Analytic Modeling of Multitier Internet Applications

BHUVAN URGAONKAR
The Penn State University
GIOVANNI PACIFICI
IBM T. J. Watson Research Center
PRASHANT SHENOY
University of Massachusetts
and
MIKE SPREITZER and ASSER TANTAWI
IBM T. J. Watson Research Center

Since many Internet applications employ a multitier architecture, in this article, we focus on the problem of analytically modeling the behavior of such applications. We present a model based on a network of queues where the queues represent different tiers of the application. Our model is sufficiently general to capture (i) the behavior of tiers with significantly different performance characteristics and (ii) application idiosyncrasies such as session-based workloads, tier replication, load imbalances across replicas, and caching at intermediate tiers. We validate our model using real multitier applications running on a Linux server cluster. Our experiments indicate that our model faithfully captures the performance of these applications for a number of workloads and configurations. Furthermore, our model successfully handles a comprehensive range of resource utilization—from 0 to near saturation for the CPU—for two separate tiers. For a variety of scenarios, including those with caching at one of the application tiers, the average response times predicted by our model were within the 95% confidence intervals of the observed average response times. Our experiments also demonstrate the utility of the model for dynamic capacity provisioning, performance prediction, bottleneck identification, and session policing. In one scenario, where the request arrival rate increased from less than 1500 to nearly 4200 requests/minute, a dynamic provisioning technique employing our model was able to maintain response time targets by increasing the capacity of two of the tiers by factors of 2 and 3.5, respectively.

A preliminary version of this paper appeared in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS'05*), Banff, Canada.
Authors' addresses: B. Urgaonkar, Department of CSE, The Penn State University, University Park, PA 16802; email: bhuvan@cse.psu.edu; G. Pacifici, M. Spreitzer, A. Tantawi, Service Management Middleware Department, IBM T. J. Watson Research Center, Hawthorne, NY 10532; email: {giovanni,mspreitz,tantawi}@us.ibm.com; P. Shenoy, Department of Computer Science, University of Massachusetts, Amherst, MA 01003; email: shenoy@cs.umass.edu.
Permission to make digital or hard copies 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 the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1559-1131/2007/05-ART2 $5.00 DOI 10.1145/1232722.1232724 http://doi.acm.org/10.1145/1232722.1232724

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

General Terms: Design, Experimentation, Measurement

Additional Key Words and Phrases: Internet service, analytical model, queuing theory, mean-value analysis, hosting platform, tier, dynamic provisioning, performance prediction, session, policing

ACM Reference Format:

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. 2007. Analytic modeling of multitier internet applications. ACM Trans. Web. 1, 1, Article 2 (May 2007), 35 pages. DOI = 10.1145/1232722.1232724 http://doi.acm.org/10.1145/1232722.1232724

1. INTRODUCTION

1.1 Motivation

Internet applications such as online news, retail, and financial sites have become commonplace in recent years. Modern Internet applications are complex software systems that employ a multitier architecture and are replicated or distributed on a cluster of servers. Each tier provides a certain functionality to its preceding tier and makes use of the functionality provided by its successor to carry out its part of the overall request processing. For instance, a typical e-commerce application consists of three tiers—a frontend Web tier that is responsible for HTTP processing, a middle-tier Java enterprise server that implements core application functionality, and a backend database that stores product catalogs and user orders. In this example, incoming requests undergo HTTP processing, processing by the Java application server, and trigger queries or transactions at the database. There are other technologies available for constructing the middle tier (such as PHP[1] and CGI[2]). A Java enterprise server, however, has increasingly become the common choice.

This article focuses on analytically modeling the behavior of multitier Internet applications. Such a model is important for the following reasons: (i) *capacity provisioning,* which enables a server farm to determine how much capacity to allocate to an application in order for it to service its peak workload; (ii) *performance prediction,* which enables the response time of the application to be determined for a given workload and a given hardware and software configuration, (iii) *application configuration,* which enables various configuration parameters of the application to be determined for a certain performance goal, (iv) *bottleneck identification and tuning,* which enables system bottlenecks to be identified for purposes of tuning, and (v) *session policing,* which enables the application to turn away excess sessions during transient overloads.

1.2 Shortcomings of Existing Modeling Approaches

Modeling of single-tier applications such as vanilla Web servers (e.g., Apache) is well studied [Doyle et al. 2003; Menasce 2003; Slothouber 1996]. In contrast, modeling of multitier applications is less well studied, even though this

[1]PHP: Hypertext Preprocessor. http://www.php.net/.
[2]CGI: Common Gateway Interface. http://www.w3.org/CGI/.

flexible architecture is widely used for constructing Internet applications and services.

Extending single-tier models to multitier scenarios is nontrivial for the following reasons. First, various application tiers such as Web, Java, and database servers have vastly different performance characteristics and collectively modeling their behavior is a difficult task. Further, in a multitier application, (i) some tiers may be replicated while others are not, (ii) the replicas may not be perfectly load balanced, and (iii) caching may be employed at intermediate tiers, all of which complicate the performance modeling.

A number of researchers 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 only for 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 multitier application using a single queue, an example, that uses a M/GI/1/PS model for an e-commerce application is Kamra et al. [2004]. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multitier application will fail to capture caching effects at other tiers. Such a model can not be used for capacity provisioning of other tiers. Finally, 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.

Some researchers have employed models based on networks of queues to explicitly capture the various tiers in a modern Internet application [Kounev and Buchmann 2003; Benani and Menasce 2005; Menasce et al. 2004]. We consider these pieces of research as complementary to our work. Several contributions made by our model, however, set our work apart. Specifically, we develop our model to capture several features of Internet services such as concurrency limits and caching at certain tiers. To the best of our knowledge, these features were not addressed by any previous modeling work. As we shall see, capturing these characteristics of modern applications is important due to their impact on the performance of these applications and their clients. Additionally, while these other models have only been validated using simulations with online and batch workloads, we validate our model using realistic Internet services on a prototype Linux cluster.

Finally, modern Internet workloads are session-based where each session comprises a sequence of requests with think-times in between. For instance, a session at an online retailer may comprise the sequence of user requests to browse the product catalog and to make a purchase. Sessions are stateful from the perspective of the application, an aspect that must be incorporated into the model. The design of an analytical model that can capture the impact of these factors is the focus of this article.

1.3 Research Contributions

This article presents a model of a multitier Internet application based on a network of queues where the queues represent different tiers of the application.

Our model can handle applications with an arbitrary number of tiers and those with significantly different performance characteristics. A key contribution of our work is that the complex task of modeling a multitier application is reduced to the modeling of request processing at individual tiers and the flow of requests across tiers. Our model is inherently designed to handle session-based workloads and can account for application idiosyncrasies such as replication at tiers, load imbalances across replicas, caching effects, and concurrency limits at each tier.

We validate the model using two open-source multitier applications running on a Linux-based server cluster. We demonstrate the ability of our model to accurately capture the effects of a number of commonly used techniques such as query caching at the database tier and class-based service differentiation. For a variety of scenarios, including an online auction application that employs query caching at its database tier, the average response times predicted by our model were within the 95% confidence intervals of the observed average response times. We conduct a detailed experimental study using our prototype to demonstrate the utility of our model for the purposes of dynamic provisioning, response time prediction, application configuration, and session policing. Our experiments demonstrate the ability of our model to correctly identify bottlenecks in the system and the shifting of bottlenecks due to variations in the Internet workload. In one scenario, where the arrival rate to an application increased from 1500 to nearly 4200 requests/minute, our model was able to continue meeting response time targets by successfully identifying the two bottleneck tiers and increasing their capacity by factors of 2 and 3.5, respectively.

The remainder of this article is structured as follows. Section 2 provides an overview of multitier applications and related work. We describe our model in Sections 3 and enhancements to it in Section 4. Sections 5 and 6 present experimental validation of the model and an illustration of its applications, respectively. Finally, Section 7 presents our conclusions.

2. BACKGROUND AND RELATED WORK

This section provides an overview of multitier applications and the underlying server platform assumed in our work. We also discuss related work in the area.

2.1 Internet Application Architecture

Modern Internet applications are designed using multiple tiers (the terms Internet application and service are used interchangeably in this article). A multitier architecture provides a flexible, modular approach for designing such applications. Each application tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. The various tiers participate in the processing of each incoming request during its lifetime in the system. Depending on the processing demand, a tier may be replicated using clustering techniques. In such an event, a dispatcher is used at each replicated tier to distribute requests among the replicas for the purpose of load balancing. Figure 1 depicts a three-tier application where the first two tiers are replicated, while the third one is

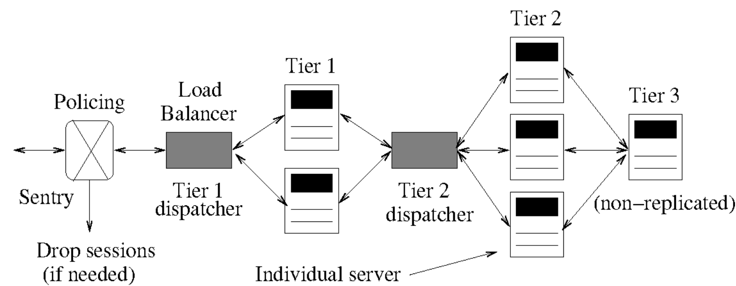


Fig. 1. A three-tier application.

not. Such an architecture is commonly employed by e-commerce applications where a clustered Web server and a clustered Java application server constitute the first two tiers, and the third tier consists of a nonreplicable database.[3]

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, which is often the case, successive requests will need to be serviced by the same server at each tier, and the dispatcher will need to account for this server state when redirecting requests.

As shown in Figure 1, each application employs a sentry that 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.

We assume that Internet applications typically run on a server cluster that is commonly referred to as a data center. A data center runs multiple third-party applications concurrently in return for revenue [Chase and Doyle 2001; Urgaonkar et al. 2002]. In this work, we assume that each tier of an application (or each replica of a tier) runs on a separate server. This is referred to as *dedicated hosting* where each application runs on a subset of the servers, and a server is allocated to at most one application tier at any given time. Unlike *shared hosting* where multiple small applications share each server, dedicated hosting is used for running large clustered applications where server sharing is infeasible due to the workload demand imposed on each individual application.

Given an Internet application, we assume that it specifies its desired performance requirement in the form of a service-level agreement (SLA). The SLA assumed in this work is a bound on the average response time that is acceptable to the application. For instance, the application SLA may specify that the average response time should not exceed one second regardless of the workload. We assume that averages are computed over intervals of length 30 minutes unless otherwise specified.

[3]Traditionally database servers have employed a shared-nothing architecture that does not support replication. However, certain new databases employ a shared-everything architecture (Oracle9i 2005 http://www.oracle.com/technology/production/oracle9i) that supports clustering and replication but with certain constraints.

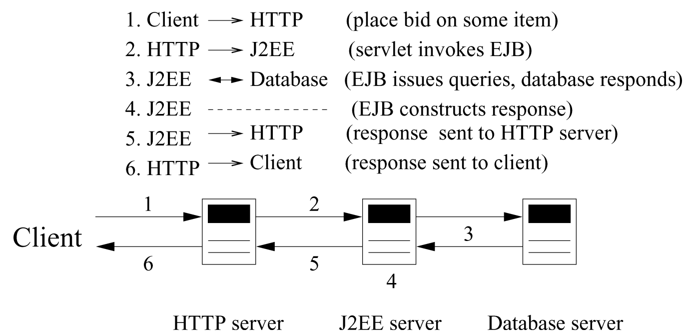


Fig. 2. Request processing in an online auction application.

