Cataclysm: Handling Extreme Overloads in Internet Services

Bhuvan Urgaonkar and **Prashant Shenoy**
Department of Computer Science,
University of Massachusetts,
Amherst MA 01003
{*bhuvan, shenoy*}*@cs.umass.edu*

Abstract

In this paper we present *Cataclysm*, a comprehensive approach for handling extreme overloads in hosted Internet applications. The primary contribution of our work is to develop an overload control approach that brings together admission control, dynamic provisioning of platform resources, and adaptive degradation of QoS into one integrated system. Cataclysm provides several desirable features under overloads, such as preferential admission of important requests, the ability to handle diverse workloads, and revenue maximization at multiple time-scales via dynamic provisioning and size-based admission control. Cataclysm can transparently tradeoff 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.

1 Introduction

1.1 Motivation

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, 29]. Certain workload variations such as time-of-day effects are easy to predict and handle by appropriate capacity provisioning [15]. 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 [29]. 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 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 number of 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.

The first two approaches have been studied in the literature. The first approach involves dynamic provisioning to match application capacity to the workload demand [9, 24, 27]. 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 [11, 14, 32, 35]. The notion of providing preferential treatment to "important" requests has also been studied, although from the perspective of maximizing revenue (e.g., by giving higher priority to certain requests, such as those involving financial transactions [6]).

In this paper, we argue that a comprehensive approach to handling extreme overloads should involve a synergistic 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 utility. 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 implemen-

tation 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 paper, we focus on an integrated approach, with a particular emphasis on handling extreme overloads.

1.2 Research Contributions of this Paper

In this paper, we present Cataclysm, a hosting platform that is designed to handle extreme overloads in Internet applications. The key contribution of our work is to integrate techniques such as admission control, dynamic resource provisioning, and adaptive degradation of QoS into one integrated system for handling overloads. Unlike recent efforts on admission control [19, 32], our techniques are specifically focused on handling extreme overloads. We propose very low overhead admission control mechanisms that can scale to very high request rates under overloads. Our mechanisms can preferentially admit important requests during overload and transparently tradeoff accuracy of their decision making with the intensity of the workload, enabling them 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). Multiple admission controllers can be employed in larger Internet applications for greater scalability.

Our dynamic provisioning mechanism employs a G/G/1-based queuing theoretic model of a replicable application in conjunction with online measurements to dynamically vary the number of servers allocated to each application. The provisioning technique can be reactively invoked by the admission controller when the request drop rates exceed certain values. 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. Further, the admission controller and provisioning mechanism cooperate with one another, and thereby enhance the ability of the platform to counter overloads.

We have implemented a prototype Cataclysm hosting platform on a cluster of 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) reactive provisioning is both agile and effective at diverting platform resources to where they are needed most, while maximizing platform revenue.

The rest of this paper is organized as follows. Section 2 provides an overview of the proposed system. Sections 3, 4 and 5 describe the mechanisms that constitute our overload management solution. Section 6 describes the implementation of our prototype. In Section 7 we present the results of our experimental evaluation. Section 8 presents related work and Section 9 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 of the nodes, and a node is assumed to run no more than one application at any given time (this is referred to as *dedicated hosting* in the literature). Not all cataclysm servers need to be assigned to applications—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. An application's server pool can be increased either by allocating servers from the free pool or by deallocating under-utilized servers from another application and reassigning them to the overloaded application.

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

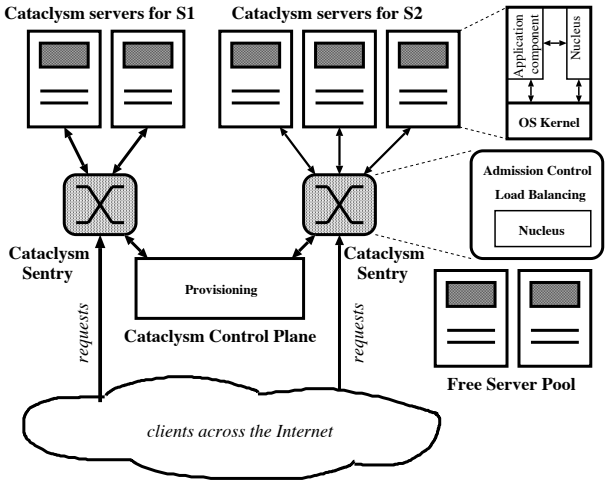


Figure 1: The Cataclysm Hosting Platform Architecture.

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. There has been substantial research on load balancing techniques for clustered Internet applications [7]; any such layer-7 load balancing technique may be used in the sentry.

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.

Cataclysm Control Plane: The control plane is responsible for provisioning servers and sentries for 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 and the sentries cooperate with one another during an overload. A sentry can invoke the provisioning mechanisms in the control plane when an overload is detected, and the control plane can provide provisioning-related information to sentries for admission control.

