Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds

Abhishek Chandra and **Prashant Shenoy**

Department of Computer Science
University of Massachusetts Amherst
{abhishek,shenoy}@cs.umass.edu

Abstract

Internet data centers host multiple Web applications on shared hardware resources. These data centers are typically provisioned to meet the expected peak demands of the hosted applications based on normal time-of-day effects. Such an over-provisioning approach is not robust to flash crowd scenarios, where the load increase of some hosted applications is much higher than their expected peak loads. In such scenarios, data centers can utilize their resources better by employing dynamic resource allocation. In this paper, we present a prototype data center implementation that we use to study the effectiveness of dynamic resource allocation for handling flash crowds with different characteristics. This prototype implements a multi-tiered server architecture along with mechanisms for monitoring, load detection, load balancing and dynamic allocation. Our experiments with this prototype show that a carefully designed dynamic allocation scheme can be effective for handling flash crowds. We show that in order to handle very sharp growth in loads, a dynamic allocation scheme must be either extremely responsive or employ low overhead mechanisms such as using hot spare servers. On the other hand, gradually increasing flash crowds can be handled equally well with larger overheads and slower reaction times. We also show that even in the presence of large allocation overhead, it is possible to achieve the same application performance by either allocating multiple servers simultaneously or allocating a few servers often. Using our results, we conclude that even without large-scale over-provisioning, it is possible to effectively handle flash crowd conditions using a dynamic allocation scheme that responds quickly to workload changes, and that can mask large allocation overheads either by deploying a few ready servers or by allocating multiple servers simultaneously.

1 Introduction

Internet data centers host multiple Web applications on a common hardware platform. The workloads of the hosted applications vary over time due to long-term periodic trends such as *time-of-day effects* (e.g., more people surfing the web during lunch hours), and also due to *flash crowds* (e.g., "Slashdot effect" or a breaking news story). Several techniques have been proposed to predict long-term time-of-day kind of workload variations [12, 26, 32] and most data centers are well-provisioned to handle such variations. However, flash crowds are generally unpredictable and have characteristics different from typical time-of-day load fluctuations. For instance, on September 11, 2001, the load on the CNN website doubled every 7 minutes to reach a peak of almost 20 times the normal load [18]. In particular, the transient load and the duration of flash crowds is difficult to predict for long-term provisioning, and the system has to be largely *reactive* to the arrival of a flash crowd relying on extremely short-term predictions [17].

To handle flash crowds, data centers can either resort to high levels of over-provisioning, or employ dynamic resource allocation, wherein resources can be borrowed from unused servers or underutilized applications to service flash crowds. While over-provisioning of resources is a simple approach, it has two main drawbacks. First of all, by their very definition, flash crowds are unpredictable and it is not easy to predict their peak load, so any imperfectly provisioned system is likely to fail under sustained overload conditions. The other alternative of massive over-provisioning can lead to severe under-utilization of resources and excessive power usage during the normal operational period.

Recent studies have shown the statistical multiplexing benefits of dynamic resource allocation over over-provisioning in a data center [3, 7]. These drawbacks of over-provisioning make it lucrative to use dynamic resource allocation to handle the relatively uncommon events of flash crowd conditions. As a result, several dynamic resource allocation schemes have been proposed to better utilize the resources in such scenarios [5, 6, 8, 20, 24]. These dynamic allocation schemes react to changing application loads by reallocating resources to overloaded applications, borrowing these resources from other under-utilized applications if necessary.

While most of the recent studies have focused on the resource utilization and provisioning gains of dynamic allocation, there has been little investigation of the effect of dynamic allocation on application performance in the presence of flash crowd conditions. There has been no definitive study of how much the application performance suffers due to the overheads and reaction time of dynamic allocation.

In particular, the following questions need to be answered.

- How effective is dynamic allocation in maintaining application performance under flash crowd conditions?
- What characteristics and parameters must a dynamic allocation scheme employ to meet application performance needs?