2.2 Request Processing in Multitier Applications

Consider a multitier application consisting of M tiers denoted by T_1, T_2 through T_M. In the simplest case, each request is processed exactly once by tier T_i and then forwarded to tier T_{i+1} for further processing. Once the result is computed by the final tier T_M, it is sent back to T_{M-1}, which processes this result and sends it to T_{M-2}, and so on. Thus, the result is processed by each tier in the reverse order until it reaches T_1, which then sends it to the client. Figure 2 illustrates the steps involved in processing a bid request at a three-tier online auction site. The figure shows how the request trickles downstream and how the result propagates upstream through the various tiers.

More complex processing at the tiers is also possible. In such scenarios, each request can visit a tier multiple times. As an example, consider a keyword search at an online superstore, which triggers a query on the music catalog, a query on the book catalog, and so on. These queries can be issued to the database tier sequentially, where each query is issued after the result of the previous query has been received, or in parallel. Thus, in the general case, each request at tier T_i can trigger multiple requests to tier T_{i+1}. In the sequential case, each of these requests is issued to T_{i+1} once the result of the previous request has finished. In the parallel case, all requests are issued to T_{i+1} at once. In both cases, all results are merged and then sent back to the upstream tier T_{i-1}.

2.3 Related Work

Single-Tier 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 (i) explicitly models CPU, memory, and disk bandwidth in the Web server, (ii) utilizes knowledge of file size and popularity distributions, and (iii) 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) has been proposed in Urgaonkar and Shenoy [2004]. 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 the these efforts assume static Web content, while multitier applications, by their very nature, serve dynamic Web content.

Extensions Based on Single-Tier Models. A few recent efforts have focused on the modeling of multitier applications. However, many of these efforts 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 only for 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 multitier application using a single queue; an example, that uses a M/GI/1/PS model for an e-commerce application is Kamra et al. [2004]. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multitier application will fail to capture caching effects at other tiers. Such a model can not 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.

Models Based on Networks of Queues. 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 are 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; 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 multitier applications could be based on the use of these existing per-tier models as building blocks, we do not pursue this direction in this article.

The research efforts most similar to that reported in our work are papers by Kounev and Buchmann [2003] and Bennani and Menasce [2005]. Kounev and Buchmann use a model based on a network of queues for performance prediction of a 2-tier SPECjAppServer2002 application and solve this model numerically using publicly available analysis software. Bennani and Menasce model a multitier Internet service serving multiple types of transactions as a network of queues with customers belonging to multiple classes [Benani and Menasce 2005; Menasce et al. 2004]. The authors employ an approximate mean-value analysis algorithm to develop an online provisioning technique using this model. Whereas Bennani and Menasce focus on using their model for provisioning, our main focus is on capturing various features of Internet services such as concurrency limits and caching at certain tiers. Second, while their model is validated using simulations with online and batch workloads, we validate our model using realistic Internet services on a prototype Linux cluster.

Machine Learning Based Models. 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. While 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.

3. A MODEL FOR A MULTITIER INTERNET APPLICATION

In this section, we present a baseline queuing model for a multitier Internet application. In the next section, we present several enhancements to this baseline model to capture certain application idiosyncrasies.

3.1 The Basic Queuing Model

Consider an application with M tiers denoted by $T_1, \ldots, T_M$. Initially we assume that no tier is replicated; each tier is assumed to run on exactly one server, an assumption that is relaxed later.

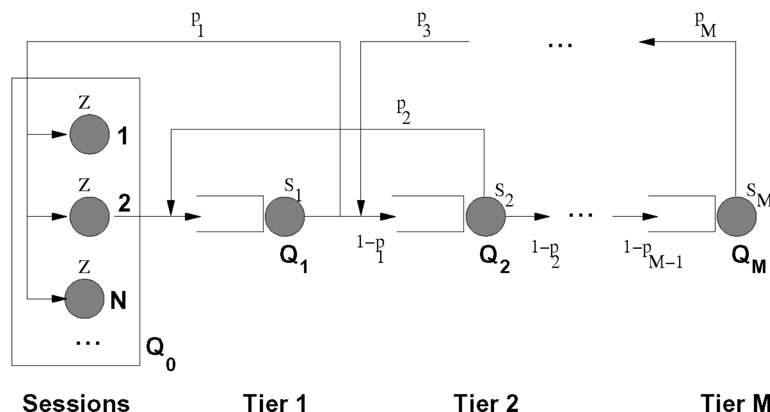


Fig. 3. Modeling a multitier application using a network of queues.

Modeling Multiple Tiers. We model the application using a network of of M queues, $\mathcal{Q}_1, \ldots, \mathcal{Q}_M$ (see Figure 3). Each queue represents an application tier and the underlying server that it runs on. We assume a processor sharing (PS) discipline at each queue since it closely approximates the scheduling policies employed by most commodity operating systems (e.g., Linux CPU time-sharing).

When a request arrives at tier T_i, it triggers one or more requests at its subsequent tier T_{i+1}; recall the example of a keyword search that triggers multiple queries at different product catalogs. In our queuing model, we can capture this phenomenon by allowing a request to make multiple visits to each of the queues during its overall execution. This is achieved by introducing a transition from each queue to its predecessor as shown in Figure 3. A request, after some processing at queue $\mathcal{Q}_i$, either returns to $\mathcal{Q}_{i-1}$ with a certain probability p_i or proceeds to $\mathcal{Q}_{i+1}$ with probability $(1 - p_i)$. The only exception is the last tier queue $\mathcal{Q}_M$, where all requests return to the previous queue, and the first queue $\mathcal{Q}_1$, where a transition to the preceding queue denotes request completion. As argued in Section 3.2, our model can handle multiple visits to a tier regardless of whether they occur sequentially or in parallel.

Observe that caching effects are naturally captured by this model. If caching is employed at tier T_i, a cache hit causes the request to immediately return to the previous queue $\mathcal{Q}_{i-1}$ without triggering any work in queues $\mathcal{Q}_{i+1}$ or later. Thus, the impact of cache hits and misses can be incorporated by appropriately determining the transition probability p_i and the service time of a request at $\mathcal{Q}_i$.

Modeling Sessions. Recall from Section 2 that Internet workloads are session-based. A session issues one or more requests during its lifetime, one after another, with think times in between (we refer to this duration as the *user think time*). Typical sessions in an Internet application may last several minutes. Thus, our model needs to capture the relatively long-lived nature of sessions as well as the response times of individual requests within a session.

We do so by augmenting our queuing network with a subsystem modeling the active sessions of the application. We model sessions using an *infinite server*

queuing system, $\mathcal{Q}_0$, that feeds our network of queues and forms the closed-queuing system shown in Figure 3. The servers in $\mathcal{Q}_0$ capture the session-based nature of the workload as follows. Each active session is assumed to occupy one server in $\mathcal{Q}_0$. As shown in Figure 3, a request issued by a session emanates from a server in $\mathcal{Q}_0$ and enters the application at $\mathcal{Q}_1$. It then moves through the queues $\mathcal{Q}_1, \ldots, \mathcal{Q}_M$, possibly visiting some queues multiple times (as captured by the transitions from each tier to its preceding tier) and getting processed at the visited queues. Eventually, its processing completes, and it returns to a server in $\mathcal{Q}_0$. The time spent at this server models the think time of the user; the next request of the session is issued subsequently. The infinite server system also enables the model to capture the independence of the user think times from the request service times at the application.

Let S_i denote the service time of a request at $\mathcal{Q}_i$ ($1 \leq i \leq M$). Also, p_i denotes the probability of a request making a transition from $\mathcal{Q}_i$ to $\mathcal{Q}_{i-1}$ (note that $p_M = 1$); p_1 denotes the probability of transition from $\mathcal{Q}_1$ to $\mathcal{Q}_0$. Finally, let Z denote the service time at any server in $\mathcal{Q}_0$ (which is essentially the user think time). Our model requires these parameters as inputs in order to compute the average end-to-end response time of a request.

Our discussion thus far has implicitly assumed that sessions never terminate. In practice, the number of sessions being serviced will vary as existing sessions terminate and new sessions arrive. Our model can compute the mean response time for a given number of concurrent sessions N. This property can be used for admission control at the application sentry as discussed in Section 6.2.

3.2 Deriving Response Times From the Model

The Mean-Value Analysis (MVA) algorithm [Reiser and Lavenberg 1980] for closed-queuing networks can be used to compute the mean response time experienced by a request in our network of queues. The MVA algorithm is based on the following key queuing theory result: in product-form closed queuing networks[4], when a request moves from queue $\mathcal{Q}_i$ to another queue $\mathcal{Q}_j$, at the time of its arrival at $\mathcal{Q}_j$, it sees, a system with the same statistics as a system with one less customer. Consider a product-form closed-queuing network with N customers. Let $\bar{A}_m(N)$ denote the average number of customers in queue $\mathcal{Q}_m$ seen by an arriving customer. Let $\bar{L}_m(N)$ denote the average length of queue $\mathcal{Q}_m$ in such a system. Then, the previous result implies

$$\bar{A}_m(N) = \bar{L}_m(N-1). \quad (1)$$

Given this result, the MVA algorithm iteratively computes the average response time of a request. The MVA algorithm uses Equation (1) to introduce customers into the queuing network one-by-one and determines the resulting average delays at various queues at each step. It terminates when all N

[4]The term product-form applies to any queuing network in which the expression for the equilibrium probability has the form of $P(n_1, \ldots, n_M) = \frac{1}{G(N)} \pi_{i=1}^{M} f_i(n_i)$, where $f_i(n_1)$ is some function of the number of jobs at the ith queue, and G(N) is a normalizing constant. Product form solutions are known to exist for a broad class of networks, including ones where the scheduling discipline at each queue is processor sharing (PS).

Algorithm 1. Mean-value analysis algorithm for an M-tier application.

```
input        : N, S̄_m, V_m, 1 ≤ m ≤ M; Z̄
output       : R̄_m (avg. delay at Q_m), R̄ (avg. resp. time)
initialization:
R̄_0 = D̄_0 = Z̄; L̄_0 = 0;
for m = 1 to M do
    L̄_m = 0;
    D̄_m = V_m · S̄_m /* service demand */;
end
/* introduce N customers, one by one */
for n = 1 to N do
    for m = 1 to M do
        R̄_m = D̄_m · (1 + L̄_m) /* average delay */;
    end
    τ = ( n / (R̄_0 + Σ_{m=1}^{M} R̄_m) ) /* throughput */;
    for m = 1 to M do
        L̄_m = τ · R̄_m /* little's law *;
    end
    L̄_0 = τ · R̄_0;
end
R̄ = Σ_{m=1}^{m=M} R̄_m /* response time */;
```

Table I. Notation Used in Describing the MVA Algorithm

Symbol	Meaning
M	Number of servers
N	Number of sessions
$\mathcal{Q}_m$	Queue representing tier T_m $(1 \leq m \leq M)$
$\mathcal{Q}_0$	Inf. server system to capture sessions
$\bar{Z}$	User think time
$\bar{S}_m$	Avg. per-request service time at $\mathcal{Q}_m$
$\bar{L}_m$	Avg. length of $\mathcal{Q}_m$
τ	Throughput
$\bar{R}_m$	Avg. per-request delay at $\mathcal{Q}_m$
$\bar{R}$	Avg. per-request response time
$\bar{D}_m$	Avg. per-request service demand at $\mathcal{Q}_m$
V_m	Visit ratio for $\mathcal{Q}_m$
$\bar{A}_m$	Avg. number of customers in $\mathcal{Q}_m$ seen by an arriving customer

customers have been introduced and yields the average response time experienced by N concurrent customers. Note that a session in our model corresponds to a customer in the result described by Equation (1). The MVA algorithm for an M-tier Internet application servicing N sessions simultaneously is presented in Algorithm 1 and the associated notation is in Table I.