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 application 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*.[1] 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. An SLA has two components: (1) a description of the QoS guarantees that the platform will provide to the application, and (2) the revenue scheme that will be used by the platform to bill the application (which may include penalties for violations of the contracted performance guarantees). The SLA is defined as follows:

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

$$\text{Revenue} = \begin{cases} \gamma_1 & if RT \leq \mathcal{R}_1 \text{and arrivals} \leq \lambda_1 \\ \gamma_2 & if RT \leq \mathcal{R}_2 \text{and arrivals} \leq \lambda_2 \\ \cdots & \\ \gamma_k & if RT \leq \mathcal{R}_k \text{and arrivals} > \lambda_{k-1} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Essentially, the SLA specifies the desired response times for different arrival rates and also the revenue earned by the platform for meeting the desired response time at the corresponding arrival rate. Figure 2 illustrates one such SLA. Additionally, the SLA may also specify a lower bound on the number of servers that an application should be assigned.

[1] A common technique for implementing a multi-tiered application is to use the so-called "LAMP" servers—Linux, Apache, mysql and php.

QoS Table

arrival rate (requests/sec)	avg. resp. time for admitted req
< 1000	1 sec
1000 - 10000	2 sec
> 10000	3 sec

Revenue Table

arrival rate (requests/sec)	revenue per admitted request
< 1000	0.05 if avg. RT $\leq$ 1 sec 0 otherwise
1000 - 10000	0.05 if avg. RT $\leq$ 2 sec 0 otherwise
> 10000	0.05 if avg. RT $\leq$ 3 sec 0 otherwise

Figure 2: A sample service level agreement.

3 Cataclysm Sentry Design

In this section, we describe the design of a Cataclysm sentry which is responsible for two tasks—admission control 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]. Given our focus on extreme overloads, our design of the admission controller focuses on two key issues: (i) to ensure very low overhead admission control tests in order to scale to very high request rates seen during overloads, and (ii) to maximize the utility and the number of requests serviced during overloads by preferentially admitting more important requests. In the rest of this section, we elaborate on these two issues in detail.

3.1 Admission Control Basics

Each Internet application is assumed to consist of L request classes: $C_1, C_2, \ldots, C_L$. The sentry maps each incoming request to a class. The class of a request determines its importance—requests mapped to class C_1 are treated as most important and those in C_L as the least important. 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 customer requests such as balance inquiries to C_2, and casual browsing requests from non-customers to C_3. Each class has a queue associated with it; incoming requests are appended to the corresponding class-specific queue (see Figure 3).

The admission controller 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.

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.

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 $O(r \cdot logr)$ due to the need to sort requests. Since the incoming request rate can be several times 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.

3.2 Efficient Batch Processing

One possible approach for reducing the admission control overhead is to process requests in *batches*. Observe that 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 $(0, s1]$ is hashed to bucket 1, that with service time in the range $(s1, s2]$ 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.

The admission control is invoked periodically on each request class. The admission control then considers each non-empty bucket in that class and invokes a single admission control test on all requests in that bucket (i.e., all requests within 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. Since successive buckets contain requests with pro-

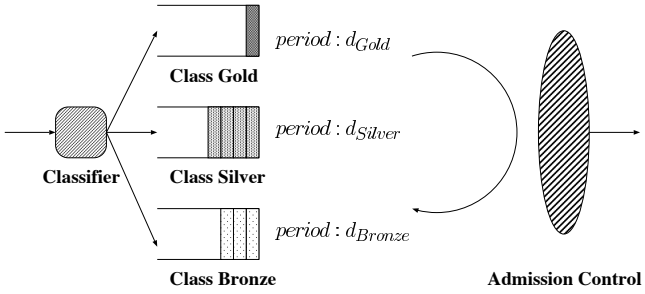


Figure 3: Class-based differentiation for admission control.

gressively 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.

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 $\mathcal{R}$ is defined to be

$$S_{inherent} = \mathcal{R}_{cpu} + \alpha \cdot \mathcal{R}_{data} \tag{3}$$

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 α is an empirically determined constant. Section 3.4 discusses online measurement techniques for estimating these parameters. 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 β 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 3.4. Let η denote the average number of requests (connections) that are currently being serviced by a server in the application's server pool. Then the β 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,

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

where S is the average inherent service time of the requests in a batch, n is the number of servers allocated to the application, and R_{sla} is the desired response time. The term $(\eta + \lceil \frac{\beta}{n} \rceil) \cdot S$ 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., η, plus the number of requests from the batch that might be assigned to the server i.e, $\lceil \frac{\beta}{n} \rceil$. 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 rage associated with the bucket.

As indicated above, the admission control is invoked for each class periodically—the invocation is more frequent for important classes and less frequent for less important classes. This ensures that important requests are always given priority (by being admitted first) over less important requests. Further, it reduces the chances of admitting less important requests into the system and have them deny service to more important requests that arrive shortly thereafter. Let $d_1, d_2, \ldots d_L$ denote the period of invocation for the L classes, $d_1 \leq d_2 \leq \ldots d_L$. 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

$$Q + \left(\eta + \left\lceil \frac{\beta}{n} \right\rceil\right) \cdot S \leq R_{sla} - d_i \tag{5}$$

In the event this condition is satisfied, all requests in the batch are admitted into the system. Otherwise requests in the batch are dropped. Techniques for estimating parameters such as the queuing delay, inherent service time, and the number of existing connections are discussed in Section 3.4.

