

# Cataclysm: Policing Extreme Overloads in Internet Applications<sup>★</sup>

Bhuvan Urgaonkar<sup>a,\*</sup>

Prashant Shenoy<sup>b</sup>

<sup>a</sup>*Department of CSE, The Pennsylvania State University, University Park, PA, 16802.*

*Tel: +1 814 865 9506*

*Fax: +1 814 865 3176*

<sup>b</sup>*Department of Computer Science, University of Massachusetts, Amherst, MA, 01002.*

---

## Abstract

In this paper we present the Cataclysm server platform for handling extreme overloads in hosted Internet applications. The primary contribution of our work is to develop a low overhead, highly scalable admission control technique for Internet applications. Cataclysm provides several desirable features, such as guarantees on response time by conducting accurate size-based admission control, revenue maximization at multiple time-scales via preferential admission of important requests and dynamic capacity provisioning, and the ability to be operational even under extreme overloads. Cataclysm can transparently trade-off the accuracy of its decision making with the intensity of the workload allowing it to handle incoming rates of several tens of thousands of requests/second. We implement a prototype Cataclysm hosting platform on a Linux cluster and demonstrate the benefits of our integrated approach using a variety of workloads.

*Key words:* Internet application, Data center, Overload, Admission control, Dynamic provisioning, Sentry

---

---

<sup>★</sup> A preliminary version of this paper appeared in the Proceedings of the Fourteenth International World Wide Web Conference (WWW 2005), Chiba, Japan, May 2005 [31].

<sup>\*</sup> Corresponding author.

*Email addresses:* `bhuvan@cse.psu.edu` (Bhuvan Urgaonkar), `shenoy@cs.umass.edu` (Prashant Shenoy).

## 1 Introduction

During the past decade, there has been a dramatic increase in the popularity of Internet applications such as online news, online auctions, and electronic commerce. It is well known that the workload seen by Internet applications varies over multiple time-scales and often in an unpredictable fashion [12]. Certain workload variations such as time-of-day effects are easy to predict and handle by appropriate capacity provisioning [16]. Other variations such as flash crowds are often unpredictable. On September 11th 2001, for instance, the workload on a popular news Web site increased by an order of magnitude in thirty minutes, with the workload doubling every seven minutes in that period [12]. Similarly, the load on e-commerce retail Web sites can increase dramatically during the final days of the popular holiday season.

In this paper, we focus on handling extreme overloads seen by Internet applications. Informally, an extreme overload is a scenario where the workload unexpectedly increases by up to an order of magnitude in a few tens of minutes. Our goals are (i) to design a system that remains operational even in the presence of an extreme overload and even when the incoming request rate is several times greater than system capacity, and (ii) to maximize the revenue due to the requests serviced by the application during such an overload. We assume that Internet applications or services run on a *hosting platform*—essentially a server cluster that rents its resources to applications. Application providers pay for server resources, and in turn, are provided performance guarantees, expressed in the form of a *service level agreement (SLA)*. A hosting platform can take one or more of three actions during an overload: (i) add capacity to the application by allocating idle or under-used servers, (ii) turn away excess requests and preferentially service only “important” requests, and (iii) degrade the performance of admitted requests in order to service a larger number of aggregate requests.

We argue that a comprehensive approach for handling extreme overloads should involve a combination of all of the above techniques. A hosting platform should, whenever possible, allocate additional capacity to an application in order to handle increased demands. The platform should degrade performance in order to temporarily increase effective capacity during overloads. When no capacity addition is possible or when the SLA does not permit any further performance degradation, the platform should turn away excess requests. While doing so, the platform should preferentially admit important requests and turn away less important requests to maximize overall revenue. For instance, small requests may be preferred over large requests, or financial transactions may be preferred over casual browsing requests.

We present the design of the Cataclysm server platform to achieve these goals. Cataclysm is specifically designed to handle extreme overloads in Internet applications and differs from past work in two significant respects.

First, since an extreme overload may involve request rates that are *an order of magnitude greater than the currently allocated capacity*, the admission controller must be able to quickly examine requests and discard a large fraction of these requests, when necessary, with minimal overheads. Thus, the efficiency of the admission controller is important during heavy overloads. To address this issue, we propose very low overhead admission control mechanisms that can scale to very

high request rates under overloads. Past work on admission control [10,14,33,36] has focused on the mechanics of policing and did not specifically consider the *scalability* of these mechanisms. In addition to imposing very low overheads, our mechanisms can preferentially admit important requests during an overload and transparently trade-off the accuracy of their decision making with the intensity of the workload. The trade-off between accuracy and efficiency is another contribution of our work and enables our implementation to scale to incoming rates of up to a few tens of thousands of requests/s.

Second, our dynamic provisioning mechanism employs a G/G/1-based queuing model of a replicable application in conjunction with online measurements to dynamically vary the number of servers allocated to each application. A novel feature of our platform is its ability to not only vary the number of servers allocated to an application but also other components such as the admission controller and the load balancing switches. Dynamic provisioning of the latter components has not been considered in prior work.

We have implemented a prototype Cataclysm hosting platform on a cluster of twenty Linux servers. We demonstrate the effectiveness of our integrated overload control approach via an experimental evaluation. Our results show that (i) preferentially admitting requests based on importance and size can increase the utility and effective capacity of an application, (ii) our admission control is highly scalable and remains functional even for arrival rates of a few thousand requests/s, and (iii) our solution based on a combination of admission control and dynamic provisioning is effective in meeting response time targets and improving platform revenue.

The rest of this paper is organized as follows. Section 2 provides an overview of the proposed system. Sections 3 and 4 describe the mechanisms that constitute our overload management solution. Section 5 describes the implementation of our prototype. In Section 6 we present the results of our experimental evaluation. Section 7 presents related work and Section 8 concludes this paper.

## 2 System Overview

In this section, we present the system model for our Cataclysm hosting platform and the model assumed for Internet applications running on the platform.

### 2.1 Cataclysm Hosting Platform

The Cataclysm hosting platform consists of a cluster of commodity servers interconnected by a modern LAN technology such as Gigabit Ethernet. One or more high bandwidth links connect this cluster to the Internet. Each node in the hosting platform can take on one of three roles: cataclysm server, cataclysm sentry, or cataclysm control plane (see Figure 1).

*Cataclysm Servers:* Cataclysm servers are nodes that run Internet applications. The hosting platform may host multiple applications concurrently. Each application is assumed to run on a subset

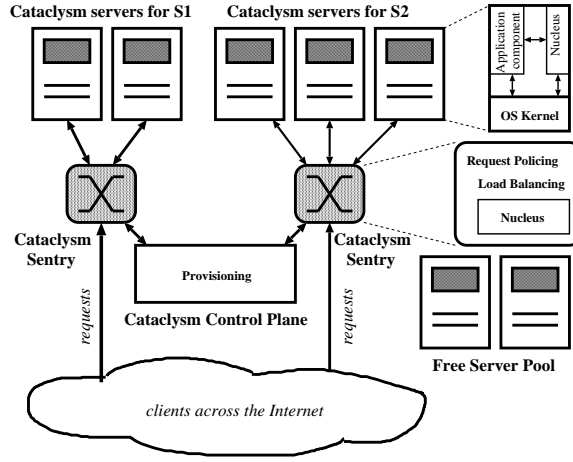


Fig. 1. The Cataclysm Hosting Platform Architecture.

of the nodes, and a node is assumed to run no more than one application at any given time. A subset of the servers may be unassigned and form the *free server pool*. The number of servers assigned to an application can change over time depending on its workload. Each server also runs the cataclysm nucleus—a software component that performs online measurements of application-specific resource usages, which are then conveyed to the other two components that we describe next.

*Cataclysm Sentry:* Each application running on the platform is assigned one or more sentries. A sentry guards the servers assigned to an application and is responsible for two tasks. First, the sentry polices all requests to an application’s server pool—incoming requests are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess requests are turned away during overloads. Second, each sentry implements a layer-7 switch that performs load balancing across servers allocated to an application. Since there has been substantial research on load balancing techniques for clustered Internet applications [26], we do not consider load balancing techniques in this work.

*Cataclysm Control Plane:* The control plane is responsible for dynamic provisioning of servers and sentries in individual applications. It tracks the resource usages on nodes, as reported by cataclysm nuclei, and determines the resources (in terms of the number of servers and sentries) to be allocated to each application. The control plane runs on a dedicated server and its scalability is not of concern in the design of our platform.

## 2.2 Model for Internet Applications

The Internet applications considered in this work are assumed to be inherently replicable. That is, the application is assumed to run on a cluster of servers, and it is assumed that running the application on a larger number of servers results in an effective increase in capacity. Many, but by no means all, Internet applications fall into this category. Vanilla clustered Web servers are an example of a replicable application. Multi-tiered Internet applications are partially replicable. A typical multi-tiered application has three components: a front-end HTTP server, a middle-tier ap-

Arrival rate (requests/sec)	Average response time for admitted requests
< 1000	1 sec
1000-10000	2 sec
> 10000	3 sec

Table 1

A sample service-level agreement.

plication server, and a back-end database server. The front-end HTTP server is easily replicable but is not necessarily the bottleneck. The middle-tier—a frequent bottleneck—can be implemented in different ways. One popular technique is to use server-side scripting such as Apache’s *php* functionality, or to use cgi-bin scripting languages such as *perl*. If the scripts are written carefully to handle concurrency, it is possible to replicate the middle-tier as well. More complex applications use Java application servers to implement the middle-tier. Dynamic replication of Java application servers is more complex and techniques for doing so are beyond the scope of this paper. Dynamic replication of back-end databases is an open research problem. Consequently, most dynamic replication techniques in the literature, including this work, assume that the database is sufficiently well provisioned and does not become a bottleneck even during overloads.