The algorithm uses the notion of a *visit ratio* for each queue $\mathcal{Q}_1, \ldots, \mathcal{Q}_M$. The visit ratio V_m for queue $\mathcal{Q}_m$ $(1 \leq m \leq M)$ is defined as the average number of

visits made by a request to $\mathcal{Q}_m$ during its processing (that is, from when it emanates from $\mathcal{Q}_0$ and when it returns to it). Visit ratios are easy to compute from the transition probabilities $p_1, \ldots, p_M$ and provide an alternate representation of the queuing network. Notice that the visit ratio is only concerned with the mean number of visits made by a request to a queue and not when or in what order these visits occur. Consequently, our algorithm is equally suitable for deriving mean response times regardless of whether a request to a tier triggers multiple requests at its succeeding tier in parallel or in sequence.

Thus, given the average service times and visit ratios for the queues, the average think time of a session, and the number of concurrent sessions, the algorithm computes the average response time $\bar{R}$ of a request.

3.3 Estimating the Model Parameters

In order to compute the response time, the model requires several parameters as inputs. In practice, these parameters can be estimated by monitoring the application as it services its workload. To do so, we assume that the underlying operating system and application software components (such as the Apache Web server) provide monitoring hooks to enable accurate estimation of these parameters. Our experience with the Linux-based multitier applications used in our experiments is that such functionality is either already available or can be implemented at a modest cost. The rest of this section describes how the various model parameters can be estimated in practice.

Estimating Visit Ratios. The visit ratio for any tier of a multitier application is the average number of times that tier is invoked during a request's lifetime. Let λ_{req} denote the number of requests serviced by the entire application over a duration t. Then the visit ratio for tier T_i can be simply estimated as

$$V_i \approx \frac{\lambda_i}{\lambda_{req}},$$

where λ_i is the number of requests serviced by that tier in that duration. By choosing a suitably large duration t, a good estimate for V_i can be obtained. We note that the visit ratios are easy to estimate in an online fashion. The number of requests serviced by the application λ_{req} can be monitored at the application sentry. For replicated tiers, the number of requests serviced by all servers of that tier can be monitored at the dispatchers. Monitoring of both parameters requires simple counters at these components. For nonreplicated tiers that lack a dispatcher, the number of serviced requests can be determined by real-time processing of the tier logs. In the database tier, for instance, the number of queries and transactions processed over a duration t can be determined by processing the database log using a script.

Estimating Service Times. Application components such as Web, Java, and database servers all support extensive logging facilities and can log a variety of useful information about each serviced request. In particular, these components can log the residence time of individual requests as observed at that tier;

the residence time includes the time spent by the request at this tier and all the subsequent tiers that processed this request. This logging facility can be used to estimate per-tier service times. Let $\bar{X}_i$ denote the average per-request residence time at tier T_i. We start by estimating the mean service time at the last tier. Since this tier does not invoke services from any other tiers, the request execution time at this tier under lightly loaded conditions is an excellent estimate of the service time. Thus, we have,

$$\bar{S}_M \approx \bar{X}_M.$$

Let S_i, X_i, and n_i be random variables denoting the service time of a request at a tier T_i, residence time of a request at tier T_i, and the number of times T_i requests service from T_{i+1} as part of the overall request processing, respectively. Then, under lightly loaded conditions,

$$S_i = X_i - n_i \cdot X_{i+1},\ 1 \le i < M.$$

Taking averages on both sides, we get,

$$\bar{S}_i = \bar{X}_i - E[n_i \cdot X_{i+1}].$$

Since n_i and X_{i+1} are independent, this gives us,

$$\bar{S}_i = \bar{X}_i - \bar{n}_i \cdot \bar{X}_{i+1} = \bar{X}_i - \left(\frac{V_{i+1}}{V_i}\right) \cdot \bar{X}_{i+1}.$$

Thus, the service times at tiers $T_1, \ldots, T_{M-1}$ can be estimated. Modern Internet applications are typically well-provisioned and are likely to experience periods of low-intensity workloads when such measurements of service times may be conducted in an online fashion.

Estimating Think Times. The average user think time, $\bar{Z}$, can be obtained by recording the arrival and finish times of individual requests at the sentry. $\bar{Z}$ is estimated as the average time elapsed between when a request finishes and when the next request (belonging to the same session) arrives at the sentry. By using a sufficient number of observations, we can obtain a good estimate of $\bar{Z}$.

Increased Service Times During Overloads. Our estimation of the tier-specific service times assumed lightly loaded conditions. As the load on a tier grows, software overheads such as waiting on locks, virtual memory paging, and context switch overheads that are not captured by our model can become significant components of the request processing time.

Incorporating the impact of increased context-switching overhead or contention for memory or locks into our model is nontrivial. Rather than explicitly modeling these effects, we implicitly account for their impact by associating increased service times with requests under heavy loads. We use the Utilization Law [Lazowska et al. 1984] for a queuing system which states that $S = \rho/\tau$, where ρ and τ are the queue utilization and throughput, respectively. Consequently, we can improve our estimate of the average service time at tier T_i as

$$\bar{S}'_i = \max\left(\bar{S}_i, \frac{\rho_i}{\tau_i}\right),$$

where ρ_i is the utilization of the busiest resource (e.g., CPU, disk, or network interface) and τ_i is the tier throughput. Since all modern operating systems support facilities for monitoring system performance (e.g., the sysstat package in Linux (Sysstat Package http://freshmeat.net/projects/sysstat)), the utilizations of various resources is easy to obtain online. Similarly, the tier throughput ρ_i can be determined at the dispatcher (or from logs) by counting the number of completed requests in a duration t.

When to Update the Model Parameters. A crucial decision affecting the performance of our model concerns the frequency at which the previous parameters are updated. We believe that this is a significant research problem on its own and consider it beyond the scope of this article. However, we identify the following intuitively meaningful requirements on when or how frequently these decisions should be made. Whereas a very high frequency may result in unstable/oscillatory behavior due to inadequate data for statistically robust estimation of parameters, a very low frequency may result in decision-making that is not reactive enough in capturing changes in workload characteristics. Some recent research on empirically determining appropriate frequencies for recomputing the parameters of a model is of direct relevance to this article [Chandra et al. 2003; Chandra et al. 2003].

4. MODEL ENHANCEMENTS

This section proposes enhancements to our baseline model to capture four application artifacts: replication and load imbalance at tiers, concurrency limits, and multiple session classes.

4.1 Replication and Load Imbalance at Tiers

Recall that our baseline model assumes a single server (queue) per-tier and consequently does not support the notion of replication at a tier. We now enhance our model to handle this scenario. Let r_i denote the number of replicas at tier T_i. Our approach to capture replication at tier T_i is to replace the single queue $\mathcal{Q}_i$ with r_i queues, $\mathcal{Q}_{i,1}, \ldots, \mathcal{Q}_{i,r_i}$, one for each replica. A request in any queue can now make a transition to any of the r_{i-1} queues of the previous tier or to any of the r_{i+1} queues of the next tier.

In general, whenever a tier is replicated, a dispatcher is necessary to distribute requests to replicas. The dispatcher determines which request to forward to which replica and directly influences the transitions made by a request.

The dispatcher is also responsible for balancing load across replicas. In a perfectly load balanced system, each replica processes $\frac{1}{r_i}$ fraction of the total workload of that tier. In practice, however, perfect load balancing is difficult to achieve for the following reasons. First, if a session is stateful, successive requests will need to be serviced by the same stateful server at each tier; the dispatcher is forced to forward all requests from a session to this replica regardless of the load on other replicas. Second, if caching is employed by a tier, a session and its requests may be preferentially forwarded to a replica where a response is likely to be cached. Thus, sessions may have affinity for particular replicas. Third, different sessions impose different processing demands.

This can result in variability in the resource usage of sessions, and simple techniques such as forwarding a new session to the least-loaded replica may not be able to counter the resulting load imbalance. Thus, the issues of replication and load imbalance are closely related. Our enhancement captures the impact of both these factors.

In order to capture the load imbalance across replicas, we explicitly model the load at individual replicas. Let λ_i^j denote the number of requests forwarded to the jth most loaded replica of tier T_i over some duration t. Let λ_i denote the total number of requests handled by that tier over this duration. Then, the imbalance factor β_i^j is computed as

$$\beta_i^j = \left(\frac{\lambda_i^j}{\lambda_i}\right).$$

We use exponentially smoothed averages of these ratios $\bar{\beta}_i^j$ as measures of the load imbalance at individual replicas. The visit ratios of the various replicas are then chosen as

$$V_{i,j} = V_i \bar{\beta}_i^j.$$

The higher the load on a replica, the higher the value of the imbalance factor, and the higher its visit ratio. In a perfectly load balanced system, $\beta_i^j = \frac{1}{r_i}, \forall j$. Observe that the number of requests forwarded to a replica λ_i^j and the total number of requests λ_i can be measured at the dispatcher using counters. The MVA algorithm can then be used with these modified visit ratios to determine the average response time.

4.2 Handling Concurrency Limits at Tiers

The software components of an Internet application have limits on the amount of concurrency they can handle. For instance, the Apache Web server uses a configurable parameter to limit the number of concurrent threads or processes that are spawned to service requests. This limit prevents the resident memory size of Apache from exceeding the available RAM and prevents thrashing. Connections may be turned away when this limit is reached. A connection request that arrives when this limit has been reached is queued up in an operating system buffer. If the connection does not get processed for a certain amount of time, it is dropped. Any connections that arrive to find the aforementioned operating system buffer full are summarily dropped. Other tiers impose similar limits.