3.3 Scalable Threshold-based Admission Control

Batch processing eliminates the need to invoke the admission control test for individual requests and reduces the overhead to one test per batch. Whereas this enhances the scalability of the sentry, it may still impose significant processing demands during extreme overloads. In this section, we present techniques for further reducing the admission control overhead. Our technique trades efficiency of the admission control for accuracy and reduces the overhead to a few arithmetic operations per request.

The key idea behind this technique is to periodically *precompute* the number of requests that should be admitted in each class and then simply enforce these limits without conducting any additional per-request tests. The technique estimates the arrival rate in each class and the average service time of a request, and uses these values to pre-compute a threshold for the number of requests that can be admitted in each class. As an example, assume that the system capacity is 100 requests/s, and that an application has two request classes with arrival rates of $\lambda_1 = 75$ requests/s and $\lambda_2 = 50$ requests/s. In this case, the threshold is set so that all requests from class C_1 and only half the requests in class C_2 are admitted. If the incoming request rates change to $\lambda = 150$ and $\lambda = 75$, then a new

threshold is computed that admits two out of every three requests in C_1 and turns away all requests in C_2. Observe that, such a strategy can be implemented very efficiently. Once a request is mapped onto a class, the admission controller only needs to determine if the request is above the threshold in order to admit it (thus the additional overhead beyond request classification is only a few arithmetic operations). A drawback of the approach is that the actual arrival rates may differ from the estimates, resulting in errors—if the threshold is set too low, the SLA may be violated and if it is too high, more requests may be turned away than necessary.

The threshold is defined to be a pair (class i, fraction p_{drop}) and has the following meaning: all requests in classes less important than i should be dropped, requests in class i should be admitted with probability p_{drop}, and all requests in classes more important than i should be admitted. We determine these parameters based on observations of arrival rates and service times for requests of various classes over periods of moderate length (we use periods of length 30sec). 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-pair (i, p_{drop}) is obtained such that

$$\sum_{j=1}^{j=i} \lambda_j \cdot s_j \geq 1 \tag{6}$$

and

$$p_{drop} \cdot \lambda_i \cdot s_i + \sum_{j=1}^{j=i-1} \lambda_j \cdot s_j < 1 \tag{7}$$

Thus, admission control now merely involves applying the inexpensive classification function on a new request to determine its class and then using the equally lightweight thresholding function to decide if it should be admitted.

The threshold-based and the batch-based admission control strategies need not be mutually exclusive. The sentry can employ the more accurate batch-based admission control strategy 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 admission control 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 admission control 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.

3.4 Online Parameter Estimation

The batch-based and threshold-based admission control 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 for admission control. The following statistics are maintained:

- *Arrival rate λ_i:* The arrival rates in each request class are measured at the sentry. 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 measured 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 η:* This parameter is measured at the Cataclysm server. By appropriately instrumenting the OS kernel, the nuclei can 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 careful instrumentation of the OS kernel and some instrumentation of the application itself. This instrumentation —discussed further in Section 6— 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 3).

- Constant α: The constant α in Equation 3 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 α is then empirically determined as the value that minimizes the difference between the actual execution time and the inherent service time in Equation 3.

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 maintains 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 usage values 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 usage values for requests that invoke these scripts may change depending on the arguments. We maintain exponentially decayed averages of their usages.

4 Provisioning for Cataclysms

Although the admission control mechanisms maintain the SLA of admitted requests even during overloads, a significant fraction of the requests may be turned away during extreme overloads. In such situations, the number of requests that are turned away can be reduced by increasing the capacity of the application. The Cataclysm control plane implements a provisioning mechanism to dynamically vary the number of servers allocated to the applications. 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 rates imposes significant processing demands on the existing sentries. The rest of this section discusses our techniques for dynamically provisioning Cataclysm servers and sentries.

4.1 Queuing-theoretic Model for Replicable Applications

The dynamic provisioning problem can be formulated as a *constrained optimization problem* whose goal is to determine a partitioning of the servers among the hosted application that will maximize the platform's expected revenue. The key challenge in the formulation of this optimization problem is that of determining the utility of assigning a certain number of servers to an application. Utility-based provisioning has been studied in the Muse system [9], where an economic approach is used for formulating the optimization problem. Chandra et al. [8] model a server resource that services multiple applications as a GPS system and devise an optimization problem to derive resource allocations. Our approach is also an instance of utility-based provisioning where we use a queuing-theoretic model for an application.

