CHAPTER 7
OVERLOAD MANAGEMENT

7.1 Introduction

7.1.1 Motivation

We have already discussed that the workload seen by Internet applications varies over multiple time-scales and often in an unpredictable fashion [1]. As argued in Chapter 4, certain workload variations such as time-of-day effects are easy to predict and handle by appropriate capacity provisioning. 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 [1]. The load on e-commerce retail Web sites can increase dramatically during the final days of the popular holiday season. Similarly, the load on online brokerage Web sites can be several times greater than the average load during an unexpected market crash.

In this chapter, 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. 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.

The first two approaches have been studied in the literature. The first approach involves dynamic provisioning to match application capacity to the workload demand and has been addressed by us (Chapter 4) and others [26, 76, 91]. The second approach involves policing in the form of admission control, which limits the number of admitted requests so that the contracted performance guarantees are met [29, 42, 117, 123]. The notion of providing preferential treatment to “important” requests has also been studied (e.g., by giving higher priority to certain requests, such as those involving financial transactions [18]). Last, the notion of gracefully degrading application performance with increasing loads, while intuitively appealing, has not been studied from the perspective of extreme overloads.

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.

It is important to note that such a comprehensive approach to handling severe overloads involves more than the implementation of separate mechanisms to achieve each of the above goals. Mechanisms such as dynamic provisioning and admission control can be coupled in useful and non-trivial ways to further improve the handling of extreme overloads. For instance, the admission controller can pro-actively invoke the dynamic provisioning mechanism when the request drop rate exceeds a certain threshold. The dynamic
provisioning mechanism in turn can provide useful information to the admission controller regarding the provisioned capacity so that the latter can set appropriate performance thresholds for admitted requests. Such an integration of mechanisms can enhance the ability of the platform to handle overloads.

An orthogonal goal for the hosting platform is robustness under severe overloads. Robustness—the ability to remain operational under overloads—requires the hosting platform to be both extremely agile and efficient. Agility requires a quick response in the face of a sudden workload spike. Efficiency requires the above-mentioned mechanisms, and in particular the admission controller, to have very low overheads. Since an extreme overload may involve request rates that are up to 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.

Whereas prior approaches for handling overloads have considered individual mechanisms such as provisioning and admission control, in this thesis, we focus on an integrated approach, with a particular emphasis on handling extreme overloads.

### 7.1.2 Research Contributions of this Chapter

We describe the aspects of our hosting platform concerned with handling extreme overloads in Internet applications. Our approach differs from past work in three significant respects.

First, since an extreme overload may involve request rates that are \textit{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 [29, 42, 117, 123] 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. (not all of these requests are necessarily admitted and serviced; the admitted fraction depends on the available capacity).

Second, our platform has the 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.

Last, our work demonstrates that dynamic provisioning and admission control can be coupled in useful ways to enhance the ability of the platform in handling extreme overloads. For instance, the admission controller can pro-actively invoke dynamic provisioning when the request drop rate exceeds a certain threshold, and the provisioning mechanisms can provide useful information to the admission controller for policing requests. Past work on admission control [29, 42, 117, 123] and dynamic provisioning [26, 91] considered each technique in isolation and did not study the impact of such couplings.

We have implemented our overload control mechanisms in our prototype Linux hosting. 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.
7.1.3 Organization

The rest of this chapter is organized as follows. Section 7.2 provides an overview of the proposed system. Sections 7.3 and 7.4 describe the mechanisms that constitute our overload management solution. Section 7.5 describes the implementation of our prototype. In Section 7.6 we present the results of our experimental evaluation. Section 7.7 presents related work and Section 7.8 concludes this chapter.

7.2 System Overview

In this section, we present the system model for our hosting platform and the service-level agreement assumed in our work.

7.2.1 Hosting Platform Architecture

We show the hosting platform architecture in Figure 7.1. We assume a dedicated hosting model in this chapter.

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 [83], we do not consider load balancing techniques in this work.