The model developed thus far assumes that each replica at any tier can service an unbounded number of simultaneous requests and fails to capture the behavior of the application when the concurrency limit is reached at any software component. This is depicted in Figure 4(a), which shows the response time of a three-tier application called Rubis that is configured with a concurrency limit of 150 for the Apache Web server and a limit of 75 for the middle Java tier (details of the application appear in Section 5.1). These concurrency limits were the default values that were found in the configuration files accompanying these software components. As shown, the response times predicted

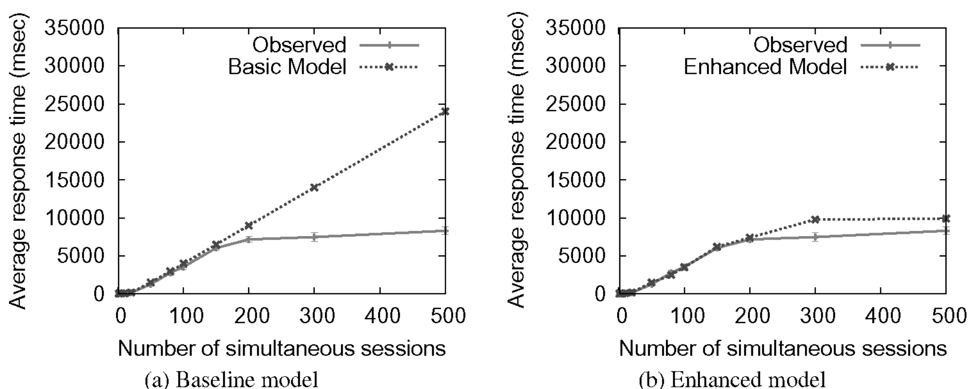


Fig. 4. Response time of Rubis with 95% confidence intervals. A concurrency limit of 150 for Apache and 75 for the Java servlet tier is used. Figure (a) depicts the deviation of the baseline model from observed behavior when the concurrency limit is reached. Figure (b) depicts the ability of the enhanced model to capture this effect.

by the model match the observed response times until the concurrency limit is reached. Beyond this point, the model continues to assume an increasing number of simultaneous requests being serviced and predicts an increase in response time, while the actual response time of successful requests shows a flat trend due to an increasing number of dropped requests.

In general, when the concurrency limit is reached at a software component in tier T_i, one of two actions are possible: (1) it can silently drop additional requests and rely upon a timeout mechanism in the software component in tier T_{i-1} that issued this request to detect these drops, or (2) it can explicitly notify tier T_{i-1} of its inability to serve the request (by returning an error message). In either case, tier T_{i-1} may reissue the request some number of times before abandoning its attempts. It will then either drop the request or explicitly notify its preceding tier. Finally, tier T_1 can notify the client of the failure.

Rather than distinguishing these possibilities, we employ a general approach for capturing these effects. As before, we use $V_{i,j}$ to denote the visit ratio to the replica $\mathcal{Q}_{i,j}$ in tier T_i. Notice that the online technique described in Section 3.3 will not accurately estimate the visit ratio at a software component if its concurrency limit has been reached. This is because the concurrency limit will cause some requests to be dropped, whereas the technique presented in Section 3.3 is based on the assumption that all requests arriving at a software component are successfully serviced by it. Therefore, our enhancement relies on visit ratios estimated using offline measurements conducted with all concurrency limits set to sufficiently high values. These visit ratios are corrected to capture load imbalances at replicated tiers exactly as described in Section 4.1. Let K_i denote the concurrency limit at $\mathcal{Q}_{i,j}(1 \leq j \leq r_i)$. To capture requests that are dropped at $\mathcal{Q}_{i,j}$ when its concurrency limit is reached, we add an additional transition to the model developed thus far. At the entrance of $\mathcal{Q}_{i,j}$, we add a transition into an infinite server queuing subsystem $\mathcal{Q}_{i,j}^{drop}$. Let $V_{i,j}^{drop}$ denote the visit ratio for $\mathcal{Q}_{i,j}^{drop}$ as shown in Figure 5. For the sake of clarity, we have only

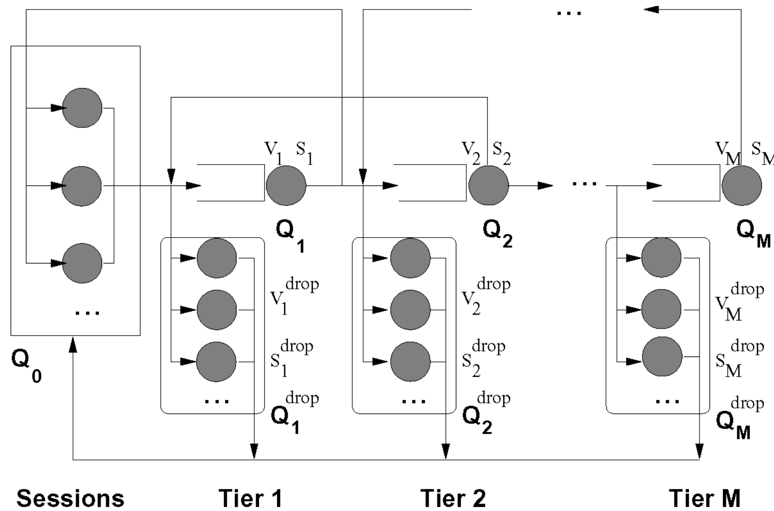


Fig. 5. Multitier application model enhanced to handle concurrency limits. Since each tier has only one replica, we use only one subscript in our notation.

shown one replica at each tier. $\mathcal{Q}_{i,j}^{drop}$ has a mean service time of S_i^{drop}; notice that this is the same for all the replicas in the tier T_i. This enhancement allows us to distinguish between the processing of requests that get dropped due to concurrency limits and those that are processed successfully. Requests that are processed successfully are modeled exactly as in the basic model. Requests that are dropped at $\mathcal{Q}_{i,j}$ experience some delay in the subsystem $\mathcal{Q}_{i,j}^{drop}$ before returning to $\mathcal{Q}_0$; this models the delay between when a request is dropped at tier T_i and when this information gets propagated to the client that initiated the request.

Like in the baseline model, we can use the MVA algorithm to compute the response time of a request. The algorithm computes the fraction of requests that finish successfully and those that encounter failures as well as the delays experienced by both types of requests. To do this, we need to estimate the additional parameters that we have added to our basic model, namely, $V_{i,j}^{drop}$ for each replica in tier T_i and S_i^{drop} for each tier T_i.

Estimating $V_{i,j}^{drop}$. Our approach to estimate $V_{i,j}^{drop}$ consists of the following two steps.

Step 1. Estimate throughput of the queuing network if there were no concurrency limits: solve the queuing network shown in Figure 5 using the MVA algorithm using $V_{i,j}^{drop} = 0$ (i.e., assuming that the queues have no concurrency limits). Let λ denote the throughput computed by the MVA algorithm in this step.

Step 2. Estimate $V_{i,j}^{drop}$: treat $\mathcal{Q}_{i,j}$ as an open, finite-buffer M/M/1/K_i queue with arrival rate $\lambda V_{i,j}$ (using the λ computed in Step (1)). Let $p_{i,j}^{drop}$ denote the probability of buffer overflow in this M/M/1/K_i queue [Kleinrock 1975]. Then $V_{i,j}^{drop}$ is estimated as $V_{i,j}^{drop} = p_{i,j}^{drop} \cdot V_{i,j}$. Also, $V_{i,j}$ is updated as $V_{i,j} = (1 - p_{i,j}^{drop}) \cdot V_{i,j}$.

Estimating S_i^{drop}. An estimate of S_i^{drop} is application-specific and depends on the manner in which information about dropped requests is conveyed to the client and how the client responds to it. In our current model, we make the simplifying assumption that, upon detecting a failed request, the client reissues the request. This is captured by the transitions from $\mathcal{Q}_{i,j}^{drop}$ $(1 \leq j \leq r_i)$ back to $\mathcal{Q}_0$ in Figure 5. Our approach for estimating S_i^{drop} is to subject the application to an offline workload that causes the limit to be exceeded only at tier T_i (this can be achieved by setting a low concurrency limit at that tier and sufficiently high limits at all the other tiers), and then record the response times of the requests that do not finish successfully. S_i^{drop} is then estimated as the difference between the average response time of these unsuccessful requests and the sum of the service times at tiers $T_1, \ldots, T_{i-1}$, multiplied by their respective visit ratios.

In Figure 4(b), we plot the response times for Rubis as predicted by our enhanced model. We find that this enhancement enables us to capture the behavior of the Rubis application even when its concurrency limit is reached.

4.3 Handling Multiple Session Classes

Internet applications typically classify incoming sessions into multiple *classes*. To illustrate, an online brokerage Web site may define three classes and may map financial transactions to the *Gold* class, customer requests such as balance inquiries to the *Silver* class, and casual browsing requests from noncustomers to the *Bronze* class. Typically such classification helps the application sentry to preferentially admit requests from more important classes during overloads and drop requests from less important classes.

We can extend our baseline model to account for the presence of different session classes and to compute the response time of requests within each class. Consider an Internet application with C session classes: $C_1, C_2, \ldots, C_C$. Assume that the sentry implements a classification algorithm to map each incoming session to one of these classes. We can use a straightforward extension of the MVA algorithm to deal with multiple session classes. This is presented in Algorithm 2. The notation used in this algorithm is a simple extension of that used in Algorithm 1 with an additional subscript c for requests of class c. N_c denotes the number of sessions of class c. We denote the total number of sessions by N as before, so $N = \sum_{c=1}^{C} N_c$. This algorithm is based on the following extension of the result (1). Let $\underline{N} - 1_c = (N_1, \ldots, N_{c-1}, N_c - 1, N_{c+1}, \ldots, N_c)$. For closed product form networks,

$$\bar{A}_{c,m}(\underline{N}) = \bar{L}_m(\underline{N} - 1_c). \tag{2}$$

The notion of feasible population used in Algorithm 2 needs explanation. A feasible population with n total sessions is a set of sessions such that the number of sessions within each class c is between 0 and N_c, and the sum of the number of sessions in all classes is n.

Algorithm 2. Mean-value analysis algorithm for an M-tier application with C classes.

input : N_c (num. sessions of class c), $\bar{S}_{c,m}, V_{c,m}, 1 \le c \le C, 1 \le m \le M; \bar{Z}$
output : $\bar{R}_{c,m}$ (avg. delays at Q_m), $\bar{R}_c$ (avg. resp. time for class c), $1 \le c \le C$

initialization:

for $c = 1$ ***to*** C **do**
  $\bar{R}_{c,0} = \bar{D}_{c,0} = \bar{Z}$;
end
$\bar{L}_0(\underline{0}) = 0$;

for $m = 1$ ***to*** M **do**
  $\bar{L}_m(\underline{0}) = 0$;
  for $c = 1$ ***to*** C **do**
    $\bar{D}_{c,m} = V_{c,m} \cdot \bar{S}_{c,m}$ /* service demand */;
  end
end
/* introduce N customers, one by one */

for $n = 1$ ***to*** N **do**
  for each feasible popl. $\underline{n} = (n_1, \ldots, n_C)$ s. t. $n = \sum_{c=1}^{C} n_c, n_c \ge 0$

  for $c = 1$ ***to*** C **do**
    for $m = 1$ ***to*** M **do**
      $\bar{R}_{c,m} = \bar{D}_{c,m} \cdot (1 + \bar{L}_m(\underline{n} - 1_c))$ /* average delay */;
    end
  end
  for $c = 1$ ***to*** C **do**
    $\tau_c = \left(\dfrac{n_c}{\bar{R}_{c,0} + \sum_{m=1}^{M} \bar{R}_{c,m}} \right)$ /* throughput */;
    for $m = 1$ ***to*** M **do**
      $\bar{L}_m(\underline{n}) = \sum_{c=1}^{C} \tau_c \cdot \bar{R}_{c,m}$ /* little's law */;
    end
  end
  $\bar{L}_0(\underline{n}) = \sum_{c=1}^{C} \tau_c \cdot \bar{R}_{c,0}$;

end
for $c = 1$ ***to*** C **do**
  for $m = 1$ ***to*** M **do**
    $\bar{R}_c = \sum_{m=1}^{m=M} \bar{R}_{c,m}$ /* response time */;
  end
end

We note that this algorithm requires the visit ratios, service times, and think time to be measured on a per-class basis. For handling load imbalances at replicated tiers, we propose to correct the per-class visit ratios by employing load imbalance factors determined using the heuristic described in Section 4.1. Our approach makes the simplifying assumption of identical load imbalance factors for the various classes at each tier. Finally, we refine our technique for dealing with concurrency limits presented in Section 4.2 to accommodate multiple classes. We estimate S_i^{drop} exactly as in Section 4.2 because this parameter is independent of the class of a request. The estimation of the drop probabilities, however, needs to be done on a per-class basis. We do this by enhancing the

two-step procedure described in Section 4.2. Let us denote by $V_{c,i,j}$ the visit ratio for class c requests at $Q_{i,j}$ and by $V_{c,i,j}^{drop}$ the visit ratio for class c requests at $Q_{i,j}^{drop}$.