Given a replicable Internet application, we assume that the application specifies the desired performance guarantees in the form of a service level agreement (SLA). An SLA provides a description of the QoS guarantees that the platform will provide to the application. The SLA we consider in our work is defined as follows:

$$\text{Avg resp time } R \text{ of adm req} = \begin{cases} \mathcal{R}_1 & \text{if arrival rate} \in [0, \lambda_1) \\ \mathcal{R}_2 & \text{if arrival rate} \in [\lambda_1, \lambda_2) \\ \dots & \\ \mathcal{R}_k & \text{if arrival rate} \in [\lambda_{k-1}, \infty) \end{cases} \quad (1)$$

The SLA specifies the revenue that is generated by each request that meets its response time target. Table 1 illustrates an example SLA.

Each Internet application consists of  $L (L \geq 1)$  request classes:  $C_1, \dots, C_L$ . Each class has an associated revenue that an admitted request yields—requests of class  $C_1$  are assumed to yield the highest revenue and those of  $C_L$  the least. The number of request classes  $L$  and the function that maps requests to classes is application-dependent. To illustrate, an online brokerage Web site may define three classes and may map financial transactions to  $C_1$ , other types of requests such as balance inquiries to  $C_2$ , and casual browsing requests from non-customers to  $C_3$ . An application’s SLA may also specify lower bounds on the request arrival rates that its classes should always be able to sustain.

### 3 Cataclysm Sentry Design

In this section, we describe the design of a cataclysm sentry. The sentry is responsible for two tasks: request policing and load balancing. As indicated earlier, the load balancing technique used in the sentry is not a focus of this work, and we assume the sentry employs a layer-7 load balancing algorithm such as [26]. The first key issue that drives the design of the request policer is to maximize the revenue yielded by the admitted requests while providing the following notion of *class-based differentiation* to the application: each class should be able to sustain the minimum request rate specified for it in the SLA. Given our focus on extreme overloads, the design of the policer is also influenced by the second key issue of *scalability*: ensuring very low overhead admission control tests in order to scale to very high request arrival rates seen during overloads. This section elaborates on these two issues.

From a theoretical point of view, the problem of conducting admission control to maximize revenue while providing class-based differentiation can be shown to be NP-hard [15]. With this insight into the property of the admission control problem, we focus on designing a policer that provides the desired differentiation *with high probability*.

#### 3.1 Request Policing Basics

The sentry maps each incoming request to one of the classes  $C_1, \dots, C_L$ . The policer needs to guarantee to each class an admission rate equal to the minimum sustainable rate desired by it (recall our SLA from Section 2). It does so by implementing leaky buckets, one for each class, that admit requests conforming to these rates. Requests conforming to these leaky buckets are forwarded to the application. Leaky buckets can be implemented very efficiently, so determining if an incoming request confirms to a leaky bucket is an inexpensive operation. Requests in excess of these rates undergo further processing as follows. Each class has a queue associated with it (see Figure 2); incoming requests are appended to the corresponding class-specific queue. Requests within each class can be processed either in FIFO order or in order of their service times. In the former case, all requests within a class are assumed to be equally important, whereas in the latter case smaller requests are given priority over larger requests within each class. Admitted requests are handed to the load balancer, which then forwards them to one of the cataclysm servers in the application's server pool.

The policer incorporates the following two features in its processing of the requests that are in excess of the guaranteed rates to maximize revenue.

- (1) The policer introduces *different amounts of delay* in the processing of newly arrived requests belonging to different classes. Specifically, requests of class  $C_i$  are processed by the policer once every  $d_i$  time units ( $d_1 = 0 \leq d_2 \leq \dots \leq d_L$ ); requests arriving during successive processing instants wait for their turn in their class-specific queues. These delay values, determined periodically, are chosen to reduce the chance of admitting less important requests into the system when they are likely to deny service to more important requests that arrive shortly thereafter. In

Section 3.4 we show how to pick these delay values such that the probability of a less important request being admitted into the system and denying service to a more important request that *arrives later* remains sufficiently small.

- (2) The policer processes queued requests in the decreasing order of importance: requests in  $C_1$  are subjected to the admission control test first, and then those in  $C_2$  and so on. Doing so ensures that requests in class  $C_i$  are given higher priority than those in class  $C_j$ ,  $j > i$ . The admission control test, which is described in detail in the next section, admits requests so long as the system has sufficient capacity to meet the contracted SLA. Note that, if requests in a certain class  $C_i$  fail the admission control test, all queued requests in less important classes can be rejected without any further tests.

Observe that the above admission control strategy meets one of our two goals: it preferentially admits only important requests during an overload and turns away less important requests. However, the strategy needs to invoke the admission control test on each individual request, resulting in a complexity of  $O(r)$ , where  $r$  is the number of queued up requests. Further, when requests within a class are examined in order of service times instead of FIFO, the complexity increases to  $O(r \cdot \log(r))$  due to the need to sort requests. Since the incoming request rate can be substantially higher than capacity during an extreme overload, running the admission control test on every request or sorting requests prior to admission control may be simply infeasible. Consequently, in what follows, we present two strategies for very low overhead admission control that scale well during overloads.

We note that a newly arriving request imposes two types of computational overheads on the policer: (i) protocol processing and (ii) the admission control test itself. Clearly, both these components need to scale for effective handling of overloads. When protocol processing starts becoming bottleneck, we respond by increasing the number of sentries guarding the overloaded application, a technique that we describe in detail in Section 4.2. In this section we present techniques to deal with the scalability of the admission control test.

### 3.2 Efficient Batch Processing

One possible approach for reducing the policing overhead is to process requests in *batches*. Request arrivals tend to be very bursty during severe overloads, with a large number of requests arriving in a short duration of time. These requests are queued up in the appropriate class-specific queues at the sentry. Our technique exploits this feature by conducting a single admission control test on an entire batch of requests within a class, instead of doing so for each individual request. Such batch processing can amortize the admission control overhead over a larger number of requests, especially during overloads.

To perform efficient batch-based admission control, we define  $b$  buckets within each request class. Each bucket has a range of request service times associated with it. The sentry estimates the service time of a request and then maps it into the bucket corresponding to that service time. To illustrate, a request with an estimated service time in the range  $(0, s_1]$  is mapped to bucket 1, that with service time in the range  $(s_1, s_2]$  to bucket 2, and so on. By defining an appropriate function, mapping a

request to a bucket can be implemented efficiently as a constant time operation.

Bucket-based mapping is motivated by two reasons. First, it groups requests with similar service times and enables the policer to conduct a single admission control test by assuming that all requests in a bucket impose similar service demands. Second, since successive buckets contain requests with progressively larger service times, the technique implicitly gives priority to smaller requests. Moreover, no sorting of requests is necessary: the mapping implicitly “sorts” requests when mapping them into buckets.

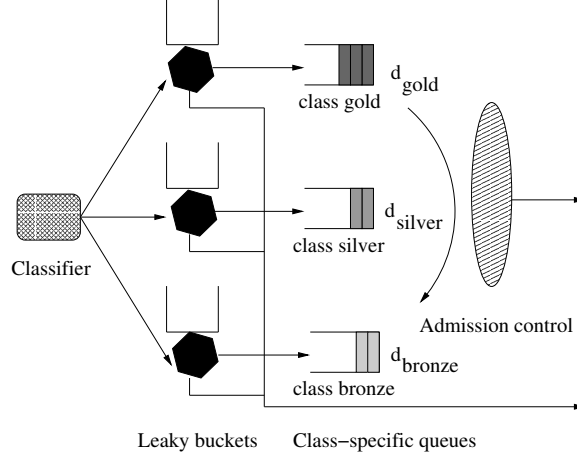


Fig. 2. Working of the cataclysm sentry. First, the class a request belongs to is determined. If the request confirms to the leaky bucket for its class, it is admitted to the application without any further processing. Otherwise, it is put into its class-specific queue. The admission control processes the requests in various queues at frequencies given by the class-specific delays. A request is admitted to the application if there is enough capacity, else it is dropped.

When the admission control is invoked on a request class, it considers each non-empty bucket in that class and conducts a single admission control test on all requests in that bucket (i.e., all requests in a bucket are treated as a batch). Consequently, no more than  $b$  admission control tests are needed within each class, one for each bucket. Since there are  $L$  request classes, this reduces the admission control overhead to  $O(b \cdot L)$ , which is substantially smaller than the  $O(r)$  overhead for admitting individual requests.

Having provided the intuition behind batch-based admission control, we discuss the mapping process and the admission control test in detail. In order to map a request into a bucket, the sentry must first estimate the *inherent service time* of that request. The inherent service time of a request is the time needed to service the request on a lightly loaded server (i.e., when the request does not see any queuing delays). The inherent service time of a request  $\mathcal{R}$  is defined to be

$$S_{inherent} = \mathcal{R}_{cpu} + \alpha \cdot \mathcal{R}_{data} \quad (2)$$

where  $\mathcal{R}_{cpu}$  is the total CPU time needed to service  $\mathcal{R}$ ,  $\mathcal{R}_{data}$  is the IO time of the request (which includes the time to fetch data from disk, the time the request is blocked on a database query, the network transfer time, etc.), and  $\alpha$  is an empirically determined constant. The inherent service time



is then used to map the request into an appropriate bucket—the request maps to a bucket  $i$  such that  $s_i \leq S_{inherent} \leq s_{i+1}$ .