In this paper, we make the following contributions. First of all, we present a prototype data center that we have implemented to conduct an experimental study for answering these questions. In this prototype, we have implemented a multi-tiered server architecture along with mechanisms for online monitoring, load detection, load balancing and dynamic allocation. This infrastructure provides us with the ability to systematically vary the parameters of an online dynamic allocation scheme. In addition, it enables us to generate real Internet application workloads and provides us with the ability to study application performance metrics under different load conditions.

As our experimental methodology, we generate flash crowd conditions for an Internet application benchmark and perform online reallocation of server resources. In our experiments, we systematically vary the parameters and overheads of dynamic allocation to study their impact on application performance. Based on our experimental results, we show that dynamic resource allocation is effective in handling flash crowds if it uses appropriate allocation parameters. These parameters include fast responsiveness to workload changes, and an ability to mask large allocation overheads by allocating a few ready servers or multiple servers simultaneously.

The remainder of this paper is organized as follows. In Section 2, we provide background on data centers and dynamic resource allocation, and define the characterization of flash crowd scenarios. We then describe our prototype data center implementation in Section 3 followed by our experimental results in Section 4. We discuss related work in Section 5 and finally present our conclusions and future work in Section 6.

2 Background

In this section, we first provide background on Internet data centers, followed by a description of the parameters used for dynamic resource allocation in such data centers. Finally, we formally define a characterization of flash crowds.

2.1 Dynamic Resource Allocation in Data Centers

Since data centers host multiple applications on a common server platform, they can dynamically reallocate resources among different applications. Several allocation schemes have been proposed [5, 6, 8, 20, 26] that perform reallocation on such platforms. Most of these schemes use a *measure-detect-allocate* cycle for reallocation, wherein they monitor the application workloads, detect any overload conditions, and then allocate resources to overloaded applications. However, different schemes vary in their underlying architecture, resulting in a difference in the allocation mechanisms they employ. For instance, Oceano [5] allocates whole server machines among applications, while MUSE [8] allows multiple applications to be co-hosted on the same server. Cluster-On-Demand [20] is a hybrid scheme that allows clusters to be allocated to virtual data centers that further share the servers among multiple applications.

In this paper, we focus on *dedicated* data center architectures like Oceano where whole server machines are allocated to individual applications, These architectures exhibit the widest choice of values for the allocation parameters that we investigate in this paper (described below) to answer the questions raised in the previous section[1]. Dedicated architectures partition their servers into different server pools, so that at any given time, these servers are in use by multiple hosted applications in the following manner.

- *Private pool:* Each application has its own set of pre-provisioned servers, that are statically allocated to it. These servers guarantee a minimum level of application performance and ensure performance isolation from other overloaded applications.
- *Borrowed pool:* Each application can add some servers to its pool dynamically based on its load. These servers are not guaranteed to stay in a single application pool, and can move between different application pools. These servers are useful for meeting overload conditions.
- *Free pool:* This is a set of server machines that are not currently allocated to any application. These can be dynamically allocated to any application when the need arises.

In what follows, we refer to an application to which a server is allocated as the *recipient*, while an application from which a server is deallocated is referred to as the *donor*. A dynamic allocation algorithm reallocates resources by moving servers between the free pool and the borrowed pools of different applications, or between borrowed pools of applications if the free pool is empty.A server allocated to the borrowed pool of a recipient application can be classified to be in one of the following states based on which pool it comes from.

- *Hot spare:* A hot spare is a server that already has the recipient application running and ready to receive requests. Such servers can be allocated by simply redirecting load to them, and hence can be brought online quickly. A hot spare could be a server already in the recipient's borrowed pool[2], or a recently freed machine from the recipient borrowed pool that still has the application running.
- *Clean spare:* A clean spare is an unused server in the free pool, but it is not ready to run a specific application. Such servers could be powered off to save power, or they might be up and running, waiting to be assigned to an overloaded application. To allocate a clean spare to an application, we need to install the recipient application on the server and bring it to a running status before load could be redirected to it.
- *Dirty spare:* A dirty spare is a server that is currently serving requests from another application. It thus resides in the borrowed pool of a donor application at the time of reallocation. A dirty spare is typically used only when the free pool is empty and the donor application is sufficiently under-utilized. Reallocation of such a server involves gracefully shutting down the currently running application on the server, clearing its application state (which might involve actions such as disk scrubbing), rebooting, installation of the new application and getting the new application to a ready state.