Step 1. Estimate throughput of the queuing network if there were no concurrency limits: solve the queuing network using the multiclass MVA algorithm with $V_{c,i,j}^{drop} = 0, 1 \leq c \leq C$ (i.e., assuming that the queues have no concurrency limits). Let $\lambda = \sum_{c=1}^{C} \lambda_c$ denote the throughput computed by the MVA algorithm in this step.

Step 2. Estimate $V_{c,i,j}^{drop}$: treat $Q_{i,j}$ as an open, finite-buffer M/M/1/K_i queue with arrival rate $\lambda V_{i,j}$ (using the λ computed in Step (1)). Let $p_{i,j}^{drop}$ denote the probability of buffer overflow in this M/M/1/K_i queue [Kleinrock 1975]. Then $V_{c,i,j}^{drop}$ is estimated as: $V_{c,i,j}^{drop} = p_{i,j}^{drop} \cdot V_{c,i,j} \cdot \frac{\lambda_c}{\lambda}$. Also, $V_{c,i,j}$ is updated as: $V_{c,i,j} = (1 - p_{i,j}^{drop}) \cdot V_{c,i,j} \cdot \frac{\lambda_c}{\lambda}$.

Given a C-tuple $(N_1, \ldots, N_C)$ of sessions belonging to the C classes that are simultaneously serviced by the application, the algorithm can compute the average delays incurred at each queue and the end-to-end response time on a per-class basis. In Section 6.2, we discuss how this algorithm can be used to flexibly implement session policing policies in an Internet application.

4.4 Other Salient Features

Our closed-queuing model has several desirable features.

Simplicity. For an M-tier application with N concurrent sessions, the MVA algorithm has a time complexity of $O(MN)$. The algorithm is simple to implement, and as argued earlier, the model parameters are easy to measure online.

Generality. Our model can handle an application with an arbitrary number of tiers. Further, when the scheduling discipline is processor sharing (PS), the MVA algorithm works without making any assumptions about the service time distributions of the customers [Lazowska et al. 1984]. This feature is highly desirable for two reasons: (1) it is representative of scheduling policies in commodity operating systems (e.g., Linux's CPU time-sharing), and (2) it implies that our model is sufficiently general to handle workloads with an arbitrary service time requirements.[5]

While our model is able to capture a number of application idiosyncrasies, certain scenarios are not explicitly captured.

Multiple Resources. We model each server occupied by a tier using a single queue. In reality, the server contains various resources such as the CPU, disk, memory, and the network interface. Our model currently does not capture the utilization of various server resources by a request at a tier. An enhancement to the model where various resources within a server are modeled as a network of queues is the subject of future work.

[5]The applicability of the MVA algorithm is more restricted with some other scheduling disciplines. For example, in the presence of a FIFO scheduling discipline at a queue, the service time at a queue needs to be exponentially distributed for the MVA algorithm to be applicable.

Resources Held Simultaneously at Multiple Tiers. Our model essentially captures the passage of a request through the tiers of an application as a juxtaposition of periods, during each of which the request utilizes the resources at exactly one tier. Although this is a reasonable assumption for a large class of Internet applications, it does not apply to certain Internet applications such as streaming video servers. A video server that is constructed as a pipeline of processing modules will have all of its modules or tiers active as it continuously processes and streams a video to a client. Our model does not apply to such applications.

4.5 Limitations of Our Model

Whereas queueing theory provides an elegant and easy approach for modeling multitier applications, it also imparts certains shortcomings that must be understood.

—*Limited applicability in highly transient workload conditions.* Queueing models capture system operation under steady-state conditions, and hence are potentially of limited use if the workload characteristics change very fast.

—*Only averages captured.* Models that can capture entire distributions of key metrics such as response time are not yet well-developed in queueing theory literature. In many Internet data centers, a high percentile of the response time, and not just the average, is of interest.

—*Only single bottleneck resource captured.* Specific to our work, our model is applicable only to scenarios where a single temporal resoure (such as the CPU or network bandwidth) might be the bottleneck, and hence the effect of contention for other resources on the response time is negligible. Treating a server as a network of queues representing multiple resources inside it is certainly possible and has been explored by other researchers [Stewart et al. 2007]. Modeling spatial resources such as virtual memory or buffer caches needs different approaches. In a separate piece of work, we have explored the use of a machine learning technique for modeling cache behavior [Das et al. 2006]

5. MODEL VALIDATION

In this section, we present our experimental setup followed by our experimental validation of the model.

5.1 Experimental Setup

Applications. We use two open source multitier 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.

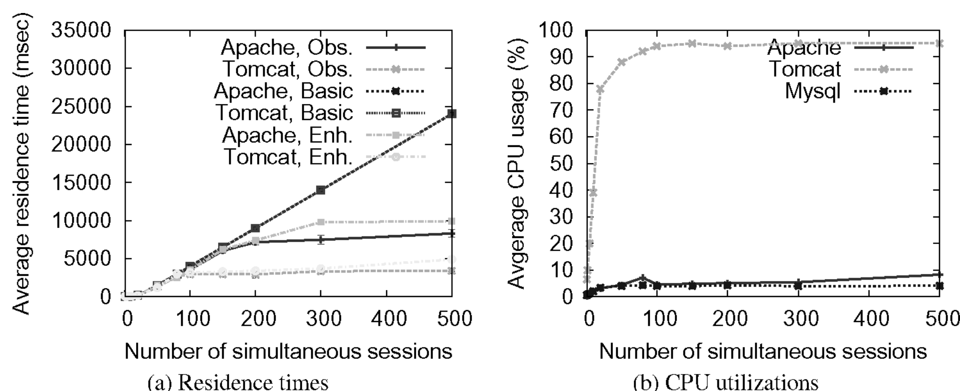
Both applications were developed by the DynaServer group at Rice University (Dynaserver project http://compsci.rice.edu/CS/Systems/DynaServer/). Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate the workloads and take the measurements needed by our experiments. We chose an average duration of 5 minutes for the sessions of both Rubis and Rubbos. For both applications, the think time was chosen from an exponential distribution with a mean of 1 second. The concurrency limits for various tiers were set to the default values that were found in the configuration files accompanying these software components.

We used 3-tier versions of these applications. The front tier was based on the Apache 2.0.48 Web server. We experimented with two implementations of the middle tier for Rubis (i) based on Java servlets, and (ii) based on Sun's J2EE Enterprise Java Beans (EJBs). The middle tier for Rubbos was based on Java servlets. We employed Tomcat 4.1.29 as the servlets container and JBoss 3.2.2 as the EJB container. We used Kernel TCP Virtual Server version 0.0.14 (Kernel TCP Virtual Server http://www.linusvirtualserver.org/software/ktcpvs/ktcpvs.html) to implement the application sentry. ktcpvs is an open source, Layer-7 request dispatcher implemented as a Linux kernel module. A round-robin load balancer implemented in ktcpvs was used for Apache. Request dispatching for the middle tier was performed by *mod_jk*, an Apache module that implements a variant of round-robin request distribution while taking into account session affinity. Finally, the database tier was based on the Mysql 4.0.18 database server.

The concurrency limits for our tiers were set as follows, unless stated otherwise: 150 for Apache, 75 for Java servlets container, and 150 for Mysql. Note that real-world Internet servers are likely to have much higher concurrency limits but they will also employ more capable hardware configurations. Our limits were chosen in accordance with the capabilities of the hardware available in our research group.

Hosting Environment. We conducted experiments with the applications hosted on two different kinds of machines. The first hosting environment consisted of IBM servers (model 6565-3BU) with 662MHz processors and 256MB RAM connected by 100Mbps ethernet. The second setting used for experiments reported in Section 6 had Dell servers with 2.8GHz processors and 512MB RAM interconnected using gigabit ethernet. This served to verify that our model was flexible enough to capture applications running on different types of machines. Finally, the workload generators were run on machines with Pentium-III processors with speeds of 450MHz-1GHz and RAM sizes in the range 128-512MB. All the machines ran the Linux 2.4.20 kernel.

Measurement Methodology. Unless otherwise specified, the average response times reported are taken over 30-minute periods. In any experiment, several initial readings are discarded until the system is perceived to have reached steady state. This is done as follows. We record the absolute differences between average response times recorded over successive 1 minute long intervals; 20 successive monotonically nonincreasing differences are taken as an indicator of the system having attained a steady state.



(a) Residence times (b) CPU utilizations

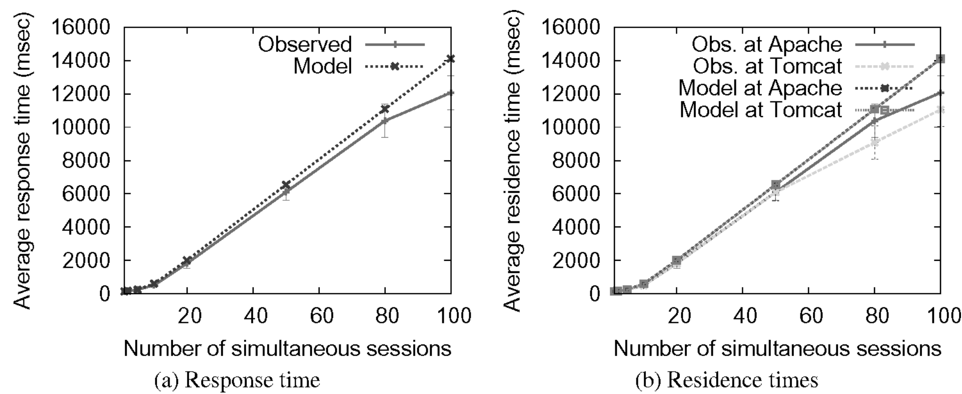
Fig. 6. Rubis based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

5.2 Performance Prediction

We conduct a set of experiments with the purpose of ascertaining the ability of our model to predict the response time of multitier applications. We experiment with (i) two kinds of applications (Rubis and Rubbos), (ii) two different implementations of Rubis (based on Java servlets and EJBs), and (iii) different workloads for Rubis. Each of the three application tiers are assigned one server except in the experiments reported in Section 5.4. We vary the number of concurrent sessions seen by the application and measure response times of successfully finished requests. Each experiment lasts an initial period during which the system is brought to a steady state followed by a recording period of 30 minutes. We compute the average response time and the 95% confidence intervals from these observations.