The specific admission control test for each batch of requests within a bucket is as follows. Let  $\beta$  denote the batch size (i.e., the number of requests) in a bucket. Let  $Q$  denote the estimated queuing delay seen by each request in the batch. The queuing delay is the time the request has to wait at a cataclysm server before it receives service; the queuing delay is a function of the current load on the server and its estimation is discussed in Section 3.5. Let  $\eta$  denote the average number of requests (connections) that are currently being serviced by a server in the application’s server pool. Then the  $\beta$  requests within a batch are admitted if and only if the sum of the queuing delay seen by a request and its actual service time does not exceed the contracted SLA. That is,

$$Q + \left( \eta + \left\lceil \frac{\beta}{n} \right\rceil \right) \cdot S \leq R_{sla} \quad (3)$$

where  $S$  is the average inherent service time of a request in the batch,  $n$  is the number of servers allocated to the application, and  $R_{sla}$  is the desired response time. The term  $\left( \eta + \left\lceil \frac{\beta}{n} \right\rceil \right) \cdot S$  is an estimate of the *actual* service time of the *last* request in the batch, and is determined by scaling the inherent service time  $S$  by the server load—which is the number of the requests currently in service, i.e.,  $\eta$ , plus the number of requests from the batch that might be assigned to the server i.e.,  $\left\lceil \frac{\beta}{n} \right\rceil$ .<sup>1</sup> Rather than actually computing the mean inherent service time of the request in a batch, it is approximated as  $S = (s_i + s_{i+1})/2$ , where  $(s_i, s_{i+1}]$  is the service time range associated with the bucket.

As indicated above, the admission control is invoked for each class periodically—once every  $d_i$  time units for newly arrived requests of class  $C_i$ . The invocation is more frequent for important classes and less frequent for less important classes, that is,  $d_1 = 0 \leq d_2 \leq \dots \leq d_L$ . Since a request may wait in a bucket for up to  $d_i$  time units before admission control is invoked for its batch, the above test is modified as

$$Q + \left( \eta + \left\lceil \frac{\beta}{n} \right\rceil \right) \cdot S \leq R_{sla} - d_i \quad (4)$$

In the event this condition is satisfied, all requests in the batch are admitted into the system. Otherwise requests in the batch are dropped.

Observe that introducing these delays into the processing of certain requests *does not cause a degradation in the response time* of the admitted requests because they now undergo a more stringent admission control test as given by (4). However, these delays would have the effect of reducing the application’s throughput when it is not overloaded. Therefore, these delays should be changed dynamically as workloads of various classes change. In particular, they should tend to 0 when the

---

<sup>1</sup> Note that we have made the assumption of perfect load balancing in the admission control test (3). One approach for capturing load imbalances can be to scale  $\eta$  and  $n$  by suitably chosen skew factors. These skew factors can be based on measurements of the load imbalance among the replicas of the application.

application has sufficient capacity to handle all the incoming traffic. We discuss in Section 3.4 how these delay values are dynamically updated. Techniques for estimating parameters such as the queuing delay, inherent service time, and the number of existing connections are discussed in Section 3.5.

### 3.3 Scalable Threshold-based Policing

We now present a second approach to further reduce the policing overhead. Our technique trades efficiency of the policer for accuracy and reduces the overhead to a few arithmetic operations per request. The key idea behind this technique is to periodically *pre-compute* the fraction of arriving requests that should be admitted in each class and then simply enforce these limits without conducting any additional per-request tests. Again, incoming requests are first classified and undergo an inexpensive test to determine if they confirm to the leaky buckets for their classes. Confirming requests are admitted to the application without any further tests. Other requests undergo a more lightweight admission control test that we describe next.

Our technique uses estimates of future arrival rates and service demands in each class to compute a *threshold*, which is defined to be a pair (class  $i$ , fraction  $p_{admit}$ ). The threshold indicates that all requests in classes more important than  $i$  should be admitted ( $p_{admit} = 1$ ), requests in class  $i$  should be admitted with probability  $p_{admit}$ , and all requests in classes less important than  $i$  should be dropped ( $p_{admit} = 0$ ). We determine these parameters based on observations of arrival rates and service times in each classes over periods of moderate length (we use periods of length 15 sec). Denoting the arrival rates to classes  $1, \dots, L$  by  $\lambda_1, \dots, \lambda_L$  and the observed average service times by  $s_1, \dots, s_L$ , the threshold  $(i, p_{admit})$  is computed such that

$$\sum_{j=1}^{j=i} \lambda_j s_j \geq 1 - \sum_{j=1}^{j=L} \lambda_j^{min} s_j \quad (5)$$

and

$$p_{admit} \cdot \lambda_i s_i + \sum_{j=1}^{j=i-1} \lambda_j s_j < 1 - \sum_{j=1}^{j=L} \lambda_j^{min} s_j \quad (6)$$

where  $\lambda_j^{min}$  denotes the minimum guaranteed rate for class  $j$ .

Thus, admission control now merely involves applying the inexpensive classification function on a new request to determine its class, determining if it confirms to the leaky bucket for that class (also a lightweight operation), and then using the equally lightweight thresholding function (if it does not confirm to the leaky bucket) to decide if it should be admitted. Observe that this admission control requires estimates of per-class arrival rates. These rates are clearly difficult to predict during unexpected overloads. However, it is possible to react fast by updating our estimates of the arrival rates frequently. Our implementation of threshold-based policing estimates arrival rates by

computing exponentially smoothed averages of arrivals over 15 sec periods. We will demonstrate the efficacy of this policer in an experiment in Section 6.3.

The threshold-based and the batch-based policing strategies need not be mutually exclusive. The sentry can employ the more accurate batch-based policing so long as the incoming request rate permits one admission control test per batch. If the incoming rate increases significantly, the processing demands of the batch-based policing may saturate the sentry. In such an event, when the load at the sentry exceeds a threshold, the sentry can trade accuracy for efficiency by dynamically switching to a threshold-based policing strategy. This ensures greater scalability and robustness during overloads. The sentry reverts to the batch-based admission control when the load decreases and stays below the threshold for a sufficiently long duration. We would like to note that several existing admission control algorithms such as [14,19,36] (discussed in Section 7) are based on dynamically set thresholds such as admission rates and can be implemented as efficiently as our threshold-based admission control. The novel feature in our approach is the flexibility to trade-off the accuracy of admission control for its computational overhead depending on the load on the sentry.

### 3.4 Analysis of the Policer

In this section we show how the sentry can, under certain assumptions, compute the delay values for various classes based on online observations. The goal is to pick delay values such that the *probability of a newly arrived request being denied service due to an already admitted less important request* is smaller than a desired threshold.

Consider the following simplified version of the admission control algorithm presented in Section 3.2: Assume that the application runs on only one server. It is easy to extend the analysis to the case of multiple servers. The admission controller lets in a new request if and only if the total number of requests that have been admitted and are being processed by the application does not exceed a threshold  $N$ . Assume the application consists of  $L$  request classes  $C_1, \dots, C_L$  in decreasing order of importance. We make the simplifying assumption of Poisson arrivals with rates  $\lambda_1, \dots, \lambda_L$ , and service times with known CDFs  $F_{s_1}(\cdot), \dots, F_{s_L}(\cdot)$  respectively. As before,  $d_1 = 0$ . For simplicity of exposition we assume that the delay for class  $C_2$  is  $d$ , and  $\forall i > 2, d_{i+1} = k_i \cdot d_i, (k_i \geq 1)$ . Denote by  $A_i$  the event that a request of class  $C_i$  has to be dropped at the processing instant  $m \cdot d_i, (m > 0)$  and there is at least one request of a less important class  $C_j, (j > i)$  still in service. Clearly,

$$Pr(A_1) = 0 \text{ and } Pr(A_L) = 0.$$

We are interested in ensuring

$$\forall i > 1, Pr(A_i) < \epsilon, 0 < \epsilon < 1. \tag{7}$$

Consider  $1 < i < L$ . For  $A_i$  to occur, all of the following must hold: (1)  $X_i$ : at least one request of class  $C_i$  arrives during the period  $[(m-1) \cdot d_i, m \cdot d_i]$ , (2)  $Y_i$ : the number of requests in service

at time  $m \cdot d_i$  is  $N$ , (3)  $Z_i$ : at least one of the requests being serviced belongs to one of the classes  $C_{i+1}, \dots, C_L$ . We have,

$$Pr(A_i) = Pr(X_i \wedge Y_i \wedge Z_i)$$

During overloads, we can assume that the number of requests in service would be  $N$  with a high probability  $p_{drop}$ . The policer will record  $p_{drop}$  over short periods of time. Also,  $X_i$  and  $Z_i$  are independent. This lets us have

$$Pr(A_i) \approx Pr(X_i) \cdot p_{drop} \cdot Pr(Z_i) \quad (8)$$

$$Pr(X_i) = 1 - e^{-\lambda_i \cdot d_i} \quad (9)$$

Denote by  $Z_i^j$ , ( $i < j \leq L$ ) the event that at least one of the requests being serviced at time  $m \cdot d_i$  belongs to the class  $j$ . Clearly,

$$Pr(Z_i) = \sum_{j=i+1}^{j=L} Pr(Z_i^j) \quad (10)$$

Let us now focus on the term  $Pr(Z_i^j)$ . The event  $Z_i^j$  is the disjunction of the following events, one for each  $l$ , ( $l > 0$ ):  $P_j^l$ : at least one request of class  $j$  arrives during the period  $[m \cdot d_i - (l + 1) \cdot d_j, m \cdot d_i - l \cdot d_j]$  and  $Q_j^l$ : at least one request of class  $j$  is admitted at the processing instant  $m \cdot d_i - l \cdot d_j$  and  $R_j^l$ : the service time of at least one admitted request is long enough so that it is still in service at time  $m \cdot d_i$ . As in Equation (9),

$$Pr(P_j^l) = 1 - e^{-\lambda_j \cdot d_j} \quad (11)$$

Consider  $R_j^l$ . During an overload each admitted request competes at the server with  $(N-1)$  other requests during most of its lifetime. A fair approximation then is to assume that a request takes  $N$  times its service time to finish. Therefore, we have,

$$Pr(R_j^l) = 1 - F_{s_j} \left( \frac{l \cdot d_j}{N} \right) \quad (12)$$

We approximate  $Q_j^l$  using the following reasoning. During overloads, a request of class  $C_j$  will be admitted at processing instant  $t$  only if the number of requests in service at time  $t$  is less than  $N$  (the probability of this is approximated as  $(1 - p_{drop})$ ) and no request of a more important class  $C_h$  arrived during  $[t - d_h, t]$ . That is,

$$Pr(Q_j^l) \approx (1 - p_{drop}) \prod_{h=1}^{h=j-1} e^{-\lambda_h d_h} \quad (13)$$

Combining Equations (8)-(13), we get the following approximation.  $Pr(A_i)$ ,

$$Pr(A_i) \approx p_{drop}(1 - p_{drop})(1 - e^{-\lambda_i d_i}) \cdot \sum_{j=i+1}^{j=L} (1 - e^{-\lambda_j d_j}) \prod_{h=1}^{h=j-1} e^{-\lambda_h d_h} \sum_{l=1}^{\infty} \left(1 - F_{s_j} \left(\frac{l d_j}{N}\right)\right)$$

The above approximation of  $Pr(A_i)$  provides a procedure for iteratively computing the  $d_i$  values using numerical methods. We pick delay values that make the term on the right hand side smaller than the desired bound  $\epsilon$  for all  $i$ . This in turn guarantees that the inequalities in (7) are satisfied.