Whereas a single sentry suffices for small applications, large applications require multiple sentries, since a single sentry server will become a bottleneck when guarding a large number of servers. Just as the number of servers allocated to an application vary with the load, our hosting platform can dynamically vary the number of sentries depending on the incoming request rate (and the corresponding load on the sentries). When a sentry is assigned or deallocated, the application’s server pool is repartitioned and each remaining sentry is assigned responsibility for a mutually exclusive subset of nodes. Each sentry then independently performs admission control and load balancing on arriving requests, thereby collectively maintaining the SLA for the application as a whole. A round-robin DNS scheme is used to partition (and loosely balance) the incoming requests across multiple sentries.

As before, the control plane is responsible for dynamic provisioning of servers and sentries in individual applications. It tracks the resource usages on servers, as reported by the 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.

7.2.2 Service-level Agreement

Given an 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:
Arrival rate | Avg. resp. time for admitted requests
---|---
< 1000 | 1 sec
1000-10000 | 2 sec
> 10000 | 3 sec

Table 7.1. A sample service-level agreement.

\[
\text{Avg resp time } R \text{ of adm req} = \begin{cases} 
R_1 & \text{if arrival rate } \in [0, \lambda_1) \\
R_2 & \text{if arrival rate } \in [\lambda_1, \lambda_2) \\
\vdots & \\
R_k & \text{if arrival rate } \in [\lambda_{k-1}, \infty) 
\end{cases} 
\quad (7.1)
\]

The SLA specifies the revenue that is generated by each request that meets its response time target. Table 7.1 illustrates an example SLA.

Each Internet application consists of \( L (L \geq 1) \) request classes: \( C_1, \ldots, C_L \). 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, a vanilla Web server may define two classes and may map all requests smaller than a certain size \( s \) to class \( C_1 \) and larger requests to \( C_2 \). In contrast, 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.

### 7.3 Sentry Design

In this section, we describe the design of a sentry. The sentry is responsible for two tasks—request policing and load balancing. As indicated earlier, the load balancing tech-
nique 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 the one proposed by Pai et al. [83]. 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.

7.3.1 Request Policing Basics

The sentry maps each incoming request to one of the classes $C_1, \ldots, C_L$. 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 7.2). It does so by implementing leaky buckets, one for each class, that admit requests confirming to these rates. Requests confirming 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 7.2); 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 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 ($d_1 = 0 \leq d_2 \leq \ldots \leq d_L$); 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 Appendix B 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 7.4.2. In this section we present techniques to deal with the scalability of the admission control test.

### 7.3.2 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 hashes 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 hashed to bucket 1, that with service time in the range $(s_1, s_2]$ to bucket 2, and so on. By defining an appropriate hashing function, hashing a request to a bucket can be implemented efficiently as a constant time operation.

Bucket-based hashing 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 assum-
ing 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 hashing implicitly “sorts” requests when mapping them into buckets.

![Diagram](image)

**Figure 7.2.** Working of the 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 hashing process and the admission control test in detail. In order to hash 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 $R$ is defined to be

$$S_{inherent} = R_{cpu} + \alpha \cdot R_{data} \quad (7.2)$$

where $R_{cpu}$ is the total CPU time needed to service $R$, $R_{data}$ 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 hash 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 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 7.3.5. 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,

$$Q + \left( \eta + \left\lceil \frac{\beta}{n} \right\rceil \right) \cdot S \leq R_{sla} \quad (7.3)$$

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., \( \lceil \frac{2}{n} \rceil \).\(^1\) 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, \( d_1 = 0 \leq d_2 \leq \ldots \leq d_L \). 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

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

(7.4)

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 (7.4). 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 application has sufficient capacity to handle all the incoming traffic. We discuss in Appendix B 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 7.3.5.

