additional server is allocated at $t = 900$ seconds (see Figure 7(b)). At this point, the two tier-1 servers are able to service all incoming requests, causing the bottleneck to shift to the Tomcat tier. The Elf running Tomcat saturates, which triggers provisioning at tier 2. An additional server is allocated to tier 2 at $t = 1200$ seconds (see Figure 7(b)). The second workload increase occurs at $t = 1200$ seconds and the cycle repeats. As shown in Figure 7(c), since multiple provisioning steps are needed to yield an effective increase in capacity, the application SLA is violated during this period. Finally, Figure 7(d) presents the CPU utilization



(a) Number of servers (b) Number of active sessions (c) 95th response time

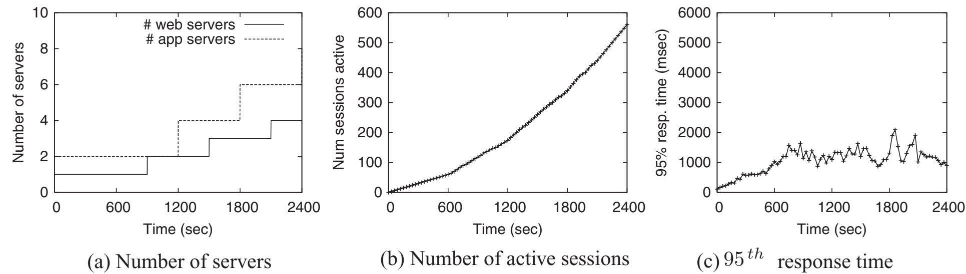
Fig. 8. Rubbos: Provision only the Tomcat tier.

of the server hosting the initial Tomcat replica throughout the experiment. As seen, the response time spikes correspond to the CPU getting saturated and subside when the addition of a new replica helps reduce the stress on the CPU.

A second strawman is to employ dynamic provisioning only at the most compute-intensive tier of the application, since it is the most common bottleneck [Villela et al. 2004]. In Rubbos, the Tomcat tier is the most compute intensive of the three tiers and we only subject this tier to dynamic provisioning. The Apache and Tomcat tiers were initially assigned 1 and 2 servers respectively. The capacity of a Tomcat server was determined to be 40 simultaneous sessions using our model, while Apache was configured with a connection limit of 256 sessions. As shown in Figure 8(a), every time the current capacity of the Tomcat tier is saturated by the increasing workload, two additional servers are allocated. The number of servers at tier 2 increases from 2 to 8 over a period of time. At $t = 1800$ seconds,the session arrival rate increases beyond the capacity of the first tier, causing the Apache server to reach its connection limit of 256. Subsequently, even though plenty of capacity was available at the Tomcat tier, newly arriving sessions are turned away due to the connection bottleneck at Apache, and the throughput reaches a plateau (see Figure 8(b)). Thus, focusing only on the the commonly bottlenecked tier is not adequate, since the bottleneck will eventually shift to other tiers.

Next, we repeat this experiment with our multi-tier provisioning technique. Since our technique is aware of the demands at each tier and can take idiosyncrasies such as connection limits into account, as shown in Figure 9(a), it is able to scale the capacity of both the Web and the Tomcat tiers with increasing workloads. Consequently, as shown in Figure 9(b), the application throughput continues to increase with the increasing workload. Figure 9(c) shows that the SLA is maintained throughout the experiment.

Result: Existing single-tier methods are inadequate for provisioning resources for multi-tier applications as they may fail to capture multiple bottlenecks. Our technique anticipates shifting bottlenecks due to capacity addition at a tier and increases capacity at all needy tiers. Further, it can identify different bottleneck resources at different tiers, for example, CPU at the Tomcat tier and Apache connections at the Web tier.



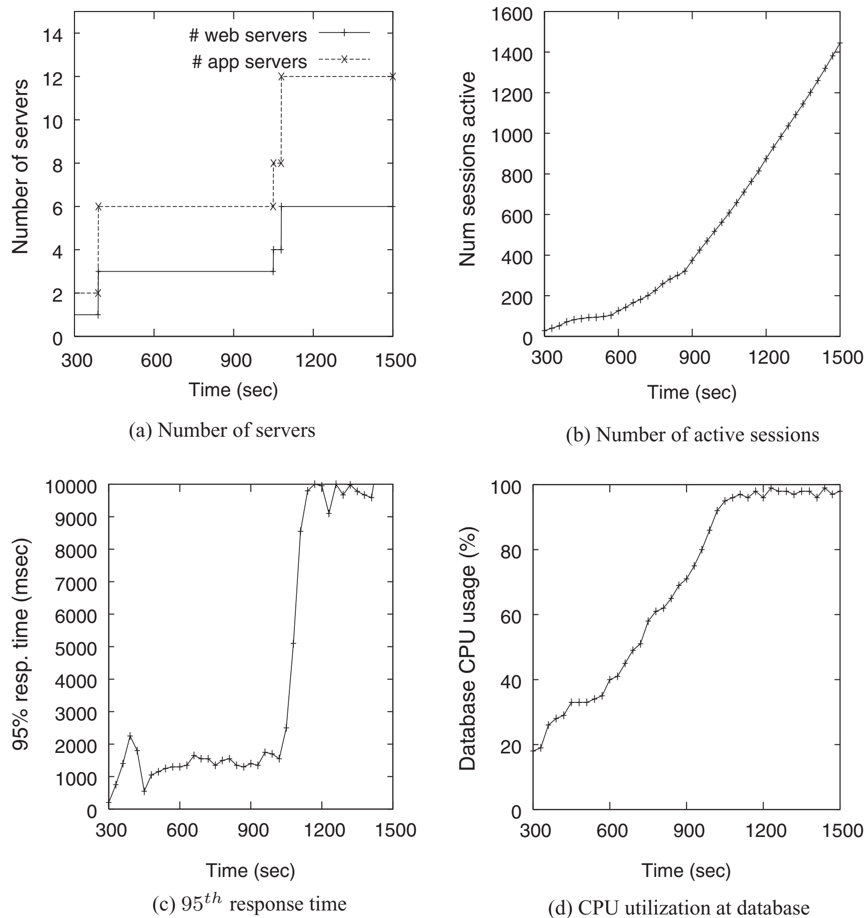
(a) Number of servers (b) Number of active sessions (c) 95th response time

Fig. 9. Rubbos: Model-based multi-tier provisioning.