### 3.5 Online Parameter Estimation

The batch-based and threshold-based policing algorithms require estimates of a number of system parameters. These parameters are estimated using online measurements. The nuclei running on the cataclysm servers and sentries collectively gather and maintain various statistics needed by the policer. The following statistics are maintained:

- *Arrival rate  $\lambda_i$* : Since each request is mapped onto a class at the sentry, it is trivial to use this information to measure the incoming arrival rates in each class.
- *Queuing delay  $Q$* : The queuing delay incurred by a request is measured at the cataclysm server. The queuing delay is estimated as the difference between the time the request arrives at the server and the time it is accepted by the HTTP server for service (we assume that the delay incurred at the sentry is negligible). The nuclei can measure these values by appropriately instrumenting the operating system kernel. The nuclei periodically report the observed queuing delays to the sentry, which then computes the mean delays across all servers in the application's pool.
- *Number of requests in service  $\eta$* : This parameter is measured at the cataclysm server. The nuclei track the number of active connections serviced by the application and periodically report the measured values to the sentry. The sentry then computes the mean of the reported values across all servers for the application.
- *Request service time  $s$* : This parameter is also measured at the server. The actual service time of a request is measured as the difference between the arrival time at the server and the time at which the last byte of the response is sent. The measurement of the inherent service time is more complex. Doing so requires instrumentation of the OS kernel and some instrumentation of the application itself. This instrumentation enables the nucleus to compute the CPU processing time for a request as well as the duration for which the requested is blocked on I/O. Together, these values determine the inherent service time (see Equation 2).
- *Constant  $\alpha$* : The constant  $\alpha$  in Equation 2 is measured using offline measurements on the cataclysm servers. We execute several requests with different CPU demands and different-sized responses under light load conditions and measure their execution times. We also compute the CPU demands and the I/O times as indicated above. The constant  $\alpha$  is then estimated as the value that minimizes the difference between the actual execution time and the inherent service time in Eq. 2.

The sentry uses past statistics to estimate the inherent service time of an incoming request in order to map it onto a bucket. To do so, the sentry uses a hash table for maintaining the usage statistics

for the requests it has admitted so far. Each entry in this table consists of the requested URL (which is used to compute the index of the entry in the table) and a vector of the resource usages for this request as reported by the various servers. Requests for static content possess the same URL every time and so always map to the same entry in the hash table. The URL for requests for dynamic content, on the other hand, may change (e.g. the arguments to a script may be specified as part of the URL). For such requests, we get rid of the arguments and hash based on the name of the script invoked. The resource usages for requests that invoke these scripts may change depending on the arguments. We maintain exponentially decayed averages of their usages.

## 4 Provisioning for Cataclysms

Policing mechanisms may turn away a significant fraction of the requests during overloads. In such a scenario, an increase in the effective application capacity is necessary to reduce the request drop rate. The cataclysm control plane implements dynamic provisioning to vary the number of allocated servers based on application workloads. The application's server pool is increased during overloads by allocating servers from the free pool or by reassigning under-used servers from other applications. The control plane can also dynamically provision sentry servers when the incoming request rate imposes significant processing demands on the existing sentries. The rest of this section discusses techniques for dynamically provisioning cataclysm servers and sentries.

### 4.1 Model-based Provisioning for Applications

We use queuing theory to model the revenue obtained from assigning a certain number of servers to a replicable application under a given workload. Our model does not make any assumptions about the nature of the request arrival processes or the service times. Our abstraction of a single replica of a service is a G/G/1 queuing system for which the following bound is known [21]:

$$\lambda \geq \left[ E[S] + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (E[R] - E[S])} \right]^{-1} \quad (14)$$

Here  $E[R]$  is the average response time,  $E[S]$  is the average service time,  $\lambda$  is the request arrival rate,  $\rho = \lambda \cdot E[S]$  is the utilization, and  $\sigma_a^2$  and  $\sigma_b^2$  are the variance of inter-arrival time and the variance of service time respectively. It should be pointed out that a number of similar models for simple, replicable applications have been proposed in recent work ([13,27,34]) and any of these could potentially be used by our provisioning algorithm. The comparison of our model with these other models is not relevant to our current discussion of overload management and therefore beyond the scope of this work. Modeling of more complex, multi-tiered Internet applications is part of our ongoing research.

Inequality (14) is used by the provisioning mechanism to determine the number of servers needed by an application to sustain a given request arrival rate. The dynamic provisioning mechanism

normally operates in a *predictive* fashion. It is invoked periodically (once every 30 minutes in our prototype) and uses the workload observations in the past to predict future request arrival rates. It then determines a partition of servers among the applications that would maximize the expected revenue during the next period. Since the focus of this paper is on scalable admission control, we omit the details of our workload prediction and server allocation algorithms here.

Since overloads are often unanticipated, a sentry of an overloaded application can dynamically invoke the provisioning mechanism whenever the request drop rate exceeds a certain pre-defined value. In such a scenario, the provisioning mechanism operates in a *reactive* mode to counter the overload. The mechanism monitors deviations in the actual workload from the predicted workload and uses these to detect overloaded applications. It allocates additional servers to the overloaded applications—these servers may be from the free server pool or underutilized servers from other applications. Undesirable oscillations in such allocations are prevented using two constraints: (i) a limit of  $\Delta$  is imposed on the number of servers that can be allocated to an application in a single step in the reactive mode and (ii) a delay of  $\delta$  time units is imposed on the duration between two successive invocations of the provisioning mechanism in the reactive mode ( $\delta$  is set to 5 minutes in our prototype). Recall that our SLA permits degraded response time targets for higher arrival rates. The provisioning mechanism may *degrade* the response time to the extent permitted by the SLA, add more capacity, or a bit of both. The optimization drives these decisions, and the resulting target response times are conveyed to the request policers. Thus, these interactions enable coupling of policing, provisioning, and adaptive performance degradation.

## 4.2 Sentry Provisioning

In general, allocation and deallocation of sentries is significantly less frequent than that of catalysms servers. Further, the number of sentries needed by an application is much smaller than the number of servers running it. Consequently, a simple provisioning scheme suffices for dynamically varying the number of sentries assigned to an application. Our scheme uses the CPU utilization of the existing sentry servers as the basis for allocating additional sentries (or deallocating active sentries). If the utilization of a sentry stays in excess of a pre-defined threshold  $high_{cpu}$  for a certain period of time, it requests the control plane for an additional sentry server. Upon receiving such requests from one or more sentries of an application, the control plane assigns each an additional sentry. Similarly, if the utilization of a sentry stays below a threshold  $low_{cpu}$ , it is returned to the free pool while ensuring that the application has at least one sentry remaining. Whenever the control plane assigns (or removes) a sentry server to an application, it repartitions the application's servers pool equally among the various sentries. The DNS entry for the application is also updated upon each allocation or deallocation; a round-robin DNS scheme is used to loosely partition incoming requests among sentries. Since each sentry manages a mutually exclusive pool of servers, it can independently perform admission control and load balancing on arriving requests; the SLA is collectively maintained by virtue of maintaining it at each sentry.

## 5 Implementation Considerations

We implemented a prototype Cataclysm hosting platform on a cluster of 20 Pentium machines connected via a 1 Gbps Ethernet switch and running Linux 2.4.20. Each machine in the cluster runs one of the following entities: (1) an application replica, (2) a cataclysm sentry, (3) the cataclysm provisioning, (4) a workload generator for an application. In this section we discuss our implementation of the cataclysm sentry and provisioning.

### 5.1 Cataclysm Sentry

We used *Kernel TCP Virtual Server* (`ktcpsvs`) version 0.0.14 [23] to implement the policing mechanisms described in Section 3. `ktcpsvs` is an open-source, Layer-7 load balancer implemented as a Linux module. It listens for client requests at a well-known port and distributes arriving requests among back end servers. It accepts TCP connections from clients, opens separate connections with servers (one for each client) and transparently relays data between these. We modified `ktcpsvs` to implement all the sentry mechanisms described in Sections 3 and 4.