We use a well known queuing theory result to devise a model for determining the utility of a server set of a given size for a replicable application under a certain workload [20]. 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. The following upper bound is known on the average response time for a G/G/1 queuing system:

$$E[R] \leq E[S] + \lambda \cdot \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (1 - \rho)} \tag{8}$$

Here $E[R]$ is the average response time, $E[S]$ is the average service time, λ is the request arrival rate, $\rho = \lambda \cdot E[S]$ is the utilization, and σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time respectively. Rewriting inequality (8), we have the following:

$$\lambda \geq \left[E[S] + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (E[R] - E[S])} \right]^{-1} \tag{9}$$

The above inequality gives a lower bound on the request arrival rate for which one application replica will be able to provide an average response time of $E[R]$. We denote this by λ_{in} henceforth. We define the *utility* of n servers for an application a, $U(a,n)$ as the expected revenue that n servers would generate for s. Recall from the SLA defined in Section 2 that to determine this, we need to predict the overall arrival rate to the service during the next provisioning cycle (simply cycle henceforth). Having predicted this rate λ_{pred}, the response time target $E[R]$ is read from the entry in the QoS table of the SLA corresponding to an arrival rate of λ_{pred}. We use the Revenue Table of the SLA to determine the revenue per admitted request r for this arrival rate. Thus we have

$$U(a,n) = n \cdot \lambda_{in} \cdot r \cdot T_{prov} \tag{10}$$

T_{prov} denotes the length of a cycle. Also, because of the lower bound k_{min} on the number of servers assigned to s, we only need to compute $U(a,n)$ for n in the following range

$$k_{min} \leq n \leq \min_p \left\{ N, p \geq \frac{\lambda_{pred}}{\lambda_{in}} \right\} \tag{11}$$

Here N is the total number of cataclysm servers in the cluster. These utility curves are then used to determine the partitioning of the cataclysm servers among applications that would maximize the expected revenue over the next cycle. This is achieved by solving the following constrained optimization problem

$$\text{Maximize: } \textstyle\sum_a U(a, n_a)$$

$$\text{Subject to: } \textstyle\sum_a n_a \leq N$$

N denotes the number of host-servers in the platform. Figure 4 outlines the steps involved in determining this optimal partitioning. The final step in this procedure is solving the above optimization problem. For the replicable applications that we consider in our work (see Section 2.2), the utility curves are linear. A simple greedy heuristic can be used to solve this problem optimally. This heuristic begins by assigning to each application a number of servers equal to the lower bound guaranteed in its SLA. It then proceeds in steps, each resulting in assigning one additional server to some application. In each step, the heuristic determines the application that would experience the most gain in revenue due to an additional server and that has not yet reached its upper bound on number of servers. This process continues till (1) all the servers have been assigned or (2) all applications have reached their upper bounds or (3) no application would experience a positive gain in revenue due to an additional server.

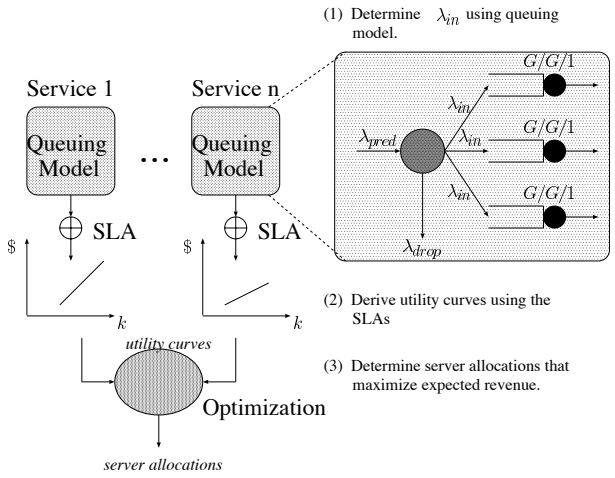


Figure 4: Dynamic provisioning in the control plane.

4.2 Sentry Provisioning

In general, allocation and deallocation of sentries is significantly less frequent than that of Cataclysm 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 Integrating Policing and Provisioning

A novel feature of our hosting platform is the cooperation between the admission controller and the provisioning mechanism in the control plane. The two mechanisms operate at different time scales—while the admission control handles overloads at request arrival time scales (milliseconds), the provisioning mechanism handles overloads at the time scale of minutes. Nevertheless, cooperation between these mechanisms enhances the capability of the platform in handling overloads.

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 previous time period to reallocate servers to applications if necessary. 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 uses recent workload measurements for the overloaded application to recompute its utility curve (unlike the predictive mode where all utility curves are recomputed, only the curve for the overloaded application is recomputed in the reactive mode). The provisioning mechanism then allocates additional servers to the overloaded application. Undesirable oscillations in such allocations are prevented using two constraints: (i) a limit of Δ 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 δ time units is imposed on the duration between two successive invocations of the provisioning mechanism in the reactive mode (δ is set to 5 minutes in our prototype).

Observe that, while provisioning of servers to applications can be both predictive and reactive, provisioning of sentries is always purely reactive—allocation of sentries is a less frequent event than the allocation of servers and it suffices to do so in a purely reactive fashion.