Our first experiment uses Rubis with a Java servlets-based middle tier. We use two different workloads—$W1$, CPU-intensive on the Java servlets tier, and $W2$, CPU-intensive on the database tier. These were created by modifying the Rubis client so that it generated an increased fraction of requests that stressed the desired tier. Earlier, in Figure 4(b), we had presented the average response time and 95% confidence intervals for sessions varying from 1 to 500 for the workload $W1$. Also plotted were the average response times predicted by our basic model and our model enhanced to handle concurrency limits. Additionally, we present the observed and predicted residence times in Figure 6(a). Figure 6(b) shows that the CPU on the Java servlets tier becomes saturated beyond 100 sessions for this workload. As already explained in Section 4.2, the basic model fails to capture the response times for workloads higher than about 100 sessions due to an increase in the fraction of requests that arrive at the Apache and servlets tiers only to be dropped because of the tiers operating at their concurrency limits. We find that our enhanced model is able to capture the effect of dropped requests at these high workloads and continues to predict response times well for the entire workload range.

Figures 7 and 8 plot the response times, the residence times, and the server CPU utilizations for servlets-based Rubis subjected to the workload $W2$ with



(a) Response time (b) Residence times

Fig. 7. Rubis based on Java servlets: bottleneck at CPU of database tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

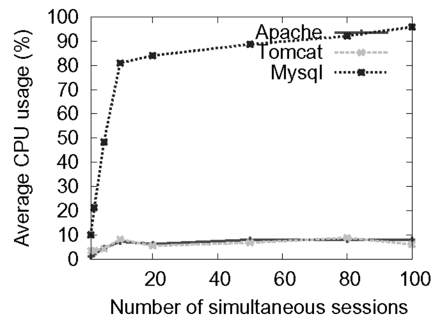
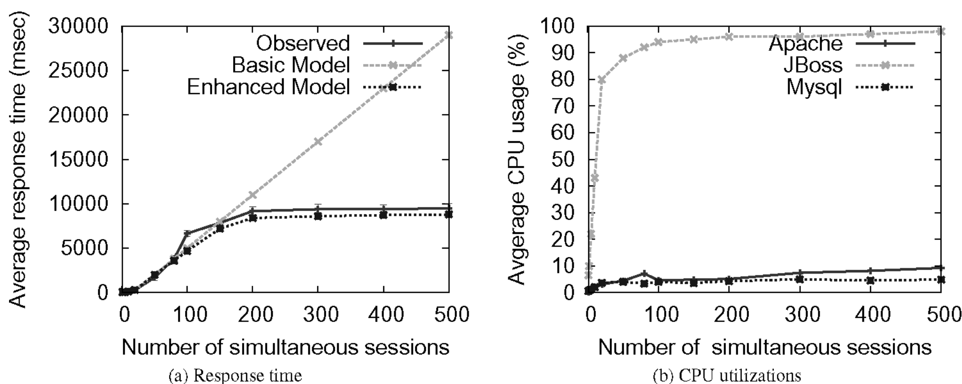


Fig. 8. Rubis based on Java servlets: bottleneck at CPU of database tier. CPU utilization at the database tier.

varying number of sessions. As shown in Figure 8, the CPU on the database server is the bottleneck resource for this workload. We find that our basic model captures response times well. The predicted response times are within the 95% confidence interval of the observed average response time for the entire workload range.

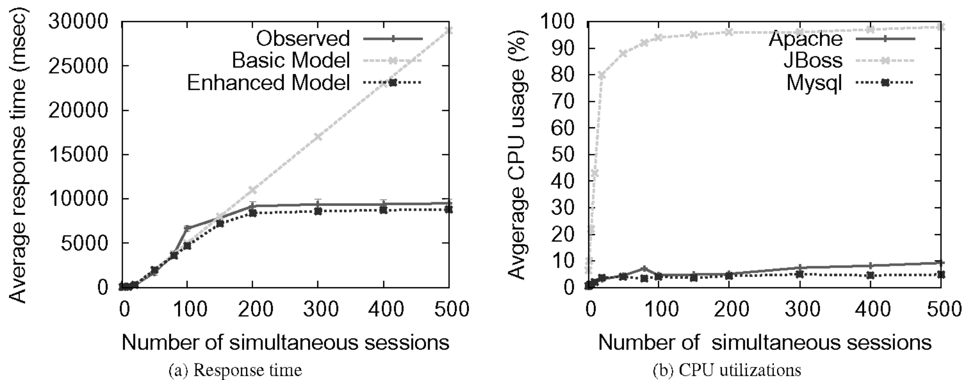
Next, we repeat the experiment just described with Rubis, based on an EJB-based middle tier. Our results are presented in Figure 9. Again, our basic model captures the response time well until the concurrency limits at Apache and JBoss are reached. As the number of sessions grows beyond this point, increasingly large fractions of requests are dropped, the request throughput saturates, and the response time of requests that finish successfully shows a flat trend. Our enhancement to the model is again found to capture this effect well.

Finally, we repeat the experiment with the Rubbos application. We use a Java servlets-based middle tier for Rubbos and subject the application to the workload $W1$ that is CPU-intensive on the servlets tier. Figure 10 presents the observed and predicted response times as well as the server CPU utilizations. We find that our enhanced model predicts response times well over the chosen workload range for Rubbos.



(a) Response time (b) CPU utilizations

Fig. 9. Rubis based on EJB: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.



(a) Response time (b) CPU utilizations

Fig. 10. Rubbos based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

5.3 Query Caching at the Database

Recent versions of the Mysql server feature a query cache. When in use, the query cache stores the text of a SELECT query together with the corresponding result that was sent to the client. If the identical query is received later, the server retrieves the results from the query cache rather than parsing and executing the query again. Query caching at the database has the effect of reducing the average service time at the database tier. We conduct an experiment to determine how well our model can capture the impact of query caching on response time. We subject Rubbos to a workload consisting of 50 simultaneous sessions. To simulate different degrees of query caching at Mysql, we use a feature of Mysql queries that allows the issuer of a query to specify that the database server not use its cache for servicing this query.[6] We modified the Rubbos servlets to make them request different fractions of the queries with this option. For each degree of caching, we plot the average response time with

[6]Specifically, replacing a `SELECT` with `SELECT SQL_NO_CACHE` ensures that Mysql does not cache this query.

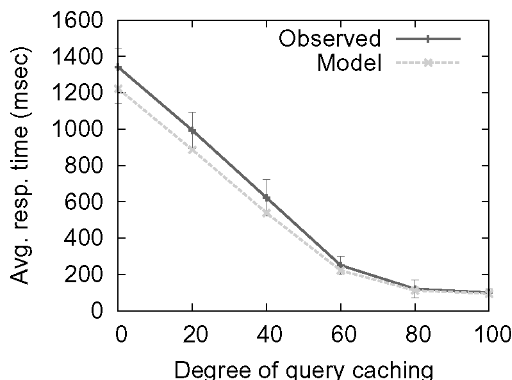


Fig. 11. Caching at the database tier of Rubbos.

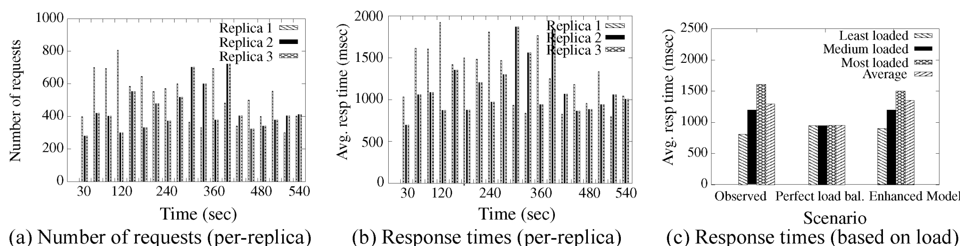


Fig. 12. Load imbalance at the middle tier of Rubis. (a) and (b) present number of requests and response times classified on a per-replica basis; (c) presents response times classified according to most loaded, second most loaded, and most loaded replicas and overall average response times.

95% confidence intervals in Figure 11. As expected, the observed response time decreases steadily as the degree of query caching increases. The average response time is nearly 1400 milliseconds without query caching and reduces to about 100 milliseconds when all the queries are cached. In Figure 11, we also plot the average response time predicted by our model for different degrees of caching. We find that our model is able to capture well the impact of the reduced query processing time with increasing degrees of caching on average response time. The predicted response times are found to be within the 95% confidence interval of the observed response times for the entire range of query caching.

5.4 Load Imbalance at Replicated Tiers

We configure Rubis using a replicated Java servlets tier; we assign three servers to this tier. We use the workload $W1$ with 100 simultaneous sessions. The user think times for a session are chosen using an exponential distribution whose mean is chosen uniformly at random from the set $\{1\ second, 5\ seconds\}$. We choose short-lived sessions with a mean session duration of 1 minute. Our results are presented in Figure 12. Note that replication at the middle tier causes the response times to be significantly smaller than in the experiment depicted in Figure 6(a). Further, choosing sessions with two widely different think times ensures variability in the workload imposed by individual sessions and creates load imbalance at the middle tier.

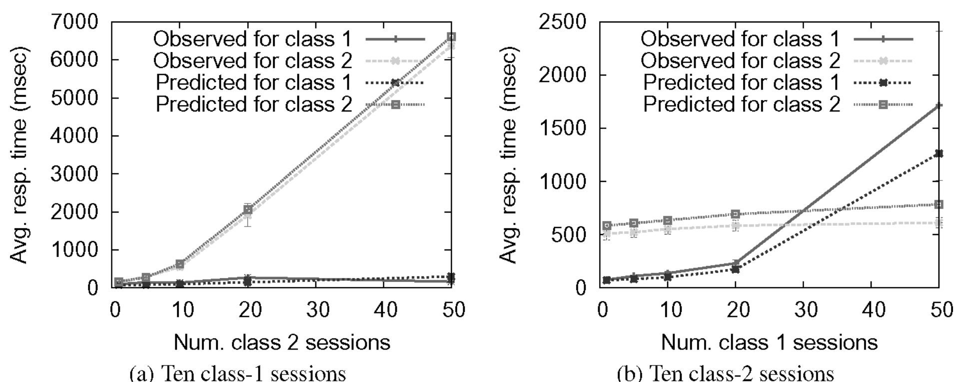
Figure 12(a) plots the number of requests passing through each of the three servers in the servlets tier over 30 second intervals during a 10-minute run of this experiment; Figure 12(b) plots the average end-to-end response times for these requests. These figures show the imbalance in the load on the three replicas. Also, the most loaded server changes over time—choosing a short session duration causes the load imbalance to shift among replicas frequently. Figure 12(c) plots the average response times observed for requests passing through the three servers; instead of presenting response times corresponding to specific servers, we plot values for the least loaded, the second least loaded, and the most loaded server. Figure 12(c) also shows the response times predicted by the model assuming perfect load balancing at the middle tier. Under this assumption, we see a deviation between the predicted values and the observed response times.