8.2.2 *The Black Box Approach.* We subjected the Rubis application to a workload that increased in steps, as shown in Figure 11(a). First, we use the black box provisioning approach described in Example 2 of Section 1. The provisioning technique monitors the per-request response times over 30s intervals and signals a capacity increase if the 95th percentile response time exceeds 2 seconds. Since the black box technique is unaware of the individual tiers, we assume that two Tomcat servers and one Apache server are added to the application every time a capacity increase is signaled. As shown in Figures 10(a) and (c), the provisioned capacity keeps increasing with increasing workload, and whenever the 95th percentile of response time is over 2 seconds. However, as shown in Figure 10(d), at $t = 1100$ seconds, the CPU on the Ent running the database, saturates. Since the database server is not replicable, increasing the capacity of the other two tiers beyond this point does not yield any further increase in effective capacity. However, the black box approach is unaware of where bottleneck lies and continues to add servers to the first two tiers until it has used up all available servers. The response time continues to degrade despite this capacity addition as the Java servlets spend increasingly larger amounts of time waiting for queries to be returned by the overloaded database (see Figures 10(c) and (d)).

We repeat this experiment using our multi-tier provisioning technique. Our results are shown in Figure 11. As shown in Figure 11(b), the control plane adds servers to the application at $t = 390$ seconds in response to the increased workload. However, beyond this point, no additional capacity is allocated. Our technique correctly identifies that the capacity of the database tier for this workload is around 600 simultaneous sessions. Consequently, when this capacity is reached and the database saturates, it triggers policing instead of provisioning. The admission control is triggered at $t = 1070$ seconds and drops any sessions in excess of this limit during the remainder of the experiment. Figure 11(d) shows that our provisioning is able to maintain a satisfactory response time throughout the experiment.

Result: Our provisioning technique is able to take constraints imposed by nonreplicable tiers into account. It can maintain response-time targets by invoking the admission control when capacity addition does not help.



(a) Number of servers

(b) Number of active sessions

(c) 95th response time

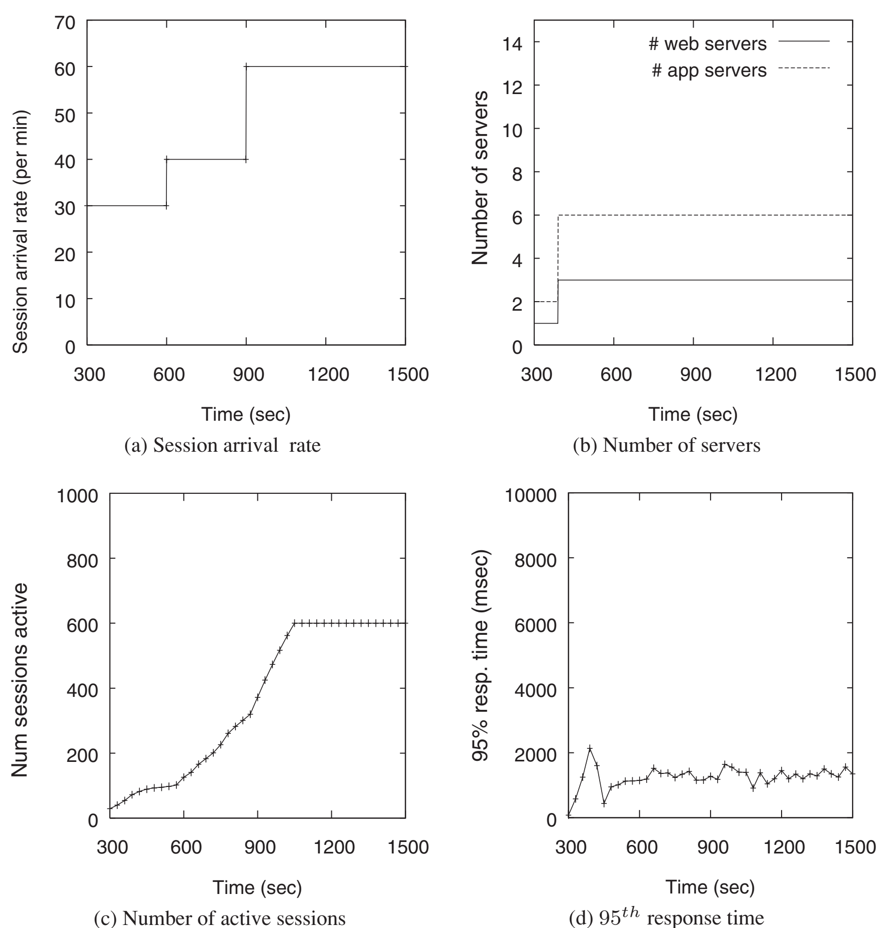
(d) CPU utilization at database

Fig. 10. Rubis: Black box provisioning.

8.3 The Need for Both Reactive and Predictive Provisioning

In this section, we first evaluate the agile server switching mechanism based on the use of VMMs. Following this we present experiments to demonstrate the need to have both predictive and reactive provisioning mechanisms.