As we will see, the server state at the time of allocation plays an important role in determining the effectiveness of an allocation scheme. Next, we describe a set of parameters that can be used to characterize dynamic allocation schemes within the above framework.

2.2 Characterizing a Dynamic Resource Allocation Scheme

The effectiveness of a dynamic resource allocation scheme depends on a large number of factors, such as the resource allocation algorithm, the granularity of allocation, the allocation overheads, etc. [7]. In our study, we take an algorithm-independent approach in describing a dynamic allocation scheme. We define a set of orthogonal parameters to characterize an allocation scheme, which we describe next. These parameters are illustrated in Figure 1.

[1] Shared and hybrid architectures exhibit several other interesting features that are beyond the scope of this paper.

[2] The hot spare allocation is implicit in this case. Here, the recipient server pool would be able to absorb some of the flash crowd load before getting overloaded.

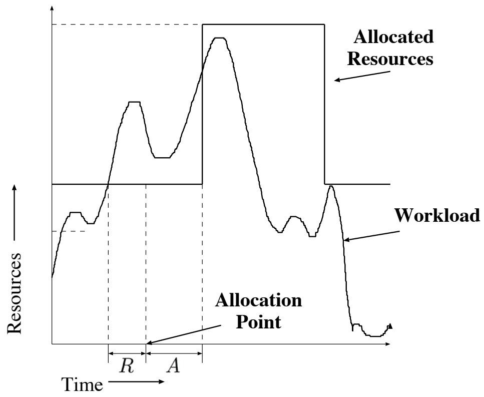


Figure 1: Characteristics of a dynamic allocation scheme

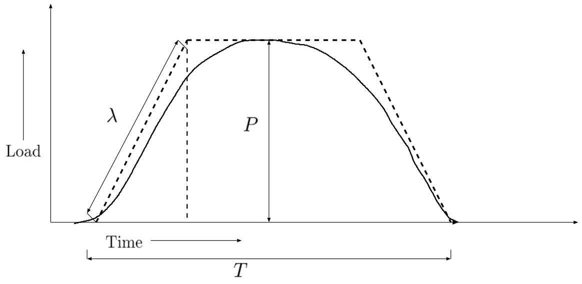


Figure 2: Flash Crowd Characteristics

Responsiveness (R)

Responsiveness is defined as the duration it takes for the allocation scheme to detect a load change and determine the new allocation. The responsiveness of an allocation scheme could depend on several factors – the frequency of monitoring the resources, the overhead of collecting monitoring statistics and load detection, and the runtime of the allocation algorithm. For instance, a simple threshold-based load detection scheme would have much smaller overhead compared to a scheme that uses a complex function of multiple load metrics to determine load changes. In a periodic allocation scheme, responsiveness would correspond to the reallocation interval.

Allocation Overhead (A)

This is the amount of time it takes to actually bring the new resources online once the allocation decision has been taken. This overhead could include such costs as de-allocating another application's server (in case of dirty spares), redirecting new requests to the new server, etc.

Based on our definition of the server states in Section 2.1, we see that the allocation overhead differs for various server states. For instance, the overhead of allocating a hot spare is of the order of a few milliseconds. On the other hand, allocation overhead for a clean spare is of the order of several seconds, while that for a dirty spare can be in the order of several minutes.