Next, we use the model enhancement described in Section 4.1 to capture load imbalance. For this workload, the values for the load imbalance factors used by our enhancement were determined to be $\bar{\beta}_2^1 = 0.25$, $\bar{\beta}_2^2 = 0.32$, and $\bar{\beta}_2^3 = 0.43$. We plot the response times predicted by the enhanced model at the extreme right in Figure 12(c). We observe that the use of these additional parameters improves our prediction of the response time. The predicted average response time (1350 milliseconds) closely matched the observed value (1295 milliseconds); with the assumption of perfect load balancing, the model underestimated the average response time to be 950 milliseconds.

It should be pointed out that the load balance described disappears if we take averages over longer time periods (on the order of tens of minutes as in the rest of our evaluation). Our concern for the impact of load imbalance, therefore, is meaningful when the SLA is sensitive to average response times over short time scales. This could happen if the SLA is concerned with averages taken over shorter time scales or with a high percentile of the response time distribution instead of the average response time. Examples of services that would find such SLAs useful are streaming media servers (Real Media Servers http://www.realnetworks.com/products/media_delivery.html) that need to ensure guaranteed transfers of large amounts of data to their clients every second or extremely time-sensitive applications such as game servers (Quake I http://www.planetquake.com).

5.5 Multiple Session Classes

We created two classes of Rubis sessions using the workloads $W1$ and $W2$, respectively. Recall that the requests in these classes have different service time requirements at different tiers, $W1$ is CPU-intensive on the Java servlets tier, while $W2$ is CPU-intensive on the database tier. We conduct two sets of experiments, each of which involves keeping the number of sessions of one class fixed at 10 and varying the number of sessions of the other class. We then compute the per-class average response time predicted by the multiclass version of our model (Section 4.3). We plot the observed and predicted response times for the two classes in Figure 13. While the predicted response times closely match the observed values for the first experiment, in the second experiment



(a) Ten class-1 sessions (b) Ten class-2 sessions

Fig. 13. Rubis serving sessions of two classes. Sessions of class 1 were generated using workload $W1$, while those of class 2 were generated using workload $W2$.

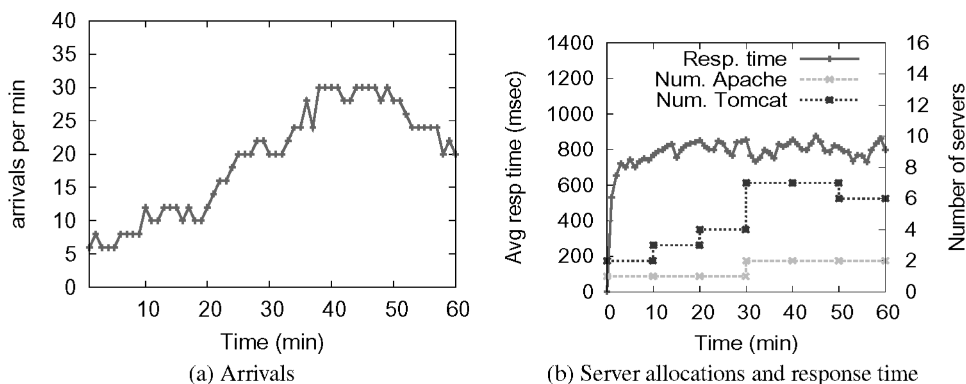
(Figure 13(b)), we observe that our model underestimates the response time for class 1 for 50 sessions. We attribute this to an inaccurate estimation of the service time of class 1 requests at the servlets tier at this load.

6. APPLICATIONS OF THE MODEL

In this section, we demonstrate some applications of our model for managing resources in a data center. We also discuss some important issues related to the online use of our model.

6.1 Dynamic Capacity Provisioning and Bottleneck Identification

Dynamic capacity provisioning is a useful technique for handling the multi-time scale variations seen in Internet workloads and has been a subject of extensive research [Chase et al. 2001; Urgaonkar et al. 2002, 2005; Doyle et al. 2003; Benani and Menasce 2005; Xu and Xu 2004; Appleby et al. 2001; Chandra et al. 2003; Chen et al. 2004, 2005; Rolia et al. 2000; Kallahalla et al. 2004; Zhu and Singhal 2001; Urgaonkar 2005; Ranjan et al. 2002]. The goal of dynamic provisioning is to dynamically allocate sufficient capacity to the tiers of an application so that its response time needs can be met even in the presence of the peak workload. Two key components of a dynamic provisioning technique are: (i) predicting the workload of an application, and (ii) determining the capacity needed to serve this predicted workload. The former problem has been addressed in papers such as Hellerstein et al. [1999]. The workload estimates made by such predictors can be used by our model to address the issue of how much capacity to provision. Observe that the inputs to our model-based provisioning technique are the workload characteristics, number of sessions to be serviced simultaneously, and the response time target, and the desired output is a capacity assignment for the application. We start with an initial assignment of one server to each tier. We use the MVA algorithm to determine the resulting average response time as described in Sections 3 and 4. In case this is worse than the target, we use the MVA algorithm to determine, for each replicable tier, the response time resulting from the addition of one more server to it. We



(a) Arrivals

(b) Server allocations and response time

Fig. 14. Model-based dynamic provisioning of servers for Rubis.

add a server to the tier that results in the most improvement in response time. We repeat this until we have an assignment for which the predicted response time is below the target; this assignment yields the capacity to be assigned to the application's tiers.[7] Note that the procedure described previously is only a heuristic for dynamic provisioning chosen for illustrative purposes. This procedure has a time complexity of $O(kMN)$, where k is the number of servers that the application is eventually assigned, M is the the number of tiers, and N is the number of sessions. Since provisioning decisions are typically made over periods of tens of minutes or hours, this overhead is practically feasible.

We conduct an experiment to demonstrate the application of our model to dynamically provision Rubis configured using Java servlets at its middle tier. We assume an idealized workload predictor that can accurately forecast the workload for the near future. We generated a 1-hour session arrival process based on a Web trace from the 1998 Soccer World Cup site [Arlitt and Jin 1999]; this is shown in Figure 14(a). Sessions are generated according to this arrival process using workload $W1$.

We implemented a provisioning unit that invokes the model-based procedure previously described every 10 minutes to determine the capacity required to handle the workload during the next interval. Our goal was to maintain an average response time of 1 second for Rubis requests. Since our model requires the number of simultaneous sessions as input, the provisioning unit converted the peak rate during the next interval into an estimate of the number of simultaneous sessions for which to allocate capacity using Little's Law [Kleinrock 1975] as $N = \Lambda \cdot d$, where Λ is the peak session arrival rate during the next interval as given by the predictor, and d is the average session duration. The provisioning unit ran on a separate server. It implemented scripts that remotely log on to the application sentry and the dispatchers for the affected tiers after every recomputation to enforce the newly computed allocations. The concurrency

[7]Note that our current discussion assumes that it is always possible to meet the response time target by adding enough servers. Sometimes this may not be possible (e.g., due to the workload exceeding the entire available capacity, or a nonreplicable tier becoming saturated), and we may have to employ admission control in addition to provisioning. This is discussed in Section 6.2.

limits of the Apache Web server and the Tomcat servlets container were both set to 100. We present the working of our provisioning unit and the performance of Rubis in Figure 14(b). The provisioning unit is successful in changing the capacity of the servlets tier to match the workload (recall that workload $W1$ is CPU-intensive on this tier). The session arrival rate goes up from about 10 sessions/minute at $t = 20$ minutes to nearly 30 sessions/minute at $t = 40$ minutes. Correspondingly, the request arrival rate increases from about 1500 requests/minute to about 4200 requests/minute. The provisioning unit increases the number of Tomcat replicas from 2 to a maximum of 7 during the experiment. Further, at $t = 30$ minutes, the number of simultaneous sessions during the upcoming 10-minute interval is predicted to be higher than the concurrency limit of the Apache tier. To prevent new sessions from being dropped due to the connection limit being reached at Apache, a second Apache server is added to the application. Thus, our model-based provisioning is able to identify potential bottlenecks at different tiers (connections at Apache and CPU at Tomcat) and maintain response time targets by adding capacity appropriately. We note that the single-tier models described in Section 2.3 will only be able to add capacity to one tier and will fail to capture such changing bottlenecks.

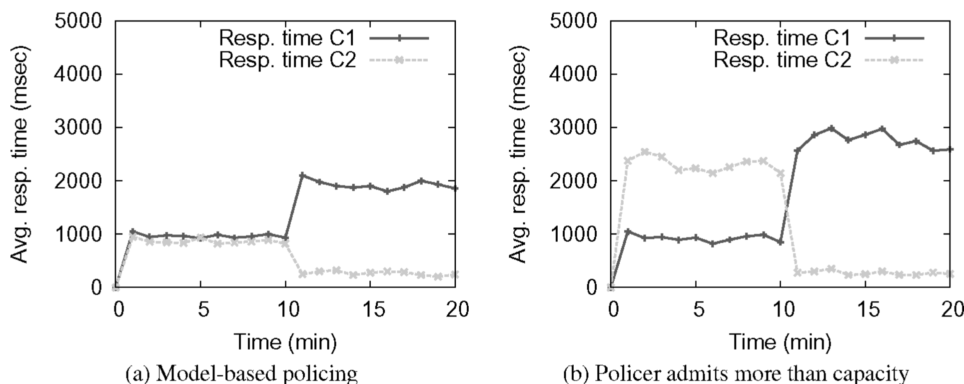
6.2 Session Policing and Class-Based Differentiation

Internet applications are known to experience unexpected surges in their workload, known as *flash crowds* [Welsh and Culler 2003]. Therefore, an important component of any such application is a sentry that 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. In an application supporting multiple classes of sessions, with possibly different response time requirements and revenue schemes for different classes, it is desirable to design a sentry that, during a flash crowd, can determine a subset of sessions admitting which would optimize a meaningful metric. An example of such a metric could be the overall expected revenue generated by the admitted sessions while meeting their response time targets (this constraint on response times will be assumed to hold in the rest of our discussion without being stated). Formally, given L session classes, $C_1, \ldots, C_L$, with up to N_i sessions of class C_i and using overall revenue as the metric to be optimized, the goal of the sentry is to determine an L-tuple $(N_1^{admit}, \ldots, N_L^{admit})$ such that

$$\forall n_i \leq N_i (1 \leq i \leq L), \ \sum_i rev_i\left(N_i^{admit}\right) \geq \sum_i rev_i(n_i),$$

where $rev_i(n_i)$ denotes the revenue generated by n_i admitted sessions of C_i.