---

\(^1\)Note that we have made the assumption of perfect load balancing in the admission control test (7.3). 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.
7.3.3 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 \( p_{\text{admit}} \)). The threshold indicates that all requests in classes more important than \( i \) should be admitted (\( p_{\text{admit}} = 1 \)), requests in class \( i \) should be admitted with probability \( p_{\text{admit}} \), and all requests in classes less important than \( i \) should be dropped (\( p_{\text{admit}} = 0 \)). 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 \( 1, \ldots, L \) by \( \lambda_1, \ldots, \lambda_L \) and the observed average service times by \( s_1, \ldots, s_L \), the threshold \((i, p_{\text{admit}})\) is computed such that

\[
\sum_{j=i}^{j=L} \lambda_j s_j \geq 1 - \sum_{j=1}^{j=L} \lambda_j^{\text{min}} s_j \tag{7.5}
\]

and

\[
p_{\text{admit}} \cdot \lambda_i s_i + \sum_{j=1}^{j=i-1} \lambda_j s_j < 1 - \sum_{j=1}^{j=L} \lambda_j^{\text{min}} s_j \tag{7.6}
\]

where \( \lambda_j^{\text{min}} \) 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 7.6.3.

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 [42, 61, 123] (discussed in Section 7.7) 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.
7.3.4 Analysis of the Policer

In Appendix B we show how the sentry can, under certain assumptions, compute the delay values for various classes based on online observations.

7.3.5 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 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 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 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 (7.2)).

- **Constant \( \alpha \):** The constant \( \alpha \) in Equation (7.2) is measured using offline measurements on the 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. (7.2).

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.

### 7.4 Capacity Provisioning

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 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 servers and sentries.

7.4.1 Model-based Provisioning for Applications

We employ the provisioning technique described in Chapter 4 in our hosting platform. Recall that this technique is based on a combination of a predictive provisioning technique based on the queuing-theoretic model presented in Chapter 2 and a reactive provisioning technique to handle errors in prediction and flash crowds.

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.

7.4.2 Sentry Provisioning

In general, allocation and deallocation of sentries is significantly less frequent than that of 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 \( \text{low}_{\text{cpu}} \), 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.

### 7.5 Implementation Considerations

We implemented a prototype hosting platform on a cluster of 40 Pentium machines connected via a 1Gbps 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 sentry, (3) the control plane, (4) a workload generator for an application.

**Sentry:** We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [66] to implement the policing mechanisms described in Section 7.3. ktcpvs is an open-source, Layer-7 load balancer implemented as a Linux module. It accepts TCP connections from clients, opens separate connections with servers (one for each client), and transparently relays data between these. We modified ktcpvs to implement all the sentry mechanisms described in Sections 7.3 and 7.4.

### 7.6 Experimental Evaluation

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

The sentries were run on dual-processor 1GHz machines with 1GB RAM. The control plane (responsible for provisioning) was run on a dual-processor 450MHz machine with 1GB RAM. The machines used as servers had 2.8GHz processors and 512MB RAM. Finally, the workload generators were run on machines with processor speeds varying from 450MHz to 1GHz and with RAM sizes in the range 128MB-512MB. 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 1kB to 256kB to represent the range from small text files to large image files. In addition, the Web servers hosted PHP scripts with different computational overheads. The dynamic component of our workload consisted of requests for these scripts. In all the experiments, the SLA presented in Figure 7.1 was used for the applications. Application requests were generated using httpperf [77], an open-source Web workload generator.

7.6.2 Revenue Maximization and Class-based Differentiation

Our first experiment investigates the efficacy of the mechanisms employed by the sentry for revenue maximization and to provide class-based differentiation to requests during overloads. The 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 95th 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 7.3(a) shows the workload used in this experiment. Nearly all the requests arriving till t=130 sec were admitted by the sentry. Between t=130 sec and t=195 sec, the Bronze requests were dropped almost exclusively. At t=195 sec 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 sec, 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 7.3(c) is an alternate representation of the system behavior in this experiment and

Figure 7.3. Demonstration of the working of the admission control during an overload.
depicts the variation of the fraction of requests of the three classes that were admitted. Figure 7.3(d) depicts the performance of admitted requests. We find that the sentry is successful in maintaining the response time below 1000 ms.

### 7.6.3 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 7.4 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 7.6.2 (Figure 7.3) but forced the sentry to employ the threshold-based admission controller. The thresholds used by the admission control were computed once every 15 sec. Figure 7.5(a) shows changes in the admission rates for requests of the three classes. The impact of the inaccuracies inherent in the threshold-based admission controller resulted in degraded performance during periods when the threshold chosen was incorrect. We observe two such periods (120-135 sec during which all Bronze requests were dropped and 190-210 sec during which all Bronze and Silver requests were dropped while Gold requests were admitted with probability of 0.5) during which the $95^{th}$ 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.