`ktcpsvs` has a multi-threaded design, with a number of kernel threads waiting for new connections to arrive. Upon the arrival of a connection, one of the threads accepts it, opens a separate TCP connection with one of the servers and copies data back and forth between these connections. When this connection closes, the thread goes back to waiting for new arrivals. We modify `ktcpsvs` so it consists of three classes of kernel threads: (1) one *accepter thread*, (2) one *policer thread* per class supported by the service, and (3) a large number of *relayer threads*. The accepter thread accepts a newly arrived connection and creates a *connection object* containing information that would be needed for transferring data over this connection if it gets admitted. It determines the class that this request belongs to.<sup>2</sup> This object is then moved to the *wait queue* corresponding to the request's class. A wait queue is a FIFO queue used as an intermediate storage for accepted requests of a class before the admission control acts on them. The policer thread corresponding to a certain class wakes up once every processing interval for that class (Section 3, see Figure 2) and invokes the admission control test on the current batch of requests in that class's wait queue. The admitted connections are moved to the *admit queue*, a single FIFO queue for admitted connections. The connections dropped by the admission control are closed. The relayer threads wait for connections arriving into the admit queue. After picking a connection from the admit queue, a relayer thread opens a new connection with the least loaded server (chosen as described in Section 3) and handles the relaying of data between these connections. It returns to waiting for the arrival of new admitted connections into the admit queue once it is done servicing this connection. In our experiments, we had upto three classes, which meant upto three policer threads and three wait queues. The number of relayer threads could change dynamically depending on the request arrival rate. We

---

<sup>2</sup> In our current implementation, the class of a request is decided based on the URL requested. This mapping is read by the sentry from a configuration file at startup time. In general a request's class may be determined based on other criterion such as client IP address.



imposed lower and upper bounds of 80 and 2560 respectively on the number of relay threads in our experiments.

## 5.2 *Cataclysm Provisioning*

Cataclysm provisioning was implemented as a user-space daemon running on a dedicated machine. At startup, it reads information needed to communicate with the sentries and information about the servers in the cluster from a configuration file. It communicates with the sentries over TCP sockets. The sentries gather and report various statistics needed by the provisioning algorithm periodically over these sockets. The provisioning algorithm can be invoked in the reactive or predictive modes as discussed in Section 4. The default `ktcpvs` implementation provides a flexible user-space utility `tcpvsadm` using which the set of back end servers that it forwards the incoming requests to can be changed dynamically. We make use of this utility in our implementation. After determining a partitioning of the cluster’s servers among the hosted applications, the provisioning daemon uses scripts to remotely log on to the nodes running the sentries to enforce the partitioning. In our experiments, we chose the length of the default provisioning duration to be 30 minutes.

# 6 Experimental Evaluation

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

## 6.1 *Experimental Setup*

The cataclysm sentries were run on dual-processor 1 GHz machines with 1 GB RAM. The cataclysm control plane (responsible for provisioning) was run on a dual-processor 450 MHz machine with 1 GB RAM. The machines used as cataclysm servers had 2.8 GHz processors and 512 MB RAM. Finally, the workload generators were run on machines with processor speeds varying from 450MHz to 1 GHz and with RAM sizes in the range 128-512 MB. All machines ran Linux 2.4.20. In our experiments we constructed replicable applications using the Apache 1.3.28 Web server with PHP support enabled. The file set serviced by these Web servers comprised files of size varying from 1 kB to 256 kB to represent the range from small text files to large image files. In addition, the Web servers host PHP scripts with different computational overheads. The dynamic component of our workload consist of requests for these scripts. The MaxClient limit for Apache was increased from the default 150 to 500.

Recall from Section 3 that knowing a request’s CPU requirement was crucial for the determination of its inherent size. We made minor modifications to Apache and the Linux CPU scheduler to enable the measurement of per-request CPU usage. In all the experiments, the SLA presented in Figure 1 was used for the applications. Recall from Section 3 that knowing a request’s CPU

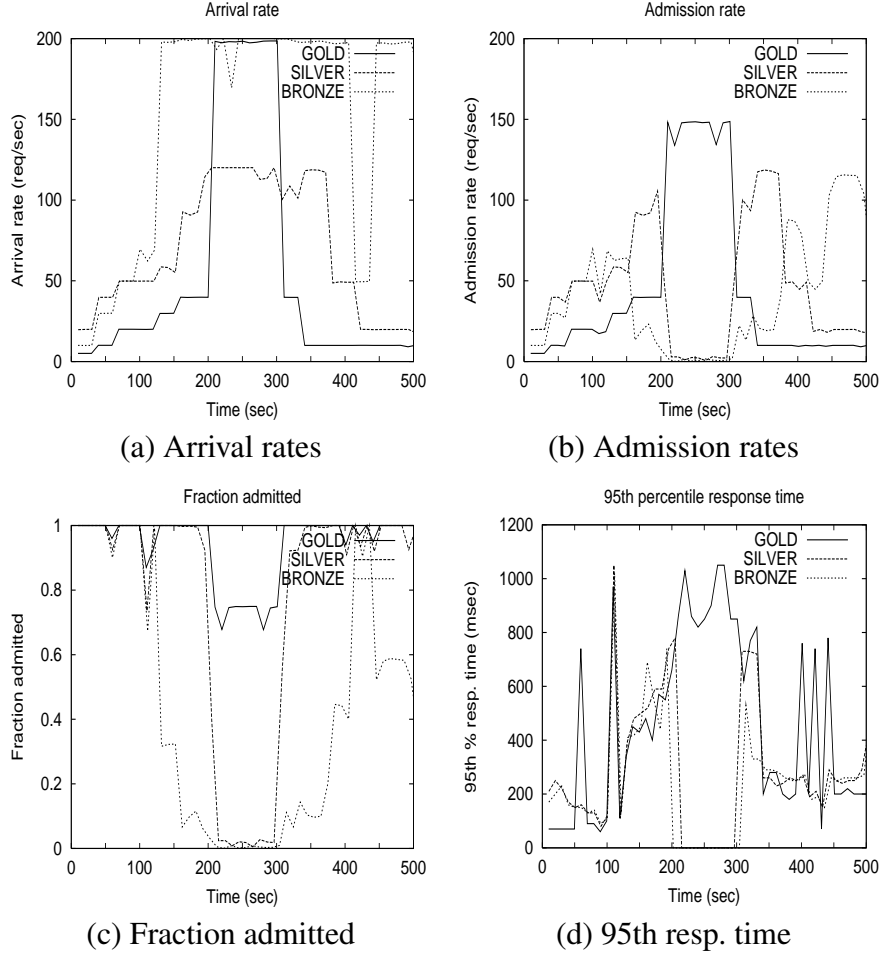


Fig. 3. Demonstration of the working of the admission control during an overload.

requirement was crucial for the determination of its inherent size. Since the notion of a request is specific to the service (the Apache web server in our case) and the operating system is oblivious of it, we made a minor modification to Apache to enable per-request CPU usage measurement. This consisted of inserting invocations of two new system calls `begin_record_cpu()` and `end_record_cpu()` in the Apache source code before request processing starts and after it ends respectively. The Linux CPU scheduler code was modified so that the CPU time used by a process (and all its descendants) is recorded between when it calls `begin_record_cpu()` till when it calls `end_record_cpu()`. Application requests are generated using `httperf` [25], an open-source Web workload generator. `httperf` offers a variety of workload generators. While running, it keeps track of a number of performance metrics that are summarized in the form of statistics that are reported at the end of the run. We enhanced `httperf` to record a specified percentile of the response time experienced by the requests during a run.

## 6.2 Revenue Maximization and Class-based Differentiation

Our first experiment investigates the efficacy of the mechanisms employed by the cataclysm sentry for revenue maximization and to provide class-based differentiation to requests during overloads. The cataclysm provisioning was kept turned off in this experiment. We constructed a replicated Web server consisting of three Apache servers. This application supported three classes of requests: Gold, Silver and Bronze in decreasing order of revenue. The class of a request could be uniquely determined from its URL. The delay values for the three classes were fixed at 0, 50, and 100 msec, respectively. The minimum sustainable requests rates desired by all three classes were chosen to be 0.

The workload consisted of requests for a set of PHP scripts. The capacity of each Apache server for this workload (i.e., the request arrival rate for which the 95<sup>th</sup> percentile response time of the requests was below the response time target) was determined offline and was found to be nearly 60 requests/sec. Figure 3(a) shows the workload used in this experiment. Nearly all the requests arriving till  $t = 130$  seconds were admitted by the sentry. Between  $t = 130$  seconds and  $t = 195$  seconds, the Bronze requests were dropped almost exclusively. At  $t = 195$  seconds the arrival rate of Silver requests shot up and reached nearly 120 requests/sec. The admission rate of Bronze requests dropped to almost zero to admit as many Silver requests as possible. At  $t = 210$  seconds, the arrival rate of Gold requests shot up to 200 requests/sec. The sentry totally suppressed all arriving Bronze and Silver requests now and let in only Gold requests as long as the increased arrival rate of Gold requests persisted. Figure 3(c) is an alternate representation of the system behavior in this experiment and depicts the variation of the fraction of requests of the three classes that were admitted. Figure 3(d) depicts the performance of admitted requests. We find that the sentry is successful in maintaining the response time below 1000 msec.

## 6.3 Scalable Admission Control

We measured the CPU utilization at the sentry server for different request arrival rates for both the batch-based and the threshold-based admission control. Figure 4 shows our observations of CPU utilization with 95% confidence intervals. Since we were interested only in the overheads of the admission control and not in the data copying overheads inherent in the design of the *ktcpvs* switch, we forced the sentry to drop all requests after conducting the admission control test. We increased the request arrival rates till the CPU at the sentry server became saturated (nearly 90% utilization). We observe more than a four-fold improvement in the sentry's scalability. Whereas the sentry CPU saturated at 4000 requests/sec with the batch-based admission control, it was able to handle almost 19000 requests/sec with the threshold-based admission control.