We used Rubis in these experiments. The workload was generated based on the Web traces from the 1998 Soccer World Cup site [Arlitt and Jin 1999]. These traces contained the number of arrivals per minute to this Web site over an 8-day period. Based on these we created several smaller traces to drive our experiments. These traces were obtained by compressing the original 24-hour long traces to 1 hour—this was done by picking arrivals for every 24th minute and discarding the rest. This enables us to capture the time-of-day effect as a time-of-hour effect. Further, we reduced the workload intensity by reducing the number of arrivals by a factor of 100. The experiment invoked predictive provisioning once every 15 minutes over the one hour duration and we refer to these periods as *Intervals 1–4*; reactive provisioning was invoked on-demand or



(a) Session arrival rate

(b) Number of servers

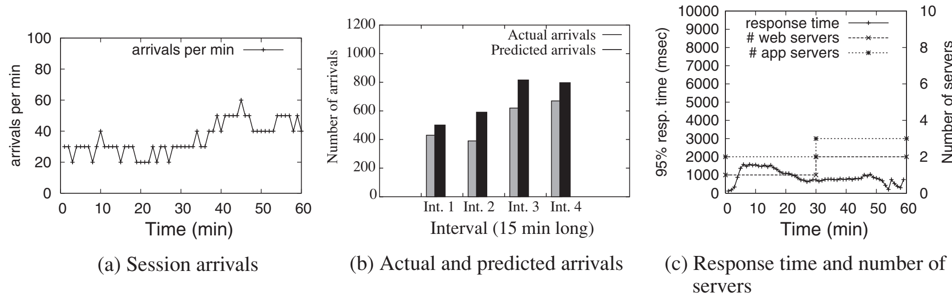
(c) Number of active sessions

(d) 95th response time

Fig. 11. Model-based multitier provisioning.

once every few minutes. For the sake of convenience, in the rest of the section, we will simply refer to these traces by the day from which they were constructed (even though they are only 1-hour long). We present three of these traces: (1) Figure 12(a) shows the workload for day 6 (a typical day), (2) Figure 13(a) shows the workload for day 7, (moderate overload), and (3) Figure 15(a) shows the workload for day 8 (extreme overload). Throughout this section, we will assume that the database tier has sufficient capacity to handle the peak observed on day 8 and does not become a bottleneck. The average session duration in our trace was 5 minutes.

In the rest of this section, we first experimentally evaluate our predictive and reactive provisioning mechanisms in isolation. Using the observations from these experiments we make a case for integrating these two mechanisms for effective handling of the workloads seen by Internet applications. We then experimentally evaluate the efficacy of this integrated provisioning approach.



(a) Session arrivals (b) Actual and predicted arrivals (c) Response time and number of servers

Fig. 12. Provisioning on day 6—typical day.

8.3.1 *Only Predictive Provisioning.* We first evaluate our predictive provisioning mechanism. Figure 12 presents the performance of the system during day 6 with the control plane employing only predictive provisioning (with reactive provisioning disabled). Day 6 was a typical day, meaning the workload closely resembled that observed during the previous days. The prediction algorithm was successful in exploiting this and was able to assign sufficient capacity to the application at all times. In Figure 12(b) we observe that the predicted arrivals closely matched the actual arrivals. The control plane adds servers at $t = 30$ minutes. This was well in time for the increased workload during the second half of the experiment. The application experiences satisfactory response time throughout the experiment (Figure 12(c)).

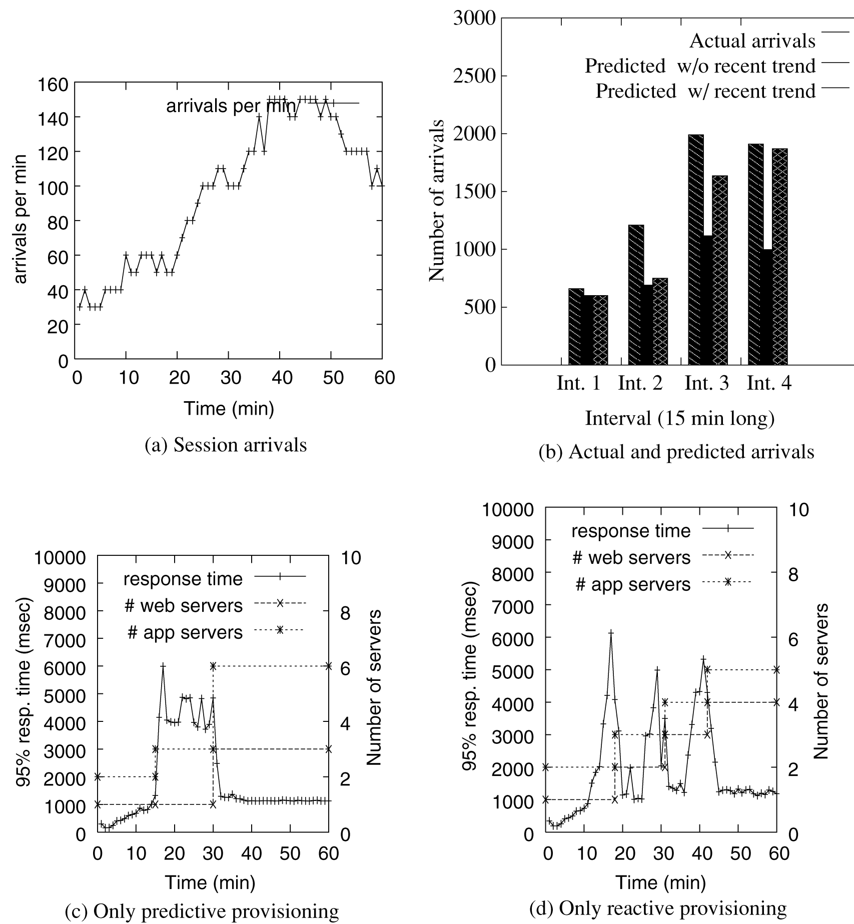
Result: Our predictive provisioning works well on typical days.

8.3.2 *Only Reactive Provisioning.* In Section 6.2 we discussed the potential problems with a purely reactive provisioning mechanism. Our next experiment demonstrates a shortcoming of such a provisioning approach.

In Figure 13 we present the results for day 7. Comparing this workload with that on day 6, we find that the application experienced a moderate overload on day 7, with the arrival rate going up to about 150 sessions/minute, more than twice the peak on day 6. The workload showed a monotonically increasing trend for the first 40 minutes.