Reactive provisioning involves communication from the sentry to the control plane. Communication from the control plane to the sentry is also present in our hosting platform. Clearly, after each provisioning step, the control plane must provide the new allocations to each sentry. In addition, the provisioning mechanism conveys the response time targets to the sentries after each invocation. Recall that the QoS table in the 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 admission controllers. Thus, these interactions enable integration of admission control, provisioning and adaptive performance degradation in the hosting platform. We experimentally demonstrate various aspects of this integration in Section 7.

6 Implementation Considerations

We implemented a prototype Cataclysm hosting platform on a cluster of 20 Pentium machines connected via a 1Gbps ethernet switch and running Linux 2.4.20. Each machine in the cluster was used for running 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.

6.1 Cataclysm Sentry

We used *Kernel TCP Virtual Server* (`ktcpvs`) version 0.0.14 [22] to implement the policing mechanisms described in Section 3. `ktcpvs` 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 connection from a client) and transparently relays data between these. We modified `ktcpvs` to implement all the sentry mechanisms described in Sections 3-5.

`ktcpvs` 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 `ktcpvs` 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. [2] 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 3) 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 imposed lower and upper bounds of 80 and 2560 respectively on the number of relayer threads in our experiments.

[2] 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.

6.2 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 over these sockets. The provisioning algorithm can be invoked in one of two ways described in Sections 4 and 5. 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 remotely logs on to the nodes running the sentries and uses `tcpvsadm` with the appropriate arguments to affect the applications' server sets in this partitioning. In our experiments, we chose the length of the default provisioning duration to be 30 min.

7 Evaluation

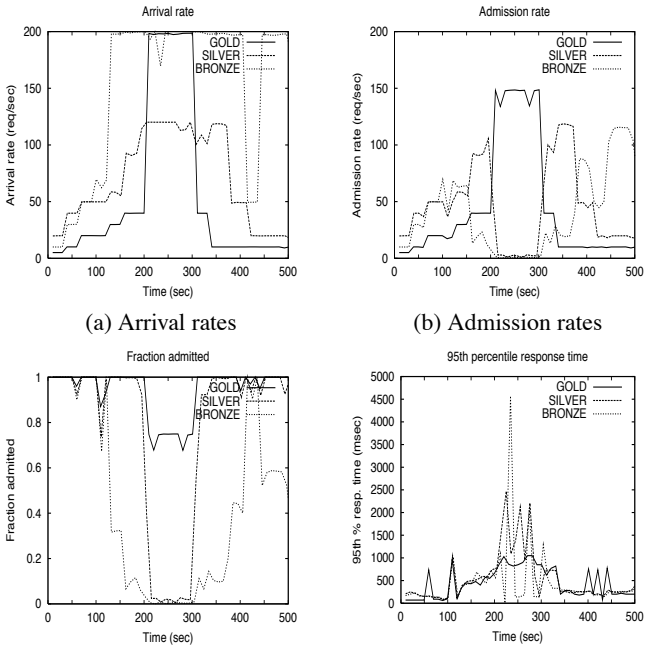
In this section we present the experimental setup followed by the results of our experimental evaluation.

7.1 Experimental Setup

Our prototype was built on a cluster of 20 Pentium machines connected by a 1Gbps Ethernet switch. The cataclysm sentries were run on dual-processor 1GHz machines with 1GB RAM. Cataclysm provisioning was run on a dual-processor 450MHz machine with 1GB RAM. The machines used as cataclysm 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 the 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, these web servers hosted PHP scripts worth different amounts of CPU computation. The dynamic component of the workload of these applications consisted 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 make 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 2 was used for the applications.

Recall from Section 3 that knowing a request's CPU requirement was crucial for the determination of its inherent



(a) Arrival rates (b) Admission rates

(c) Fraction admitted (d) 95th resp. time

Figure 5: Demonstration of class-based differentiation achieved by the admission control.

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()`.

The workload generators for these applications were based on `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 enhance `httperf` to record a specified percentile of the response time experienced by the requests during a run.

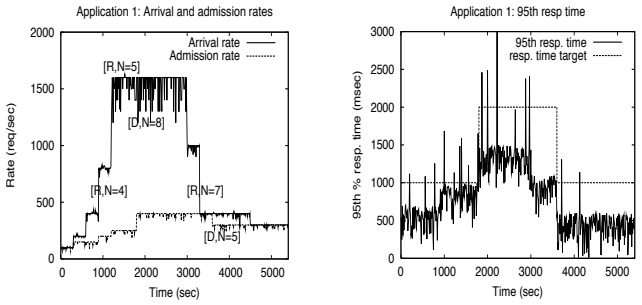
7.2 Class-based Differentiation

Our first experiment investigates the efficacy of the mechanisms employed by the Cataclysm sentry to provide class-based differentiation to requests. 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 significance. The class of a request could be uniquely determined from its URL. The invocation period of the admission control was set to 0, 50 and 100 msec for the three classes respectively.