A second experiment was conducted to investigate the degradation in the decision making due to the threshold-based admission controller. We repeated the experiment reported in Section 6.2 (Figure 3) but forced the sentry to employ the threshold-based admission controller. The thresholds used by the admission control were computed once every 15 seconds. Figure 5(a) shows changes in the admission rates for requests of the three classes. The impact of the inaccuracies inherent in the

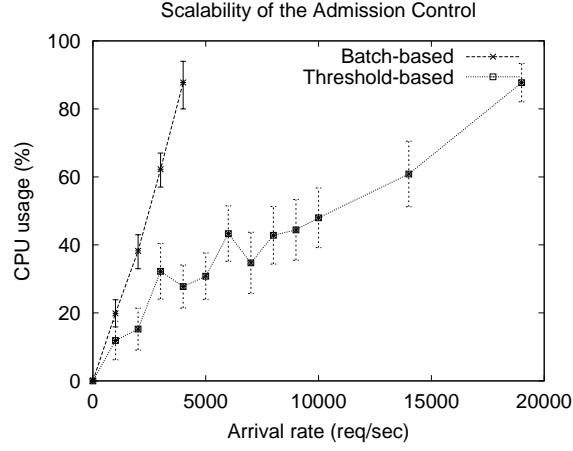


Fig. 4. Scalability of the admission control.

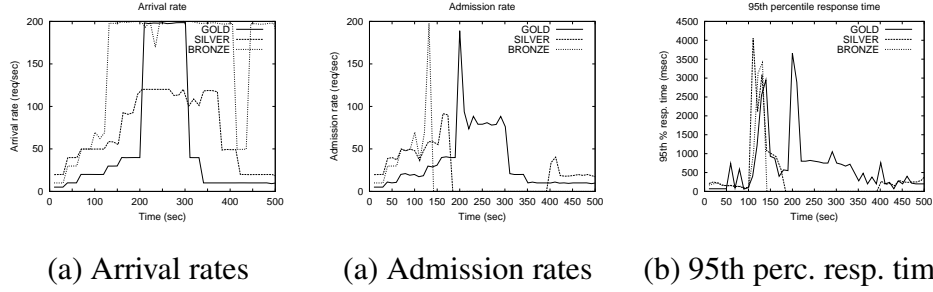


Fig. 5. Performance of the threshold-based admission control. At  $t = 135$  seconds, the threshold was set to reject all Bronze requests; at  $t = 180$  seconds, it was updated to reject all Bronze and Silver requests; at  $t = 210$  seconds it was updated to also reject Gold requests with a probability 0.5; finally, at  $t = 390$  seconds, it was again set to reject only Bronze requests.

threshold-based admission controller resulted in degraded performance during periods when the threshold chosen was incorrect. We observe two such periods (120-135 seconds during which all Bronze requests were dropped and 190-210 seconds during which all Bronze and Silver requests were dropped while Gold requests were admitted with probability of 0.5) during which the 95<sup>th</sup> percentile of the response time deteriorated compared to the target of 1000 msec. The response times during the rest of the experiment were kept under control due to the threshold getting updated to a strict enough value.

#### 6.4 Sentry Provisioning

We conducted an experiment to demonstrate the ability of the system to dynamically provision additional sentries to a heavily overloaded service. Figure 6 shows the outcome of our experiment. The workload consisted of requests for small static files sent to the sentry starting at 4000 requests/sec and increasing by 4000 requests/sec every minute and is shown in Figure 6(a). If the CPU utilization of the sentry server remained above 80% for more than 30 seconds, a request was issued to the control plane for an additional sentry. Figure 6(b) shows the variation of the CPU

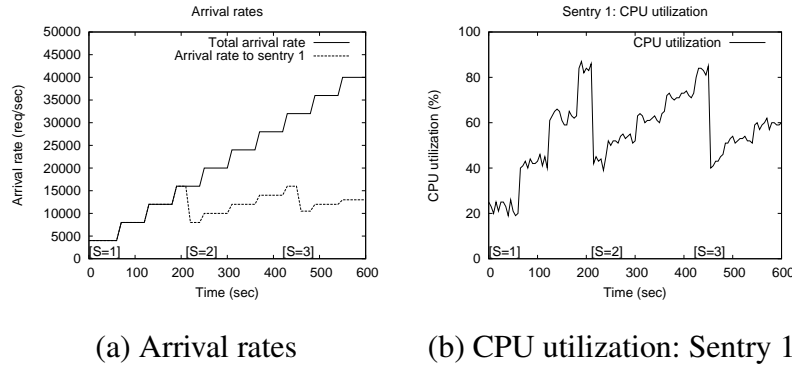


Fig. 6. Dynamic provisioning of sentries.  $[S=n]$  means the number of sentries is  $n$  now.

utilization at the first sentry. At  $t = 210$  seconds, a second sentry was added to the service. Subsequent requests were distributed equally between the two sentries causing the arrival rate and the CPU utilization at the first sentry to drop. A third sentry was added at  $t = 420$  seconds, when the total arrival rate to the service had reached 32000 requests/sec overwhelming both the existing sentries.

### 6.5 Revenue Maximization via Size-aware Admission Control

We conducted an experiment with an application constructed as in the first experiment to demonstrate the impact of the size-aware working of the admission control on system performance. The workload consisted of text files (1-8 kB), image files (64-128 kB) and dynamic requests for a collection of PHP scripts. Figures 7(a), (b) depict the workload used in the experiment and the rate at which requests of the three kinds were admitted by the sentry. There are three regions of overloads: first between  $t = 100$  seconds and  $t = 1250$  seconds due to the increased arrival rates of both static and dynamic, second between  $t = 1250$  seconds and  $t = 1900$  seconds due to high incidence of the dynamic requests and finally after  $t = 1900$  seconds due to arrival of excessive static requests. Because the admission control utilizes information about the resource requirements of the incoming requests gathered using the profiling described in Section 3.5, it is able to selectively admit almost all the requests for the lightweight text files. Requests for image files that are the most resource demanding in this workload are punished most severely. This demonstrates the ability of the admission control to maximize the number of admitted requests in the face of diverse workloads. A comparison of Figures 7(c) and (d) illustrates the efficacy of the admission control in maintaining response time targets. Without the admission control, the response times are observed to degrade uncontrollably under overload conditions as seen in Figure 7(d).

### 6.6 Handling Diverse Workloads

Our next experiment demonstrates that due accounting for different components of a request's execution time in its inherent service time, the admission control is able to operate effectively for workloads consisting of diverse and changing mix requests. We used a single web service identical

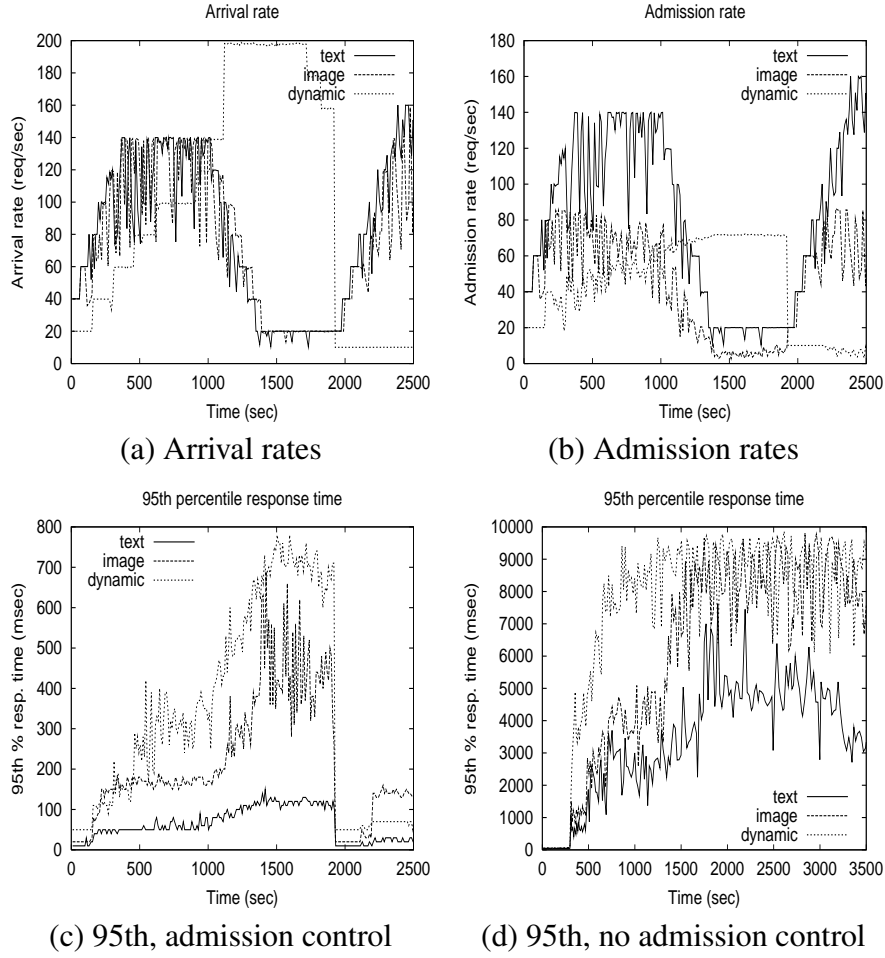


Fig. 7. Revenue maximization via size-aware admission control.