We first let the control plane employ only predictive provisioning. Figure 13(b) shows the performance of our prediction algorithm, both with and without using recent trends, to correct the prediction. We find that the prediction algorithm severely underestimated the number of arrivals in Interval 2. The use of recent trends allowed it to progressively improve its estimate in Intervals 3 and 4 (predicted arrivals were nearly 80% of the actual arrivals in Interval 3 and almost equal in Interval 4). In Figure 13(c) we observe that the response time target was violated in Interval 2 due to under-allocation of servers.

Next, we repeat the experiment with the control plane, using only reactive provisioning. Figure 13(d) presents the application performance. Consider Interval 2 first. We observe that, unlike predictive provisioning, the reactive mechanism was able to pull additional servers at $t = 15$ minutes in response to the increased arrival rate, thus bringing down the response time within target.



(a) Session arrivals
(b) Actual and predicted arrivals
(c) Only predictive provisioning
(d) Only reactive provisioning

Fig. 13. Provisioning on day 7—moderate overload.

However, as the experiment progresses, the server allocation lags behind the continuously increasing workload. Since reactive provisioning only responds to very recent workload trends, it does not anticipate future requirements well and takes multiple allocation steps to add sufficient capacity. Meanwhile, the application experiences repeated violations of SLA during Intervals 2 and 3.

Additionally, as pointed out in Section 6, the presence of residual sessions may increase the effective time to switch a server from one application to another. Therefore, despite the use of VMM-based switching, there may be a considerable delay before additional servers become available to an application experiencing increased workload.

Finally, we present an experiment to illustrate another shortcoming of a purely reactive provisioning scheme. We present the time needed to activate a dormant replica in the application tier of Rubis under different degrees of network utilization. The Tomcat replica was hosted in a VM with a 1GB memory allocation. The image for the VM was stored on an NFS server connected

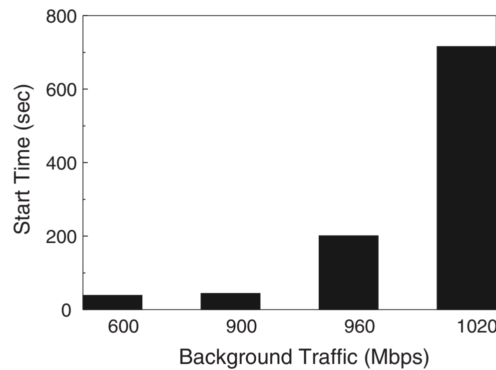


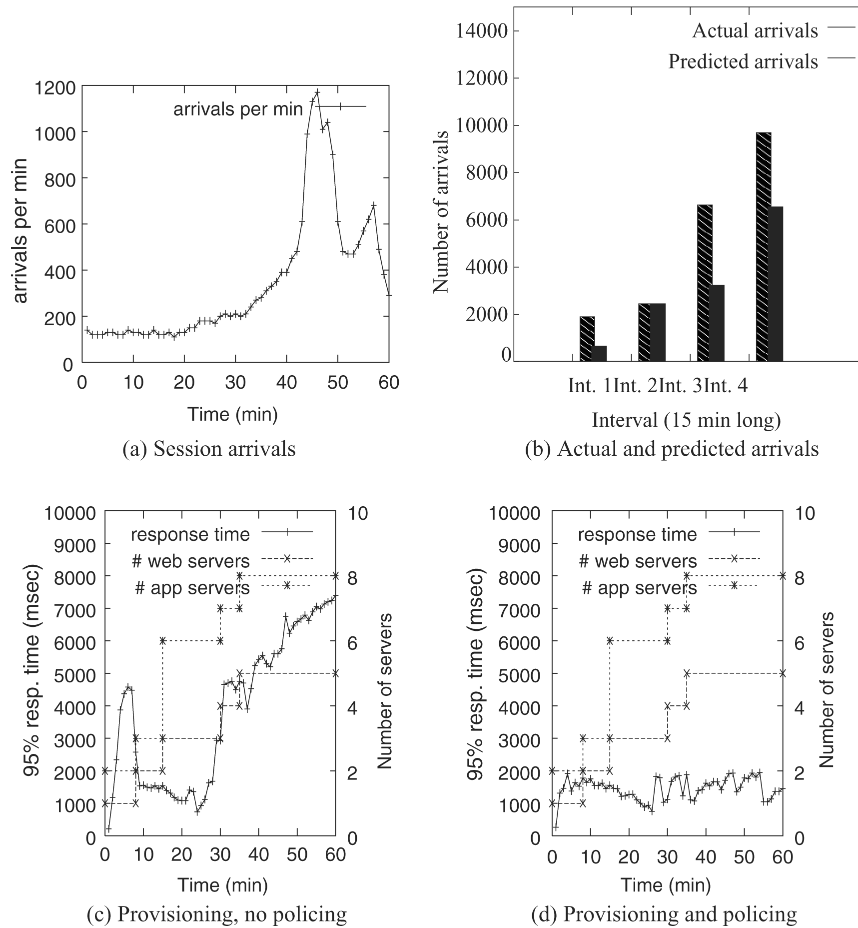
Fig. 14. Rubis: Time to activate a Tomcat replica under varying network traffic conditions.

by a 1 Gbps link and the VM was restored from a saved state. The background network traffic was created by 6 machines sending iperf streams of UDP traffic at a constant rate to the NFS server. For each level of traffic, we restored the VM eight times and present the average of these in Figure 14. Comparing the points with the least and the most network traffic, we observe the time to activate the replica went up by a factor of 20. Whereas, with a background traffic of 600 Mbps, the replica was ready in only 39 seconds, with a traffic of 1 Gbps, it took more than 11 minutes. It should be clear that a predictive mechanism that can proactively determine the right occasions for conducting such reactive provisioning can result in significant improvement in the agility of provisioning, and thereby improve the performance provided to the application.

Result: We need reactive mechanisms to deal with large flash crowds. However, the actions taken by reactive provisioning may lag the workload. Furthermore, the presence of residual sessions may render any benefits offered by VMM-based switching futile. Therefore, reactive provisioning alone may not be effective.