2.3 Characterizing Flash Crowd Scenarios

Having described parameters of dynamic allocation in a data center, we now describe how to characterize flash crowd conditions. Flash crowds are characterized by an atypical increase in load. Further, these loads last for relatively short

Machine type	Hardware configuration	Operating System	Application
Client	1 GHz/512 MB	Linux Redhat 7.2 kernel v2.4.20	httperf
Load Balancer	2.8 GHz/1 GB	Linux Redhat 7.3 kernel v2.4.20	ktcpvs v0.0.14
Web Server	864 MHz/256 MB	Linux Redhat 7.2 kernel v2.4.7	Apache v1.3.28
Database Server	2.8 GHz/512 MB	Linux Redhat 7.3 kernel v2.4.20	Mysql v max-3.23.57

Table 1: Cluster Configuration

durations of time (compared to the normal operational period of an application). However, flash crowds can differ from each other in several respects. For instance, some flash crowds exhibit very sharp growth rates, while some are comparatively long-lived and some have very high magnitude peaks. We characterize a flash crowd based on the following defining characteristics, that are illustrated in Figure 2.

- *Load growth rate (λ):* This is the rate at which the load increases before the flash crowd condition either subsides or stabilizes. An abnormal increase in the rate of workload arrival for an application might be used to detect the arrival of the flash crowd. This parameter relates to the speed at which the data center needs to detect and react to the flash crowd. λ could be measured using units such as user-sessions/second or requests/second.

- *Peak load (P):* This is the maximum load imposed by the flash crowd. It can be expressed in workload units (such as number of user sessions), or in terms of server resources (for instance, number of servers) required to service it. It relates to the maximum resource capacity required to successfully handle the flash crowd.

- *Flash crowd duration (T):* This is the duration for which the flash crowd condition lasts. This is the length of the time period from the arrival of the crowd till the load goes down to normal. This would correspond to the duration of "abnormality" for the application. A very large value of T (for example, of the order of days, or several hours) could indicate a basic shift in the application workload levels, and might necessitate long-term re-provisioning of resources.

Having provided background on data centers and characterized various allocation parameters and flash crowd conditions, we now present our prototype data center implementation.

3 Prototype Data Center Implementation

In this section, we describe the implementation of our prototype data center on a testbed cluster. The cluster consists of 16 Pentium machines connected by a 1Gbps ethernet switch. We divide the cluster into two sets — a *client set* to generate Web workloads, and a *data center* that hosts Internet applications servicing these workloads. We have implemented a multi-tiered server architecture in our data center prototype along with online monitoring, load balancing and dynamic allocation mechanisms. Table 1 shows the hardware and software configuration of the machines in the cluster. Next, we describe these components in more detail.

3.1 Multi-tiered Server Architecture

Figure 3 shows the server architecture of our prototype data center that enables it to host multi-tiered Internet applications. Our data center employs the following component servers:

- *Load balancer:* The load balancer in a data center environment is typically a layer-4 (IP layer) or a layer-7 (application layer) switch. It is responsible for redirecting incoming requests to different back-end servers based on criteria such as the back-end server loads, the requested URL, the client IP address, etc.

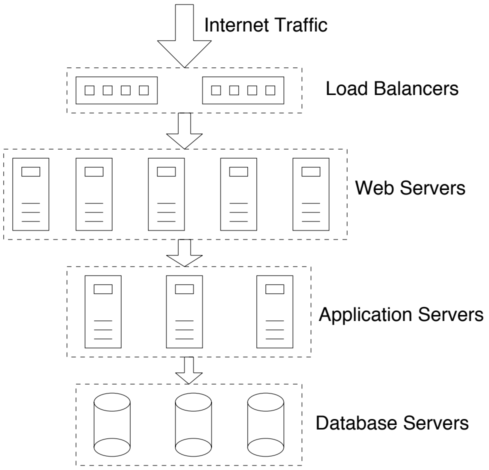


Figure 3: Multi-tiered Server Architecture

In our implementation, we use *Kernel TCP Virtual Server (ktcpvs)* version 0.0.14 [16] as the front-end load balancer. Ktcpvs is an open-source layer-7 load balancer that forwards incoming user connections to back-end servers using separate TCP connections.

- *Web Server:* The load balancer hands off requests to Web servers that handle functions such as http processing and serve up static content such as images and static web pages.

  In our implementation, we used *Apache* version 1.3.28 [4] as our Web server. Apache was enabled with SSL and PHP support and had a MaxClient limit set to 256.