The workload consisted of requests for members of a set of PHP scripts. The capacity of each Apache server for this workload (i.e., the request arrival rate for which the 95^{th} 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 5(a) shows the workload used in this experiment. Request arrival rates for the three classes started at low values but gradually picked up to create an overload, first increasing at a slow and steady rate and then suddenly attaining a high value. The durations during which the peak loads for the three classes persisted were chosen to have the *nested* structure as seen in Figure 5(a)—arrival rate of Bronze requests was the first to peak and the last to drop down, Silver requests peaked next etc—to clearly bring out the class-based differentiation achieved by the sentry. Figures 5(b) and 5(d) show the rates at which these requests were admitted and the 95^{th} percentile response time of admitted requests respectively during the 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 5(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 5(d) depicts the performance of admitted requests. We find that the sentry was very successful in maintaining the response time below 1000 msec. From the admission rates shown in Figure 5(b) we find that the admission control was somewhat conservative and admitted requests at slightly lower rate than that capacity revealed by offline measurement of system capacity. Observe that the few instances when the 95^{th} percentile response time for some class was worse than 1000 msec correspond to when very few requests of that class were admitted implying there was not enough data for these observations to be considered statistically significant.

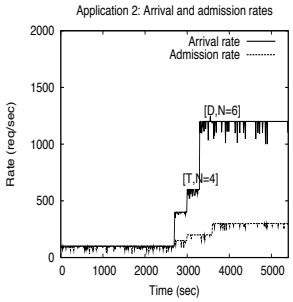
7.3 Cataclysm Provisioning

We conducted an experiment with two web applications hosted on our Cataclysm platform. The control plane ran on a dedicated node. Each application had its sentry running on a separate node. 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 2. Further, the SLAs imposed a lower bound of 3 on the number of servers that each application could be assigned. The workload generators for the applications ran on four nodes in the cluster. The default provi-

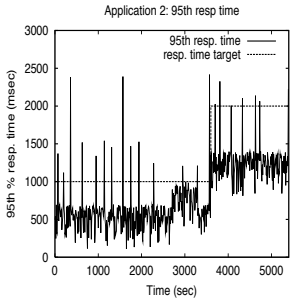


(a) Arrival and admission rates

(b) 95th resp. time



(c) Arrival and admission rates



(d) 95th resp. time

Figure 6: 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.

sioning duration used by the control plane was 30 min. In this experiment, when migrating a server from one application to another, we did not have the control plane actually reboot the server and install the data for the new application. We installed all the content served by the two applications on all the cataclysm servers in the cluster. Therefore, migration was achieved simply by having the control plane send messages to the sentries for the donor and the receiver applications of the server being moved. As a result, server migration time was negligible.

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, corresponding to the lower bound in their SLAs. The free pool contained 5 servers at this time. A value of 50% was used for the threshold f_{drop} by the sentries. Figures 6(a) and 6(c) depict the arrival rates to the two applications. The request arrival rate for Application 2 was kept at a sustainable 100 requests/sec. 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). Also shown in Figure 6(a) are the working of the sentry for Application 1 and the provisioning. At t=910 sec the sentry, having observed more than 50% of the requests being dropped over the last 5 min, triggered the provisioning algorithm as described in Section 5. The provisioning algorithm responded by pulling one server from the free pool and adding it to Application 1. At t=1210 sec, the continuously increasing rate of arrivals to Application 1 resulted in the addition of another server from the free pool. Observe in Figure 6(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 2).

In the second half of the experiment, the overload of Application 1 subsided and Application 2 got overloaded. Figure 6(c) depicts the arrival and admission rates for Application 2. Till t=2710 sec, Application 2 was well-provisioned for its workload and its sentry admitted almost all the requests. After this, the service was exposed to increasingly high arrival rates peaking at approximately 1200 requests/sec that sustained for the remainder of the experiment. The functioning of the provisioning was qualitatively similar to when Service 1 was overloaded. The sentry of Application 2 triggered the provisioning at t=3190 sec resulting in one server being moved from Application 1. At t=3600 sec, the second default invocation of provisioning occurred. This resulted in 2 more servers being moved from Application 1 to Application 2. The response time target for Application 1 was reduced back to 1000 msec because its arrival rate was predicted to be less than 1000 requests/sec, while that for Application 2 was degraded to 2000 msec.

Figures 6(b) and 6(d) show the 95^{th} percentile response times for the two services during the experiment. There are two key observations. First, 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. Secondly, 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).