8.3.3 *Integrated Provisioning and Policing.* We used the workload on day 8 where the application experienced an extremely large overload (Figure 15(a)). The peak workload on this day was an order of magnitude (about 20 times) higher than on a typical day. Figure 15(b) shows how the prediction algorithm performs during this overload. The algorithm fails to predict the sharp increase in the workload during Interval 1. In Interval 2 it corrects its estimate based on the observed workload during Interval 1. The workload increases drastically (reaching up to 1200 sessions/second) during Intervals 3 and 4, and the algorithm fails to predict it.

In Figure 15(c) we show the performance of Rubis when the control plane employs both predictive and reactive mechanisms and session policing is disabled. In Interval 1, the reactive mechanism successfully adds additional capacity (at $t = 8$ minutes) to lower the response time. It is invoked again at $t = 34$ minutes (Observe that predictive provisioning is operating in concert with reactive provisioning; it results in the server allocations at $t = 15, 30, 45$



(a) Session arrivals

(b) Actual and predicted arrivals

(c) Provisioning, no policing

(d) Provisioning and policing

Fig. 15. Provisioning on day 8—extreme overload.

minutes). However, by this time (and for the remainder of the experiment) the workload is simply too high to be serviced by the servers available. We impose a resource limit of 13 servers for illustrative purposes. Beyond this, excess sessions must be turned away to continue meeting the SLA for admitted sessions. The lack of session policing causes response times to degrade during Intervals 3 and 4.

Next, we repeat this experiment with the session policing enabled. The performance of Rubis is shown in Figure 15(d). The behavior of our provisioning mechanisms is exactly as before. However, by turning away excess sessions, the sentry is able to maintain the SLA throughout.

Result: Predictive and reactive mechanisms, and policing, are all integral components of an effective provisioning technique. Our data center integrates all of these, enabling it to handle diverse workloads.

Table IV. Performance of VM-Based Switching; "n/a" Stands for "Not Applicable"

Scenario	Switching Time	r.t. During Switching
1	10 ± 1 sec	n/a
2	0	n/a
3	17 ± 2 min	n/a
4	< 1 sec	2400 ± 200
5	< 1 sec	950 ± 100

8.4 VM-Based Switching of Server Resources

We present measurements on our testbed to demonstrate the benefits that our VM-based switching can provide. We switch a server from a Tomcat capsule of Rubis to a Tomcat capsule of Rubbos. We compare five different ways of switching a server to illustrate the salient features of our scheme:

Scenario 1: New server taken from the free pool of servers; capsule and nucleus have to be started on the server.
Scenario 2: New server taken from the free pool of servers; capsule already running on a VM.
Scenario 3: New server taken from another application with residual sessions; we wait for all residual sessions to finish.
Scenario 4: New server taken from another application with residual sessions; we let the two VMs share the CPU equally while the residual sessions still exist.
Scenario 5: New server taken from another application with residual sessions; we change the CPU shares of the involved VMs using the fixed rate ramp-down strategy of Section 6.

Table IV presents the switching time and performance of residual sessions of Rubis in each of these scenarios. Comparing scenarios 2 and 3, we find that in our VM-based scheme, the time to switch a server is solely dependent on the residual sessions—the residual sessions of Rubis took about 17 minutes to finish, resulting in the large switching time in scenario 3. Scenarios 4 and 5 show that by letting the two VMs coexist while the residual sessions finish, we can eliminate this switching time. However, it is essential to continue providing sufficient capacity to the residual sessions during the switching period to ensure good performance—in scenario 4, new Rubbos sessions deprived the residual sessions of Rubis of the capacity they needed, thus degrading their response time.
Result: Use of virtual machines can enable agile switching of servers. Our adaptive techniques reduce the delays in switching caused by residual sessions.

8.5 System Overheads

Two sources of overhead in the proposed system are the virtual machines that run on the Elf nodes and the nuclei that run on all nodes. Measurements on our prototype indicate that the CPU overhead and network traffic caused by the nuclei are negligible. The control plane runs on a dedicated node and its scalability is not a cause of concern. We chose the Xen VMM to implement

our switching scheme since the performance of Xen/Linux has been shown to be consistently close to native Linux [Barham et al. 2003]. Further, Xen has been shown to provide good performance isolation when running multiple VMs simultaneously, and is capable of scaling to 128 concurrent VMs.

9. RELATED WORK

Previous literature on issues related to managing resources in platforms hosting Internet services spans several areas. In this section we describe the important pieces of work on these topics.

Dynamic provisioning and managing resources in clusters: The work on dynamic provisioning of a platform's resources may be classified into two categories. Some papers have addressed the problem of provisioning resources at the granularity of individual servers, as in our work. Ranjan et al. [2002] consider the problem of dynamically varying the number of servers assigned to a single service hosted on a data center. Their objective is to minimize the number of servers needed to meet the service's QoS targets. The algorithm is based on a simple scheme to extrapolate the current size of the server-set, based on observations of utilization levels and workloads to determine the server-set of the right size, and is evaluated via simulations. The Oceano project at IBM [Appleby et al. 2001] has developed a server farm in which servers can be moved dynamically across hosted applications depending on their changing needs. The main focus of this paper was on the implementation issues involved in building such a platform rather than the exact algorithms for provisioning.