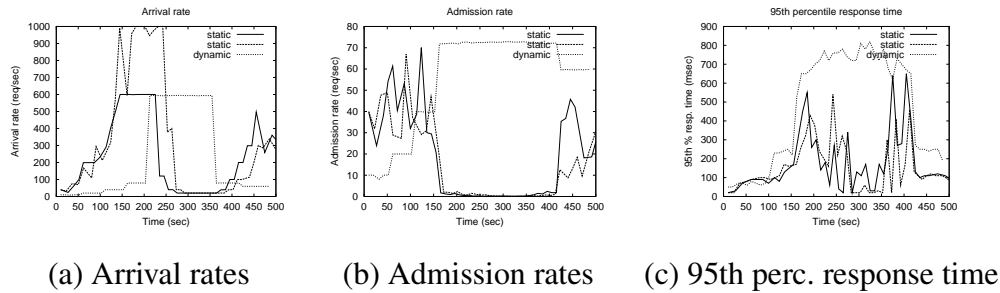


Fig. 8. Effective handling of diverse workloads.

to that described in Section 6.2. The workload consisted of three kinds of requests: static requests for members of a set of moderately large files (16-64 kB), static requests for members of a set of large files (64-128 kB) and dynamic requests for members of a set of PHP scripts. All the requests were made to belong to the same class. The workload the service was subjected to is presented in Figure 8(a). Between  $t = 0$  second and  $t = 160$  seconds, the arrival rates for both types of static requests rose fast and reached about 1000 requests/sec for large files and 600 requests/sec for moderately large files. These rates were an *order of magnitude higher* than the service's capacity for these requests, which was nearly 50 requests/sec for requests of either type. Meanwhile, the

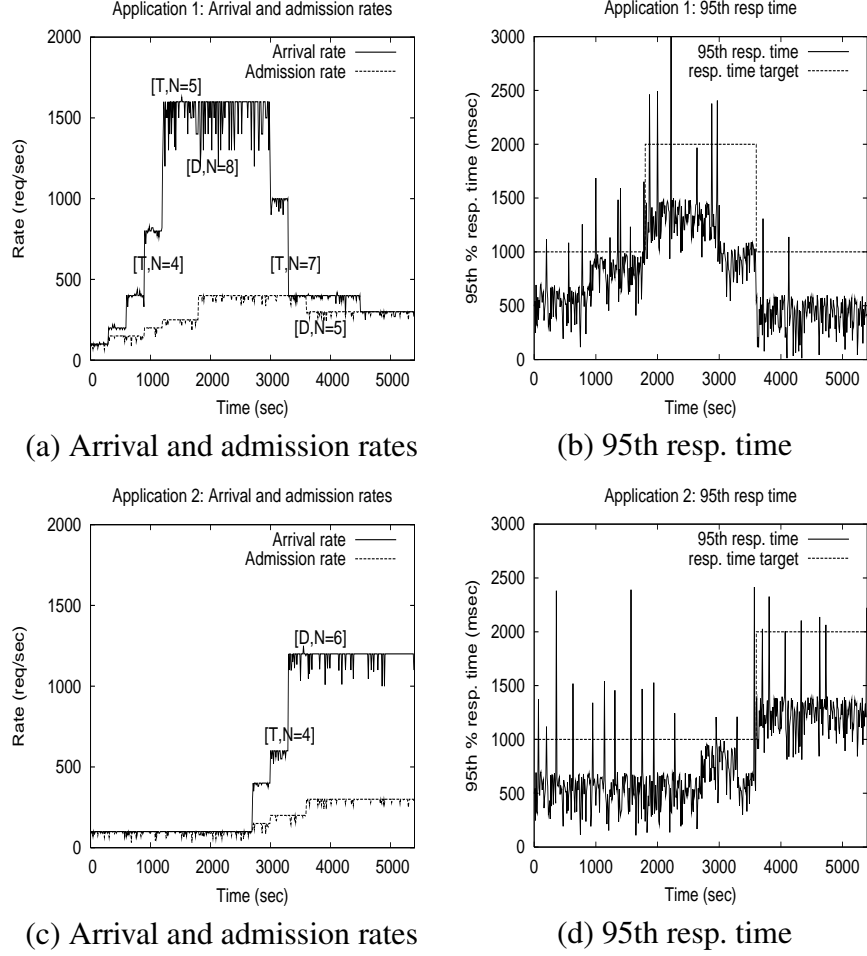


Fig. 9. Dynamic provisioning and admission control: Performance of Applications 1 and 2. D: Default invocation of provisioning, T: Provisioning triggered by excessive drops, [N=n]: size of the server set is n now. Only selected provisioning events are shown.

dynamic requests arrived at low rates and were all admitted while a limited number of the static requests were admitted. Between  $t = 160$  seconds and  $t = 410$  seconds, the workload mix changed and an overload was created by excessive dynamic requests. Due to their smaller inherent service times, the admission control exclusively admitted these requests and dropped all the static requests, thus maximizing the number of requests admitted. Beyond  $t = 410$  seconds, the dynamic requests subsided, allowing some static requests to be admitted. Figure 8 shows that response times were successfully kept under control during the experiment.

## 6.7 Cataclysm Provisioning

We conducted an experiment with two Web applications hosted on our Cataclysm platform. The total number of cataclysm servers available in this experiment was 11. The SLAs for both the applications were identical and are described in Figure 1. Further, the SLAs imposed a lower bound of 3 on the number of servers that each application could be assigned. The default provisioning

duration used by the control plane was 30 minutes.

The workloads for the two applications consisted of requests for an assortment of PHP scripts and files in the size range 1-128 kB. Requests were sent at a sustainable base rate to the two applications throughout the experiment. Overloads were created by sending increased number of requests for a small subset of the scripts and static files (to simulate a subset of the content becoming popular). The experiment began with the two applications running on 3 servers each. Sentries invoked the provisioning algorithm when more than 50% of the requests were dropped over an interval of length 5 minutes. Figures 9(a) and 9(c) depict the arrival rates to the two applications. The arrival rate for Application 1 was made to increase in a step-like fashion starting from 100 requests/sec, doubling roughly once every 5 minutes till it reached a peak value of 1600 requests/sec. At this point Application 1 was heavily overloaded with the arrival rate several times higher than system capacity (which was roughly 60 requests/sec per server assigned to the service as determined by offline measurements). At  $t = 910$  seconds the sentry, having observed more than 50% of the request being dropped, triggered the provisioning algorithm as described in Section 4. The provisioning algorithm responded by pulling one server from the free pool and adding it to Application 1. At  $t = 1210$  seconds, another server was added to Application 1 from the free pool. Observe in Figure 9(a) the increases in the admission rates corresponding to these additional servers being made available to Application 1. The next interesting event was the default invocation of provisioning at  $t = 1800$  seconds. The provisioning algorithm added all the 3 servers remaining in the free pool to the heavily overloaded Application 1. Also, based on recent observation of arrival rates, it predicted an arrival rate in the range 1000-10000 requests/sec and *degraded the response time target for Application 1* to 2000 msec based on its QoS table (see Figure 1). In the latter part of the experiment, the overload of Application 1 subsided and Application 2 got overloaded. The functioning of the provisioning was qualitatively similar to when Service 1 was overloaded. Figures 9(b) and 9(d) show the 95<sup>th</sup> percentile response times for the two services during the experiment. The control plane was able to predict changes to arrival rates and degrade the response time target according to the SLA resulting in an increased number of requests being admitted. Moreover, the sentries were able to keep the admission rates well below system capacity to achieve response times within the appropriate target with only sporadic violations (which were on fewer than 4% of the occasions).

## 7 Related Work

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

**Admission Control for Internet Services:** Many papers have developed overload management solutions based on doing admission control. Several admission controllers operate by controlling the rate of admission but without distinguishing requests based on their sizes. Voigt et al. [35] present kernel-based admission control mechanisms to protect web servers against overloads—*SYN policing* controls the rate and burst at which new connections are accepted, *prioritized listen queue* reorders the listen queue based on pre-defined connection priorities, *HTTP header-based*



*control* enables rate policing based on URL names. Welsh and Culler [36] propose an overload management solution for Internet services built using the SEDA architecture. A salient feature of their solution is feedback-based admission controllers embedded into individual *stages* of the service. The admission controllers work by gradually increasing admission rate when performance is satisfactory and decreasing it multiplicatively upon observing QoS violations. The QGuard system [18] proposes an adaptive mechanism that exploits rate controls for inbound to fend off overload and provide QoS differentiation between traffic classes. The determination of these rate limits, however, is not dynamic but is delegated to the administrator. Iyer et al. [17] propose a system based on two mechanisms—using thresholds on the connection queue length to decide when to start dropping new connection requests and sending feedback to the proxy during overloads which would cause it to restrict the traffic being forwarded to the server. However, they do not address how these thresholds may be determined online. Cherkasova and Phaal [10] propose an admission control scheme that works at the granularity of sessions rather than individual requests and evaluate it using a simple simulation study. This was based on a simple model to characterize sessions. The admission controller was based on rejecting *all* sessions for a small duration if the server utilization exceeded a pre-specified threshold and has some similarity to our approximate admission control, except we use information about the sizes of requests in various classes to determine the drop threshold.

Several efforts have proposed solutions based on analytical characterization of the workloads of Internet services and modeling of the servers. Kanodia and Knightly utilize a modeling technique called *service envelopes* to devise an admission control for web services that attempts to different response time targets for multiple classes of requests [20]. Li and Jamin [24] present a measurement-based admission control to distribute bandwidth across clients of unequal requirement. A key distinguishing feature of their algorithm is the introduction of controlled amounts of delay in the processing of certain requests during overloads to ensure different classes of requests are receiving the appropriate share of the bandwidth. Knightly and Shroff [22] describe and classify a broad class of admission control algorithms and evaluate the accuracy of these algorithms via experiments. They identify key aspects of admission control that enable it to achieve high statistical multiplexing gains.

Two admission control algorithms have been proposed recently that utilize measurements of request sizes to guide their decision making. Verma and Ghosal [33] propose a service time based admission control that uses predictions of arrivals and service times in the short-term future to admit a subset of requests that would maximize the profit of the service provider. Elnikety et al. [14] present an admission control for multi-tier e-commerce sites that externally observes execution costs of requests, distinguishing different requests types. Our measurement-based admission control is based on similar ideas, although the techniques differ in the details.