7.4 Scaling to Handle Extreme Overloads

We conducted an experiment to investigate the improvement in the scalability of the sentry due to the threshold-based 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

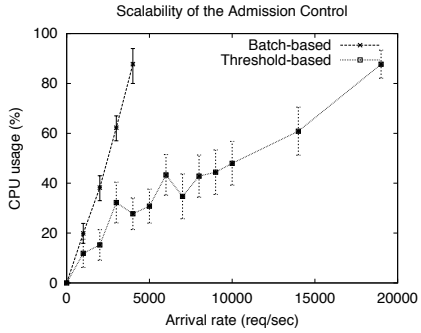
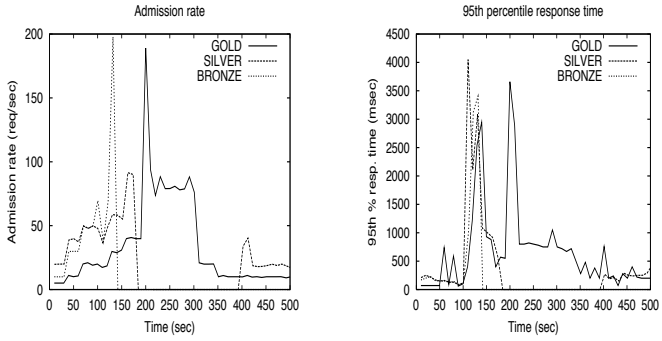


Figure 7: Scalability of the admission control.

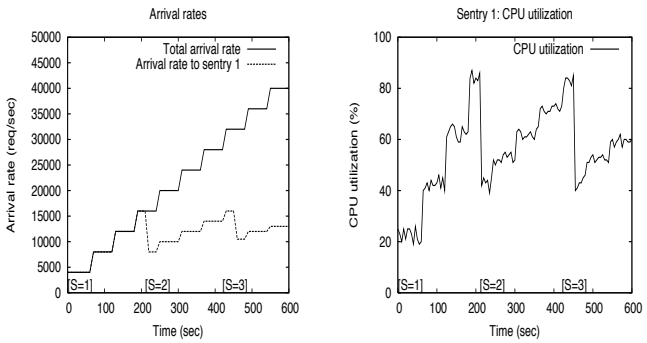


(a) Admission rates (b) 95th percentile response time

Figure 8: Performance of the threshold-based admission control. At t=135sec, the threshold was set to reject all Bronze requests; at t=180sec, it was updated to reject all Bronze and Silver requests; at t=210sec it was updated to also reject Gold requests with a probability 0.5; finally, at t=390sec, it was again set to reject only Bronze requests.

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.

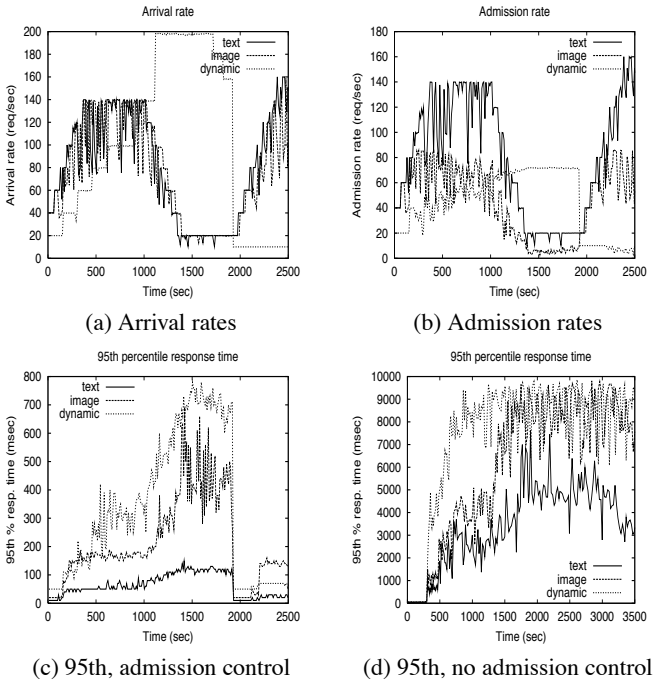
A second experiment was conducted to investigate the degradation in the decision making due to the threshold-based admission controller. We repeated the experiment reported in Section 7.2 (Figure 5) but forced the sentry to employ the threshold-based admission controller. The thresholds used by the admission control were computed once every 15 sec. Figure 8(a) shows changes in the admission rates for requests of the three classes. At t=135 sec, after observing the high arrival rate of Bronze requests, the threshold was set to reject all Bronze requests. Following this, at t=180 sec, when requests to all three classes were observed to arrive at high rates, the threshold was recomputed to reject all Bronze and Silver requests. At t=210 sec it was made tighter—all Bronze and Silver requests were dropped while Gold requests were admitted with a probability of 0.5. At t=390 sec, after the overload due to Gold and Silver requests had subsided, the threshold changed back to reject only Bronze requests. 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 and 190-210 sec) 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.



(a) Arrival rates (b) CPU utilization: Sentry 1.

Figure 9: Dynamic provisioning of sentries. [S=n] means the number of sentries is n now.

Finally, we conducted an experiment to demonstrate the ability of the system to dynamically provision additional sentries to a heavily overloaded service. Figure 9 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 9(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 9(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.



(a) Arrival rates (b) Admission rates

(c) 95th, admission control (d) 95th, no admission control

Figure 10: Revenue maximization via size-aware admission control.

7.5 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 (1kB-8kB), image files (64kB-128kB) and dynamic requests for a collection of PHP scripts. Figures 10(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 sec and t=1250 sec due to the increased arrival rates of both static and dynamic, second between t=1250 sec and t=1900sec due to high incidence of the dynamic requests and finally after t=1900 sec 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 3.4, 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 10(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 10(d).