Although a lot of research has been conducted on policing [Kasera et al. 2005; Abdelzaher and Bhatti 1999; Voigt et al. 2001; Welsh and Culler 2003; Abdelzaher et al. 2002; Cherkasova and Phaal 2002; Kanodia and Knightly 2000; Li and Jamin 2000; Elnikety et al. 2004; Kamra et al. 2004; Schroeder and Harchol-Balter 2003; Fox et al. 1997; Iyer et al. 2000; Appleby et al. 2001; Lassettre et al. 2003; Li and Jamin 2000; Chase and Doyle 2001; Urgaonkar and Shenoy 2004; Verma and Ghosal 2003], most of this work does not consider or extend



(a) Model-based policing (b) Policer admits more than capacity

Fig. 15. Maximizing revenue via differentiated session policing in Rubis. The application serves two classes of sessions.

to multitiered applications servicing multiple classes of clients. Our multiclass model described in Section 4.3 provides a flexible procedure for realizing this. First observe that the inputs to this procedure are the workload characteristics of various classes and the capacity assigned to the application tiers, and the desired output is the number of sessions of each class to admit. In theory, we could use the multiclass MVA algorithm to determine the revenue yielded by every admissible L-tuple. Clearly this would be computationally prohibitive. Instead, we use a heuristic that considers the session classes in a nonincreasing order of their revenue-per-session. For the class under consideration, it adds sessions until either all available sessions are exhausted or adding another session would cause the response time of at least one class, as predicted by the model, to violate its target. The outcome of this procedure is an L-tuple of the number of sessions that can be used by the policer to make admission control decisions.

We now describe our experiments to demonstrate the working of the session policer for Rubis. We configured the servlets version of Rubis with 2 replicas of the servlets tier. Similar to Section 4.3, we chose $W1$ and $W2$ to construct two session classes C_1 and C_2, respectively. The response time targets for the two classes were chosen to be 1 second and 2 seconds; the revenue yielded by each admitted session was assumed to be \$0.1 and \$1, respectively. We assume session durations of exactly 10 minutes for illustrative purposes. We create the following flash crowd scenarios. We assume that 150 sessions of C_1 and 10 sessions of C_2 arrive at $t = 0$; 50 sessions each of C_1 and C_2 are assumed to arrive at $t = 10$ minutes. Figure 15(a) presents the working of our model-based policer. At $t = 0$, based on the procedure described previously, the policer first admits all 10 sessions of the class with higher revenue-per-session, namely, C_2; it then proceeds to admit as many sessions of C_1 as it can (90), while keeping the average response times under target. At $t = 10$ minutes, the policer first admits as many sessions of C_2 as it can (21); it then admits 5 sessions of C_1; admitting more would, according to the model, cause the response time of C_2 to be violated. We find from Figure 15(a) that the response time requirements of both the classes are met during the experiment. We make two additional observations: (i) during [0, 10] minutes, the response time of C_2 is well below

its target of 2 seconds. This is because there are only 10 sessions of this class, less than the capacity of the database tier for the desired response time target. Since the 90 sessions of C_1 mainly stress the servlets tier (recall the nature of $W1$ and $W2$), they have minimal impact on the response time of C_2 sessions, which mainly exercise the database tier, and (ii) during (10, 20] minutes, the response time of C_1 is well below its target of 1 second. This is because the policer admits only 5 C_1 sessions; the servlets tier is lightly loaded since the C_2 sessions do not stress it, and, therefore, the C_1 sessions experience low response times.

Figure 15(b) demonstrates the impact of admitting more sessions on application response time. At $t = 0$, the policer admits excess C_1 sessions; it admits 140 and 10 sessions, respectively. We find that sessions of C_1 experience degraded response times (in excess of 2 seconds as opposed to the desired 1 second). Similarly, at $t = 10$ minutes, it admits excess C_2 sessions; it admits 5 and 31 sessions, respectively. Now sessions of C_2 experience response time violations. Observe that admitting excess sessions of one class does not cause a perceptible degradation in the performance of the other class because they exercise different tiers of the application.

7. CONCLUSIONS

In this article, we presented an analytical model for multitier Internet applications. Our model is based on using a network of queues to represent how the tiers in a multitier application cooperate to process requests. Our model is (i) general enough to capture Internet applications with an arbitrary number of heterogeneous tiers, (ii) is inherently designed to handle session-based workloads, and (iii) can account for application idiosyncrasies such as load imbalances within a replicated tier, caching effects, the presence of multiple classes of sessions, and limits on the amount of concurrency at each tier. The model parameters are easy to measure and update. We validated the model using two open-source multitier applications running on a Linux-based server cluster. Our experiments demonstrated that our model faithfully captures the performance of these applications for a variety of workloads and configurations. Furthermore, our model successfully handles a comprehensive range of resource utilization—from 0 to near saturation for the CPU—for two separate tiers. We demonstrated the utility of our model in managing resources for Internet applications under varying workloads and shifting bottlenecks. In one scenario, where the request arrival rate to the online auction application Rubis increased from 1500 to nearly 4200 requests/minute, a dynamic provisioning technique based on our model was able to maintain the response time target by quickly increasing the capacity of the Web tier and the Java servlets tier by factors of 2 and 3.5, respectively.

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, Canada.

Abdelzaher, T., Shin, K. G., and Bhatti, N. 2002. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Trans. Parall. Distrib. Syst. 13*, 1.

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.

Benani, M. and Menasce, D. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC'05)*. Seattle, WA.

Chandra, A., Gong, W., and Shenoy, P. 2003. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 11th International Workshop on Quality of Service (IWQoS'03)*. Monterey, CA.

Chandra, A., Goyal, P., and Shenoy, P. 2003. Quantifying the benefits of resource multiplexing in on-demand data centers. In *1st Workshop on Algorithms and Architectures for Self-Managing Systems*.

Chase, J., Anderson, D., Thakur, P., and Vahdat, A. 2001. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*.

Chase, J. and Doyle, R. 2001. Balance of power: Energy management for server clusters. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Elmau, Germany.

Chen, Y., Das, A., Gautam, N., Wang, Q., and Sivasubramaniam, A. 2004. Pricing-based strategies for autonomic control of web servers for time-varying request arrivals. *Engin. Applic. Arti. Intell.* Special Issue on Automatic Computing System *17*, 7 (Oct.), Elsevier, 841–854.

Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., and Gautam, N. 2005. Managing server energy and operational costs in hosting centers. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. Banff, Canada.

Cherkasova, L. and Phaal, P. 2002. Session-based admission control: A mechanism for peak load management of commercial Web sites. *IEEE Trans. Comput. 51*, 6 (June), 669–685.

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 6th USENIX Symposium in Operating Systems Design and Implementation (OSDI'04)*. San Francisco, CA.

Das, A., Datta, R., Sivasubramaniam, A., and Urgaonkar, B. 2006. Predicting Web cache behavior using stochastic state space models. Tech. rep., Department of Computer Science and Engineering, The Pennsylvania State University.

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 4th USITS*.

Elnikety, S., Nahum, E., Tracey, J., and Zwaenepoel, W. 2004. A method for transparent admission control and request scheduling in e-eommerce Web sites. In *Proceedings of the 13th International Conference on 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 16th Symposium on Operating Systems Principles (SOSP'97)*.

Franks, R. G. 1999. Performance analysis of distributed server systems. Ph.D. thesis, Carleton University.

Hellerstein, J., Zhang, F., and Shahabuddin, P. 1999. An approach to predictive detection for service management. In *Proceedings of the IEEE Intl. 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*.

Kallahalla, M., Uysal, M., Swaminathan, R., Lowell, D. E., Wray, M., Christian, T., Edwards, N., Dalton, C. I., and Gittler, F. 2004. SoftUDC: A software-based data center for utility computing. *IEEE Comput. 37*, 11, 38–46.

Kamra, A., Misra, V., and Nahum, E. 2004. Yaksha: A controller for managing the performance of 3-tiered Web sites. In *Proceedings of the 12th International Workshop on Quality of Service (IWQoS)*.

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

KASERA, S., PINHEIRO, J., LOADER, C., LAPORTA, T., KARAUL, M., AND HARI, A. 2005. Robust multi-class signaling overload control. In *Proceedings of IEEE International Conference on Networks Protocol (ICNP)*.

LASSETTRE, E., COLEMAN, D., DIAO, Y., FROELICH, S., HELLERSTEIN, J., HSIUNG, L., MUMMERT, T., RAGHAVACHARI, M., PARKER, G., RUSSELL, L., SURENDRA, M., TSENG, V., WADIA, N., AND YE, P. 2003. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*.

LAZOWSKA, E., ZAHORJAN, J., GRAHAM, G., AND SEVCIK, K. 1984. *Quantitative System Performance*. Prentice–Hall.

LEVY, R., NAGARAJARAO, J., PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. 2003. Performance management for cluster based web services. In *IFIP/IEEE 8th 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 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'00)*. 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. 7*.

MENASCE, D., ALMEIDA, V., AND DOWDY, L. 2004. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall.

RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. 2002. QoS-driven server migration for internet data centers. In *Proceedings of the 10th International Workshop on Quality of Service,* Miami, FL.

REISER, M. AND LAVENBERG, S. 1980. Mean-value analysis of closed multichain queuing networks. *J. ACM 27*, 2, 313–322.

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

ROLIA, J., SINGHAL, S., AND FRIEDRICH, R. 2000. Adaptive internet data centers. In *Proceedings of SSGRR 2000*.

S. KOUNEV, A. BUCHMANN. 2003. Performance modeling and evaluation of large-scale J2EE applications. In *Proceedings of the Computer Measurement Group's International Conference (CMG'03)*. Dallas, TX.

SCHROEDER, B. AND HARCHOL-BALTER, M. 2003. Web servers under overload: How scheduling can help. In *Proceedings of the 18th International Teletraffic Congress*.

SLOTHOUBER, L. 1996. A model of web server performance. In *Proceedings of the 5th International World Wide Web Conference*.

STEWART, C., KELLY, T., AND ZHANG, A. 2007. Exploiting nonstationarity for performance prediction. In *Proceedings of EuroSys 2007*. Lisbon, Portugal.

URGAONKAR, B. 2005. Dynamic resource management in internet data centers. Ph.D. thesis, University of Massachusetts, Amherst, MA.

URGAONKAR, B. AND SHENOY, P. 2004. Cataclysm: Handling extreme overloads in internet services. In *Proceedings of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'04)*. St. John's, Newfoundland, Canada.

URGAONKAR, B., SHENOY, P., CHANDRA, A., AND GOYAL, P. 2005. Dynamic provisioning of multitier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC'05)*. Seattle, WA.

URGAONKAR, B., SHENOY, P., AND ROSCOE, T. 2002. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*. 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 (WWW'03)*. Budapest, Hungary.

VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. 2004. Provisioning servers in the application tier for e-commerce systems. In *Proceedings of the 12th International Workshop on Quality of Service (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 4th USENIX Conference on Internet Technologies and Systems (USITS'03)*.

WOODSIDE, C. AND RAGHUNATH, G. 1995. General bypass architecture for high-performance distributed algorithms. In *Proceedings of the 6th IFIP Conference on Performance of Computer Networks*. Istanbul, Turkey.

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

XU, M. AND XU, C. 2004. Decay function model for resource configuration and adaptive allocation on internet servers. In *Proceedings of the 12th International Workshop on Quality-of-Service (IWQoS'04)*.

ZHU, X. AND SINGHAL, S. 2001. Optimal resource assignment in internet data centers. In *Proceedings of the 9th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'01)*.

Received August 2006; revised January 2007; accepted January 2007