Other papers have considered the provisioning of resources at finer granularity of resources. Muse [Chase and Doyle 2001] presents an architecture for resource management in a hosting center. Muse employs an economic model for dynamic provisioning of resources to multiple applications. In the model, each application has a utility function that is a function of its throughput and reflects the revenue generated by the application. There is also a penalty that the application charges the system when its goals are not met. The system computes resource allocations by attempting to maximize the overall profit. *Cluster Reserves* [Aron et al. 2000] investigated resource allocation in server clusters. The work assumes a large application running on a cluster, where the aim is to provide differentiated service to clients based on some notion of service class. This is achieved by making the OS schedulers provide fixed resource shares to applications spanning multiple nodes. The *Cluster-On Demand (COD)* [Chase et al. 2003] work presents an automated framework to manage resources in a shared hosting platform. COD introduces the notion of a *virtual cluster*, which is a functionally isolated group of hosts within a shared hardware base. A key element of COD is a protocol to resize virtual clusters dynamically in cooperation with pluggable middleware components. Chandra et al. [2003] model a server resource that services multiple applications as a GPS system and presents online workload prediction and optimization-based techniques for dynamic resource allocation. Some of the coauthors address the problem of providing resource guarantees to distributed applications running on a shared hosting platform [Urgaonkar and Shenoy 2004b]. In another paper, some of

the coauthors propose a resource overbooking based scheme for maximizing revenue in a shared platform [Urgaonkar et al. 2002].

An alternate approach for improving performance of overloaded Web servers is based on redesigning the scheduling policy employed by the servers. Schroeder and Harchol-Balter [2003] propose employing the SRPT algorithm based on scheduling the connection with the shortest remaining time, and demonstrate that it leads to improved average response time. While scheduling can improve response times, under extreme overloads admission control, and the ability to add extra capacity are indispensable. Better scheduling algorithms are complementary to our solutions for handling overloads.

Modeling of Internet Applications: Modeling of single-tier Internet applications, of which HTTP servers are the most common example, has been studied extensively. A queuing model of a Web server serving static content was proposed in Slothouber [1996]. The model employs a network of four queues—two modeling the Web server itself, and the other two modeling the Internet communication network. A queuing model for performance prediction of single-tier Web servers with static content was proposed in Doyle et al. [2003]. This approach, (1) explicitly models CPU, memory, and disk bandwidth, in the Web server, (2) utilizes knowledge of file size and popularity distributions, and (3) relates average response time to available resources. A GPS-based queuing model of a single resource, such as the CPU, at a Web server was proposed in Chandra et al. [2003]. The model is parameterized by online measurements and is used to determine the resource allocation needed to meet desired average response time targets. A G/G/1 queuing model for replicated single-tier applications (e.g., clustered Web servers) was proposed in Urgaonkar and Shenoy [2004a]. The architecture and prototype implementation of a performance management system for cluster-based Web services was proposed in Levy et al. [2003]. The work employs an M/M/1 queuing model to compute responses times of Web requests. A model of a Web server for the purpose of performance control using classical feedback control theory was studied in Abdelzaher et al. [2002]; an implementation and evaluation using the Apache Web server was also presented in the work. A combination of a Markov chain model and a queuing network model to capture the operation of a Web server was presented in Menasce [2003]—the former model represents the software architecture employed by the Web server (e.g., process-based versus thread-based) while the latter computes the Web server's throughput.

Since these efforts focus primarily on single-tier Web servers, they are not directly applicable to applications employing multiple tiers, or to components such as Java enterprise servers or database servers employed by multitier applications. Further, many of these efforts assume static Web content, while multi-tier applications, by their very nature, serve dynamic Web content.

A few recent efforts have focused on the modeling of multi-tier applications. However, many of these either make simplifying assumptions or are based on simple extensions of single-tier models. A number of papers have taken the approach of modeling only the most constrained or the most bottlenecked tier of the application. For instance, Villela et al. [2004] considers the problem of provisioning servers for only the Java application tier; it uses an M/G/1/PS model

for each server in this tier. Similarly, the Java application tier of an e-commerce application with N servers is modeled as a G/G/N queuing system in Ranjan et al. [2002]. Other efforts have modeled the entire multi-tier application using a single queue—an example is Kamra et al. [2004], which uses an M/GI/1/PS model for an e-commerce application. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multi-tier application will fail to capture caching effects at other tiers. Such a model cannot be used for capacity provisioning of other tiers. Finally, as we show in our experiments, system bottlenecks can shift from one tier to another with changes in workload characteristics. Under these scenarios, there is no single tier that is the most constrained. In this article, we present a model of a multitier application that overcomes these drawbacks. Our model explicitly accounts for the presence of all tiers and also captures application artifacts such as session-based workloads, tier replication, load imbalances, caching effects, and concurrency limits.

Some researchers have developed sophisticated queueing models capable of capturing the simultaneous resource demands and parallel subpaths that occur within a tier of a multitier application. An important example of such models is Layered Queueing Networks (LQN). LQNs are an adaptation of the Extended Queueing Network, defined specifically to represent the fact that software servers are executed on top of other layers of servers and processors, giving complex combinations of simultaneous requests for resources [Rolia and Sevcik 1995; Woodside and Raghunath 1995; Liu et al. 2001; Xu et al. 2006; and Franks 1999]. The focus of most of these papers is on an Enterprise Java Beans based application tier whereas the work reported in this article is concerned with a model for an entire multitier application. While one possible approach to modeling multi-tier applications could be based on the use of these existing per-tier models as building blocks, we do not pursue that direction in this article.

The research efforts on modeling of most interest to our work are papers by Kounev and Buchmann [2003], Bennani and Menasce [2005], and a paper by some of our co-authors Urgaonkar et al. [2005], all of which develop sophisticated queuing models based on networks of queues to capture multi-tier applications. The authors employ an approximate mean-value analysis algorithm to develop an online provisioning technique using this model. We believe that the simpler model used in the research reported in this article can be replaced by these models to obtain more accurate predictive provisioning decisions.

Work by Cohen et al. [2004] uses a probabilistic modeling approach called Tree-Augmented Bayesian Networks (TANs) to identify combinations of system-level metrics and threshold values that correlate with high-level performance states—compliance with service-level agreements for average response time—in a three-tier Web service under a variety of conditions. Experiments based on real applications and workloads indicate that this model is a suitable candidate for use in offline fault diagnosis and online performance prediction. Whereas it would be a useful exercise to compare such a learning-based modeling approach with our queuing-theory-based model, it is

beyond the scope of this article. In the absence of such a comparative study and given the widely different natures of these two modeling approaches, we do not make any assertions about the pros and cons of our model over the TAN-based model.