8 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. [33] 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 [35] 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 [17] 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. [16] 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 [11] 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. In [19], 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. Li and Jamin [23] 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 [21] 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 [32] 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. In [14], the authors 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 [3] 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 [9] 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* [4] 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)* [10] 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. [8] 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. In [30] the authors address the problem of providing resource guarantees to distributed applications running on a shared hosting platform. [31] proposes 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. In [5] 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. [7] provides 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. In [18] Kamra et al. use an M/GI/1 processor sharing queue as an abstraction for a 3-tier e-commerce application.

SLAs and Adaptive QoS Degradation: The WSLA project at IBM [34] 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.

9 Conclusions and Future Work

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 an overload management solution that brought together the techniques of admission control, dynamic provisioning of the platform's resources and adaptive degradation of QoS into one integrated system. Cataclysm provides several desirable features under overloads, such as preferential admission of more important requests, the ability to handle diverse workloads and revenue maximization at multiple time-scales via dynamic provisioning of resources and size-based admission control. The cataclysm sentry can transparently tradeoff 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 the benefits of our integrated approach using a variety of workloads.

As part of future work, we plan to extend our overload management techniques to complex, multi-tiered Internet applications.

References

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

[2] M. Andreolini, M. Colajanni and M. Nuccio. Scalability of content-aware server switches for cluster-based Web information systems. In Proc. of 12th Int'l World Wide Web Conf. (WWW2003), Budapest, Hungary, May 2003.

[3] K. Appleby, S. Fakhouri, L. Fong, M. 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.

[4] 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.

[5] 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.

[6] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. In IEEE Network, vol. 13, no. 5, pp. 64-71, Sept. 1999.

[7] 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 Volume 34 , Issue 2*, Pages 263-311, June 2002.

[8] 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.

[9] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[10] J. Chase, L. Grit, D. Irwin, J. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *the Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[11] 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, May 31-June 4, 1999.

[12] Self-similarity in World Wide Web traffic: evidence and possible causes. In *Proceedings of the 1996 ACM SIGMETRICS*, Philadelphia, PA.

[13] R. Doyle, J. Chase, O. Asad, W. Jin and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility In Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03), 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. Submitted for publication.

[15] J. Hellerstein, F. Zhang and P. Shahabuddin. A Statistical Approach to Predictive Detection. In *Computer Networks*, January 2000.

[16] R. Iyer, V. Tewari and K. Kant. Overload Control Mechanisms for Web Servers. In *Performance and QoS of Next Generation Network*, Nagoya, Japan, November 2000.

[17] H. Jamjoom, J. Reumann and K. Shin. QGuard: Protecting Internet Servers from Overload. Technical Report, Department of Computer Science and Engineering, University of Michigan, CSE-TR-427-00, 2000.

[18] A. Kamra, V. Misra and E. Nahum. Controlling the Performance of 3-Tiered Web Sites: Modeling, Design and Implementation. Submitted for publication.

[19] V. Kanodia and E. Knightly. Multi-class Latency-bounded Web services. In *Proceedings of IEEE/IFIP IWQoS 2000*, Pittsburgh, PA, June 2000.

[20] L. Kleinrock. Queuing Systems, Volume 2: Computer Applications. *John Wiley and Sons, Inc*., 1976.

[21] E. Knightly and N. Shroff. Admission Control for Statistical QoS: Theory and Practice. In *IEEE Network*, 13(2):20-29, March/April 1999.

[22] Kernel TCP Virtual Server. http://www.linuxvirtualserver.org.

[23] K. Li and S. Jamin. A measurement-based admission-controlled web server. In *Proceedings of INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[24] J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase. Managing Mixed-Use Clusters with Cluster-on-Demand. Cluster-on-Demand Draft, Internet Systems and Storage Group, Duke University.

[25] D. Mosberger and T. Jin. httperf — 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, California, October 1998.

[27] S. Ranjan, J. Rolia and E. Knightly, IWQoS 2002. QoS-Driven Server Migration for Internet Data Centers. In *Proceedings of IEEE IWQoS 2002*.

[28] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. In *18th International Teletraffic Congress*, Berlin, Germany, 2003.

[29] The Internet Under Crisis Conditions: Learning from September 11. Committee on the Internet Under Crisis Conditions: Learning from September 11, NRC.

[30] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, January 2004, Vol. 15, No. 1.

[31] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[32] A. Verma and S. Ghosal. On Admission Control for Profit Maximization of Networked Service Providers. In Proc. of 12th Int'l World Wide Web Conf. (WWW2003), Budapest, Hungary, May 2003.

[33] T. Voigt, R. Tewari and D. Freimuth. Kernel mechanisms for service differentiation in overloaded web servers. In Proceedings of the USENIX Annual Technical Conference, 2001.

[34] Web Service Level Agreements (WSLA) project. http://www.research.ibm.com/wsla

[35] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03), March 2003.