**Dynamic Provisioning and Managing Resources in Clusters:** The work on dynamic provisioning of a platform’s resources may be classified into two categories. Some papers have addressed the problem of provisioning resources at the granularity of individual servers as in our work. Ranjan et al. [27] consider the problem of dynamically varying the number of servers assigned to a single service hosted on a data center. Their objective is to minimize the number of servers needed to meet the service’s QoS targets. The algorithm is based on a simple scheme to extrapolate the

current size of the server set based on observations of utilization levels and workloads to determine the server set of the right size and is evaluated via simulations. The Oceano project at IBM [2] has developed a server farm in which servers can be moved dynamically across hosted applications depending on their changing needs. The main focus of this paper was on the implementation issues involved in building such a platform rather than the exact algorithms for provisioning.

Other papers have considered the provisioning of resources at finer granularity of resources. Muse [8] presents an architecture for resource management in a hosting center. Muse employs an economic model for dynamic provisioning of resources to multiple applications. In the model, each application has a utility function which is a function of its throughput and reflects the revenue generated by the application. There is also a penalty that the application charges the system when its goals are not met. The system computes resource allocations by attempting to maximize the overall profit. *Cluster Reserves* [3] has also investigated resource allocation in server clusters. The work assumes a large application running on a cluster, where the aim is to provide differentiated service to clients based on some notion of service *class*. This is achieved by making the OS schedulers provide fixed resource shares to applications spanning multiple nodes. The *Cluster-On Demand (COD)* [9] work presents an automated framework to manage resources in a shared hosting platform. COD introduces the notion of a *virtual cluster*, which is a functionally isolated group of hosts within a shared hardware base. A key element of COD is a protocol to resize virtual clusters dynamically in cooperation with pluggable middleware components. Chandra et al. [7] model a server resource that services multiple applications as a GPS system and presents online workload prediction and optimization-based techniques for dynamic resource allocation. Urgaonkar and Shenoy [30] address the problem of providing resource guarantees to distributed applications running on a shared hosting platform. Urgaonkar et al. [32] propose a resource overbooking based scheme for maximizing revenue in a shared platform.

An alternate approach for improving performance of overloaded web servers is based on re-designing the scheduling policy employed by the servers. Schroeder and Harchol-Balter [28] propose to employ the SRPT algorithm based on scheduling the connection with the shortest remaining time and demonstrate that it leads to improved average response time. While scheduling can improve response times, under extreme overloads admission control and the ability to add extra capacity are indispensable. Better scheduling algorithms are complementary to our solutions for handling overloads.

**Design of Efficient Load Balancers:** Our admission control scheme is necessarily based on the use of a Layer-7 switch and hence the scalable design of such switches is important to our implementation. Pai et al. [26] design locality-aware request distribution (LARD), a strategy for content-based request distribution that can be employed by front servers in network servers to achieve high locality in the back end servers and good load balancing. They introduce a *TCP handoff protocol* that can hand off an established TCP connection in a client-transparent manner. A load balancer based on TCP handoff has been shown to be more scalable than the `ktcpvs` load balancer we have used. Aron et al. [4] present a highly scalable architecture for content-aware request distribution in Web server clusters. The front switch is a Layer-4 switch that distributed requests to a number of back-end nodes. Content-based distribution is performed by these back-end servers. Cardellini et al. [6] provide a comprehensive survey of the main mechanisms to split traffic among the servers

in a cluster, discussing both the various architectures and the load sharing policies. Our admission control and load balancing schemes are independent of the actual switch implementation so long as it is Layer-7 and hence may be implemented in any of the aforementioned scalable switches.

**Modeling of Internet Services:** Accurate modeling of applications is crucial for translating QoS needs to resource requirements. Doyle et al. [13] present a *model-based* utility resource management focusing on coordinated management of memory and storage. They develop an analytical model for a web service with static content. Kamra et al. [19] use an M/GI/1 processor sharing queue as an abstraction for a 3-tier e-commerce application. Urgaonkar et al. present an analytical model that captures many idiosyncrasies of modern Internet applications such as the presence of multiple tiers, concurrency limits, caching, replication etc. [29]. Bennani and Menasce utilize a queueing model for multi-tier applications for capacity provisioning [5]. A machine-learning based approach for modeling multi-tier applications has been proposed by Cohen et al. [11]. The dynamic provisioning scheme presented in this paper can be improved by replacing the G/G/1-model with these improved models.

**SLAs and Adaptive QoS Degradation:** The WSLA project at IBM [37] addresses service level management issues and challenges in designing an unambiguous and clear specification of SLAs that can be monitored by the service provider, customer and even by a third-party. Abdelzaher and Bhatti [1] propose to deal with server overloads by adapting delivered content to load conditions. This is a different kind of QoS degradation than what we have proposed in our work, but it can be integrated into a Cataclysm platform by defining appropriate SLAs based on it.

## 8 Conclusions

In this paper we presented *Cataclysm*, a comprehensive approach for handling extreme overloads in a hosting platform running multiple Internet services. The primary contribution of our work was to develop a low overhead, highly scalable admission control technique for Internet applications. Cataclysm provides several desirable features, such as guarantees on response time by conducting accurate size-based admission control, revenue maximization at multiple time-scales via preferential admission of important requests and dynamic capacity provisioning, and the ability to be operational even under extreme overloads. The cataclysm sentry can transparently trade-off the accuracy of its decision making with the intensity of the workload allowing it to handle incoming rates of up to 19000 requests/second. We implemented a prototype Cataclysm hosting platform on a Linux cluster and demonstrated its benefits using a variety of workloads.

## References

- [1] T. Abdelzaher and N. Bhatti. Web Content Adaptation to Improve Server Overload Behavior. In *Proceedings of the World Wide Web Conference (WWW8)*, Toronto, 1999.

- [2] K. Appleby, S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based Management of a Computing Utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May 2001.
- [3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA*, June 2000.
- [4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA*, June 2000.
- [5] M. Benani and D. Menasce. Resource Allocation for Autonomic Data Centers Using Analytic Performance Models. In *Proceedings of IEEE International Conference on Autonomic Computing, Seattle (ICAC-05), WA*, June 2005.
- [6] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The State of the Art in Locally Distributed Web-server Systems. In *ACM Computing Surveys (CSUR) archive*, 34:2, pages 263–311, June 2002.
- [7] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of the Eleventh International Workshop on Quality of Service (IWQoS 2003), Monterey, CA*, June 2003.
- [8] J. Chase and R. Doyle. Balance of Power: Energy Management for Server Clusters. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany*, May 2001.
- [9] J. Chase, L. Grit, D. Irwin, J. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.
- [10] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving performance of commercial web sites. In *Proceedings of Seventh International Workshop on Quality of Service, IEEE/IFIP event, London*, June 1999.
- [11] I. Cohen, J. Chase, M. Goldszmidt, T. Kelly, and Julie Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the Sixth USENIX Symposium in Operating Systems Design and Implementation (OSDI 2004), San Francisco, CA*, December 2004.
- [12] Computer Science and Telecommunications Board Division on Engineering and Physical Sciences, National Research Council. The Internet Under Crisis Conditions: Learning from September 11. In *Report on the Internet Under Crisis Conditions*, 2002.
- [13] R. Doyle, J. Chase, O. Asad, W. Jin, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the Fourth USITS*, March 2003.
- [14] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the Thirteenth international conference on World Wide Web, New York, NY, USA*, pages 276–286, 2004.
- [15] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, January 1979.

- [16] J. Hellerstein, F. Zhang, and P. Shahabuddin. A Statistical Approach to Predictive Detection. *Computer Networks*, January 2000.
- [17] R. Iyer, V. Tewari, and K. Kant. Overload Control Mechanisms for Web Servers. In *Workshop on Performance and QoS of Next Generation Networks*, November 2000.
- [18] H. Jamjoom, J. Reumann, and K. Shin. QGuard: Protecting Internet Servers from Overload. Technical Report CSE-TR-427-00, Department of Computer Science, University of Michigan, 2000.
- [19] A. Kamra, V. Misra, and E. Nahum. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. In *Proceedings of the Twelfth IWQoS*, 2004.
- [20] V. Kanodia and E. Knightly. Multi-Class Latency-Bounded Web Servers. In *Proceedings of International Workshop on Quality of Service (IWQoS'00)*, June 2000.
- [21] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc., 1976.
- [22] E. Knightly and N. Shroff. Admission Control for Statistical QoS: Theory and Practice. In *IEEE Network* 13:2, pages 20–29, March/April 1999.
- [23] Kernel TCP Virtual Server. <http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html>.
- [24] S. Li and S. Jamin. A Measurement-based Admission-controlled Web Server. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, Tel Aviv, Israel, March 2000.
- [25] D. Mosberger and T. Jin. httpperf—a tool for measuring web server performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [26] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwanepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.
- [27] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-Driven Server Migration for Internet Data Centers. In *Proceedings of the Tenth International Workshop on Quality of Service*, Miami, FL, 2002.
- [28] B. Schroeder and M. Harchol-Balter. Web Servers Under Overload: How Scheduling Can Help. In *Proceedings of the Eighteenth International Teletraffic Congress*, 2003.
- [29] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and its Applications. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2005)*, Banff, Canada, June 2005.
- [30] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. In *IEEE Transactions on Parallel and Distributed Systems*, 15:1, pages 2–17, January 2004.
- [31] B. Urgaonkar and P. Shenoy. Cataclysm: Policing Extreme Overloads in Internet Applications. In *Proceedings of the Fourteenth International World Wide Web Conference (WWW 2005)*, Chiba, Japan, May 2005., May 2005.
- [32] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

- [33] A. Verma and S. Ghosal. On Admission Control for Profit Maximization of Networked Service Providers. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 2003.
- [34] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning Servers in the Application Tier for E-commerce Systems. In *Proceedings of the Twelfth IWQoS*, June 2004.
- [35] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of USENIX Annual Technical Conference*, June 2001.
- [36] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the Fourth USENIX Conference on Internet Technologies and Systems (USITS'03)*, March 2003.
- [37] Web service level agreements (wsa) project. <http://www.research.ibm.com/wsla>.