SLAs and Adaptive QoS Degradation: The WSLA project at IBM [WSLA http://www.research.ibm.com/wsla] addresses service level management issues and challenges in designing an unambiguous and clear specification of SLAs that can be monitored by the service provider, customer, and even by a third party. Abdelzaher and Bhatti [1999] propose dealing with dynamically changing workloads by adapting delivered content to load conditions.

Admission Control for Internet Services: Many papers have developed overload management solutions based on doing admission control. Several admission controllers operate by controlling the rate of admission but without distinguishing requests based on their sizes. Voigt et al. [2001] present kernel-based admission control mechanisms to protect web servers against overloads—*SYN policing* controls the rate and burst at which new connections are accepted, *prioritized listen queue* reorders the listen queue based on predefined connection priorities, *HTTP header-based control* enables rate policing based on URL names. Welsh and Culler [2003] propose an overload management solution for Internet services built using the SEDA architecture. A salient feature of their solution is feedback-based admission controllers embedded into individual stages of the service. The admission controllers work by gradually increasing admission rate when performance is satisfactory and decreasing it multiplicatively upon observing QoS violations. The QGuard system [Jamjoom et al. 2000] proposes an adaptive mechanism that exploits inbound rate controls to fend off overload and provide QoS differentiation among traffic classes. The determination of these rate limits, however, is not dynamic but is delegated to the administrator. Iyer et al. [2000] propose a system based on two mechanisms: using thresholds on the connection queue length to decide when to start dropping new connection requests and sending feedback to the proxy during overloads that would cause it to restrict the traffic being forwarded to the server. However, they do not address how these thresholds may be determined online. Cherkasova and Phaal [1999] propose an admission control scheme that works at the granularity of sessions rather than individual requests, and evaluate it using a simple simulation study. This was based on a simple model to characterize sessions. The admission controller was based on rejecting all sessions for a small duration if the server utilization exceeded a prespecified threshold, and has some similarity to our approximate admission control, except we use information about the sizes of requests in various classes to determine the drop threshold.

Several efforts have proposed solutions based on analytical characterization of the workloads of Internet services and modeling of the servers. Kanodia and Knightly [2000] utilize a modeling technique called *service envelops* to devise an admission control for Web services that attempts to set different response time targets for multiple classes of requests. Li and Jamin [2000] present a measurement-based admission control to distribute bandwidth across clients of unequal requirements. A key distinguishing feature of their algorithm is the

introduction of controlled amounts of delay in the processing of certain requests during overloads, to ensure different classes of requests are receiving the appropriate share of the bandwidth. Knightly and Shroff [1999] describe and classify a broad class of admission control algorithms and evaluate the accuracy of these algorithms via experiments. They identify key aspects of admission control that enable it to achieve high statistical multiplexing gains.

Two admission control algorithms have been proposed recently that utilize measurements of request sizes to guide their decision making. Verma and Ghosal [2003] propose a service-time-based admission control that uses predictions of arrivals and service times in the short-term future to admit a subset of requests that would maximize the profit of the service provider. Elnikety et al. [2004] present an admission control for multitier e-commerce sites that externally observes execution costs of requests, distinguishing different request types. Our measurement-based admission control is based on similar ideas, although the techniques differ in the details.

10. CONCLUSIONS

In this article, we argued that dynamic provisioning of multi-tier Internet applications raises new challenges not addressed by prior work on provisioning single-tier applications. We proposed a novel dynamic provisioning technique for multitier Internet applications that employs (1) a flexible queuing model to determine how much resources to allocate to each tier of the application, and (2) a combination of predictive and reactive methods that determine when to provision these resources, both at at large and small time scales. Our experiments on a forty machine Xen/Linux-based hosting platform demonstrate the responsiveness of our technique in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes while maintaining response time targets. Our technique also reduced the overhead of switching servers across applications from several minutes or more, to less than a second, while meeting the performance targets of residual sessions.

REFERENCES

ABDELZAHER, T. AND BHATTI, N. 1999. Web content adaptation to improve server overload behavior. In *Proceedings of the World Wide Web Conference (WWW8)*. Tornoto.

ABDELZAHER, T., SHIN, K. G., AND BHATTI, N. 2002. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Trans. Para. Distrib. Syst. 13,* 1 (Jan.).

AMAZON. 2000. The Holiday Shopping Season, So Far. `http://www.fool.com/news/2000/wmt001127.htm`.

APPLEBY, K., FAKHOURI, S., FONG, L., GOLDZMIDT, M. K. G., KRISHNAKUMAR, S., PAZEL, D., PERSHING, J., AND ROCHWERGER, B. 2001. Oceano—SLA-based management of a computing utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*.

ARLITT, M. AND JIN, T. 1999. Workload characterization of the 1998 World Cup Web site. Tech. Rep. HPL-1999-35R1, HP Labs.

ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. 2000. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference*. Santa Clara, CA.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtulization. In *Proceedings of the Nineteenth Symposium on Operating Systems Principles (SOSP)*.

BENANI, M. AND MENASCE, D. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the IEEE International Conference on Autonomic Computing, Seattle (ICAC-05)*. WA.

CHANDRA, A., GONG, W., AND SHENOY, P. 2003. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the Eleventh International Workshop on Quality of Service (IWQoS 2003)*. Monterey, CA.

CHASE, J. AND DOYLE, R. 2001. Balance of power: Energy management for server clusters. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Elmau, Germany.

CHASE, J., GRIT, L., IRWIN, D., MOORE, J., AND SPRENKLE, S. 2003. Dynamic virtual clusters in a grid site manager. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*.

CHERKASOVA, L. AND PHAAL, P. 1999. Session-based admission control: A mechanism for improving performance of commercial Web sites. In *Proceedings of the Seventh International Workshop on Quality of Service, IEEE/IFIP Event*. London.

COHEN, I., CHASE, J., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the Sixth USENIX Symposium in Operating Systems Design and Implementation (OSDI 2004)*. San Francisco, CA.