### 7.6.4 Sentry Provisioning

We conducted an experiment to demonstrate the ability of the system to dynamically provision additional sentries to a heavily overloaded service. Figure 7.6 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 7.6(a). If the CPU utilization of the sentry server remained above 80% for
more than 30 sec, a request was issued to the control plane for an additional sentry.
Figure 7.6(b) shows the variation of the CPU utilization at the first sentry. At t=210 sec,
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 sec, when the total arrival rate to the service had
reached 32000 requests/sec overwhelming both the existing sentries.

7.6.5 Provisioning

We conducted an experiment with two Web applications hosted on our platform. The
total number of servers available in this experiment was 11. The SLAs for both the appli-
cations were identical and are described in Figure 7.1. 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 min.
The workloads for the two applications consisted of requests for an assortment of PHP scripts and files in the size range 1kB-128kB. 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 a 5 min interval. Figures 7.7(a) and 7.7(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 min 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 request/sec per server assigned to the service as determined by offline measurements). At t=910 sec the sentry, having observed more than 50% of the request being dropped, triggered the provisioning algorithm as described in Section 7.4. The provisioning algorithm responded by pulling one server from the free pool and adding it to Application 1. At t=1210 sec, another server was added to Application 1 from the free pool. Observe in Figure 7.7(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 sec. 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 7.1). 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 7.7(b) and 7.7(d) show the 95th 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).
7.7 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 imposing fixed, statically-determined limits on one or more service parameters. The simplest example of such admission control is the upper limit on the number of simultaneous processes or threads in commonly used servers such as Apache [5]. Voigt et al. 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 [120]. Welsh and Culler propose an overload management solution for Internet services built using the SEDA architecture [123]. 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 proposes an adaptive mechanism that exploits rate controls for inbound to fend off overload and provide QoS differentiation between traffic classes [56]. The determination of these rate limits, however, is not dynamic but is delegated to the administrator. Iyer et al. 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 [54]. However, they do not address how these thresholds may be determined online. Cherkasova and Phaal propose an admission control scheme that works at the granularity of sessions rather than
individual requests and evaluate it using a simple simulation study [29]. 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 envelops to devise an admission control for web services that attempts to different response time targets for multiple classes of requests [62]. Li and Jamin present a measurement-based admission control to distribute bandwidth across clients of unequal requirement [70]. 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 describe and classify a broad class of admission control algorithms and evaluate the accuracy of these algorithms via experiments [65]. 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 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 [117]. Elnikety et al. [42] present an admission control for multi-tier e-commerce sites that externally observes execution costs of requests, distinguishing different requests types [42]. Our measurement-based admission control is based on similar ideas, although the techniques differ in the details.

**Dynamic Provisioning and Managing Resources in Clusters:**
We refer the reader to Chapters 4 and 6 for discussions on state-of-the-art dynamic capacity provisioning for clusters of servers.

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 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 [97]. 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. 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 [83]. They introduce a TCP hand-off protocol that can hand off an established TCP connection in a client-transparent manner. A load balancer based on TCP hand-off has been shown to be more scalable than the ktcpvs load balancer we have used. Aron et al. present a highly scalable architecture for content-aware request distribution in Web server clusters [11]. 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. 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 [21]. 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.

**SLAs and Adaptive QoS Degradation:** The WSLA project at IBM addresses service level management issues and challenges in designing an unambiguous and clear specifi-
cation of SLAs that can be monitored by the service provider, customer and even by a third-party [125]. Abdelzaher and Bhatti propose to deal with server overloads by adapting delivered content to load conditions [2]. This is a different kind of QoS degradation than what we have proposed in our work, but it can be integrated into a platform by defining appropriate SLAs based on it.

In this chapter we showed the utility of coupling policing and provisioning, in contrast to prior approaches that considered these techniques in isolation.

7.8 Conclusions

In this chapter we presented, 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. It 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 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 hosting platform on a Linux cluster and demonstrated its benefits using a variety of workloads.