DOYLE, R., CHASE, J., ASAD, O., JIN, W., AND VAHDAT, A. 2003. Model-based resource provisioning in a Web service utility. In *Proceedings of the Fourth USITS*.

DYNASERVER. 2005. Dynaserver project. `http://compsci.rice.edu/CS/Systems/DynaServer/`.

ELNIKETY, S., NAHUM, E., TRACEY, J., AND ZWAENEPOEL, W. 2004. A method for transparent admission control and request scheduling in e-commerce Web sites. In *Proceedings of the Thirteenth International Conference on the World Wide Web*. New York, NY. 276–286.

FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles (SOSP'97)*.

FRANKS, R. G. 1999. Performance analysis of distributed server systems. Ph.D. Dissertation, Carleton University.

GOLDBERG, R. 1974. Survey of virtual machine research. *IEEE Comput.* 34–45.

HELLERSTEIN, J., ZHANG, F., AND SHAHABUDDIN, P. 1999. An approach to predictive detection for service management. In *Proceedings of the IEEE International Conference on Systems and Network Management*.

IYER, R., TEWARI, V., AND KANT, K. 2000. Overload control mechanisms for Web servers. In *Workshop on Performance and QoS of Next Generation Networks*.

JAMJOOM, H., REUMANN, J., AND SHIN, K. 2000. QGuard: Protecting Internet servers from overload. Tech. rep., CSE-TR-427-00, Department of Computer Science, University of Michigan.

KAMRA, A., MISRA, V., AND NAHUM, E. 2004. Yaksha: A controller for managing the performance of 3-tiered Websites. In *Proceedings of the Twelfth IWQoS*.

KANODIA, V. AND KNIGHTLY, E. 2000. Multi-class latency-bounded Web servers. In *Proceedings of International Workshop on Quality of Service (IWQoS'00)*.

KLEINROCK, L. 1976. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc.

KOUNEV, S. AND BUCHMANN, A. 2003. Performance modeling and evaluation of large-scale J2EE applications. In *Proceedings of the Computer Measurement Group's 2003 International Conference (CMG 2003)*. Dallas, TX.

KNIGHTLY, E. AND SHROFF, N. 1999. Admission control for statistical QoS: Theory and practice. In *IEEE Netw. 13*, 2, 20–29.

KTCPVS. 2005. Kernel TCP Virtual Server. `http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html`.

LEVY, R., NAGARAJARAO, J., PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. 2003. Performance management for cluster based Web services. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management*. vol. 246, 247–261.

LI, S. AND JAMIN, S. 2000. A Measurement-based admission-controlled Web server. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*. Tel Aviv, Israel.

LIU, T.-K., KUMARAN, S., AND LUO, Z. 2001. Layered queueing models for enterprise Java Beans applications. Tech. rep., IBM. (June).

MENASCE, D. 2003. Web server software architectures. In *IEEE Internet Comput.* vol. 7.

ORACLE9I. 2005. Oracle9i. http://www.oracle.com/technology/products/oracle9i.

PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWANEPOEL, W., AND NAHUM, E. 1998. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose, CA.

RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. 2002. QoS-driven server migration for Internet data centers. In *Proceedings of the Tenth International Workshop on Quality of Service*. Miami, FL.

ROLIA, J. AND SEVCIK, K. 1995. The method of layers. *IEEE Trans. Softw. Eng. 21,* 8, 689–700.

ROLIA, J., ZHU, X., ARLITT, M., AND ANDRZEJAK, A. 2002. Statistical service assurances for applications in utility grid environments. Tech. rep. HPL-2002-155, HP Labs.

SAITO, Y., BERSHAD, B., AND LEVY, H. 1999. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles (SOSP'99)*.

SAR. 2005. Sysstat Package. http://freshmeat.net/projects/sysstat.

SCHROEDER, B. AND HARCHOL-BALTER, M. 2003. Web servers under overload: How scheduling can help. In *Proceedings of the Eighteenth International Teletraffic Congress*.

SHEN, K., TANG, H., YANG, T., AND CHU, L. 2002. Integrated resource management for cluster-based Internet services. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA.

SLOTHOUBER, L. 1996. A model of Web server performance. In *Proceedings of the Fifth International World Wide Web Conference*.

URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. 2005. An analytical model for multi-tier Internet services and its applications. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2005)*. Banff, Canada.

URGAONKAR, B. AND SHENOY, P. 2004a. Cataclysm: Handling extreme overloads in Internet services. In *Proceedings of the Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004)*. St. John's, Newfoundland, Canada.

URGAONKAR, B. AND SHENOY, P. 2004b. Sharc: Managing CPU and network bandwidth in shared clusters. In *IEEE Trans. Para. Distrib. Syst. 15*, 1, 2–17.

URGAONKAR, B., SHENOY, P., AND ROSCOE, T. 2002. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*. Boston, MA.

VERMA, A. AND GHOSAL, S. 2003. On admission control for profit maximization of networked service providers. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*. Budapest, Hungary.

VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. 2004. Provisioning servers in the application tier for e-commerce systems. In *Proceedings of the Twelfth IWQoS*.

VOIGT, T., TEWARI, R., FREIMUTH, D., AND MEHRA, A. 2001. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of USENIX Annual Technical Conference*.

WELSH, M. AND CULLER, D. 2003. Adaptive overload control for busy Internet servers. In *Proceedings of the Fourth USENIX Conference on Internet Technologies and Systems (USITS'03)*.

WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*.

WOODSIDE, C. AND RAGHUNATH, G. 1995. General bypass architecture for high-performance distributed algorithms. In *Proceedings of the Sixth IFIP Conference on Performance of Computer Networks*. Istanbul, Turkey.

WSLA. http://www.research.ibm.com/wsla. Web service level agreements (wsla) project.

XU, J., OUFIMTSEV, A., WOODSIDE, M., AND MURPHY, L. 2006. Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes 31,* 2.

Received October 2005; revised April 2007; accepted December 2007