- *Application Server:* Application servers run behind the Web server tier and execute dynamic scripts serving up dynamic content. These servers could be running small scripts or large enterprise application server programs.

  In our experiments, we used a PHP-enabled Internet application (described later) that ran dynamic PHP scripts on the Apache server machines. In this scenario, the Web and the application servers ran on the same machines, and there was no physical separation of the two tiers. In the rest of the paper, we refer to the combined Web/application servers simply as Web servers.

- *Database Server:* The last tier in the architecture consists of database servers that store most of the application data and process queries from the application servers for the execution of dynamic scripts. We used MySQL version max-3.23.57 [22] — a popular open-source database server — as the back-end database server in our prototype.

Dynamic resource allocation can be performed within each of these tiers individually, because of their functional differences and also because their bottleneck resources can be different. For instance, the network bandwidth or number of connections is more likely to become the bottleneck on the Web servers, while the disk bandwidth might be the bottleneck for the database servers. Also, the degree of replication and reallocation mechanisms differ between different tiers. In our experimental study, we employ dynamic resource allocation on the Web server tier. Through experimental evaluation, we confirm the Web server tier to be the bottleneck tier. We next describe the implementation of dynamic resource allocation in our data center.

3.2 Dynamic Resource Allocation

In this section, we describe our implementation of dynamic resource allocation on the data center configuration we described above. Dynamic resource allocation requires monitoring of server resources, overload detection and reallocation. We describe each of these in detail below.

3.2.1 Monitoring and Load detection

We installed a Linux-based monitoring software – *sar* version 4.0.1 [27] – on all the cluster machines. Sar is part of an open-source system monitoring package called sysstat, that can periodically measure various system metrics like CPU usage, run queue length, memory usage, number of open sockets, network traffic, etc. We used a monitoring period of 10 seconds in our experiments.

We used a *reactive* allocation scheme that relies on the current workload arrival rate to detect overload conditions. Note that such a scheme is different from a *proactive* allocation scheme, as there is no active prediction being done for future time intervals. A reactive scheme is essential to handle flash crowds, because their long-term prediction is difficult. We measured the incoming workload rate by monitoring the number of user connections being opened at the front-end load balancer. Using a sequence of workload measurements, we did overload/underload detection periodically and performed reallocation accordingly. We emulated the allocation parameter of responsiveness R (as defined in Section 2.2) by varying the frequency with which this overload detection was done. Thus, the higher the overload detection frequency, the more responsive our system was to changing workload levels.

To detect overloads and underloads in the system, we used a *threshold-based* scheme that was used to trigger dynamic allocation. The goal of this scheme was to maintain the average load on the allocated Web servers under a threshold value. The way this load detection worked was as follows. Based on the current load, the load detector computed the minimum number of servers required to maintain the average load on these servers under the threshold. If the number of currently allocated servers was less than this value, then the requisite number of additional servers was allocated. On the other hand, if the currently allocated servers exceeded the requirement, then the extra servers were deallocated. In our experiments, we used a threshold value of 50%. Thus, for instance, if the load on the system was 200%, it implied a system requirement of 4 servers for handling the load at an average load of 50% each. If the actual number of allocated servers was 2, then we allocated 2 extra servers, while if the actual number was 6, we deallocated 2 servers. Note that by using different values for the overload and underload thresholds, it is easy to build hysteresis in server allocation to avoid oscillations under rapid load fluctuations.

3.2.2 Resource Allocation and Load Balancing

We implemented a centralized allocation scheme at the load balancer. The load balancing software ktcpvs allows adding and removing servers to a list of available servers using user commands. The load balancer then forwards incoming user sessions and requests only to the servers on its available list. We used this facility for performing dynamic resource allocation. We maintained a list of currently allocated and free servers. Whenever the load detector signaled an overload condition, the allocation scheme added the requisite number of servers from the free server list to the load balancer's list of available servers. Similarly, in an underloaded condition, deallocation was performed by removing servers from the load balancer's available server list and adding them back to the free list. To emulate the allocation overhead parameter Δ, we introduced a wait time before adding or removing a server from the available list of the load balancer.

For balancing the load on the Web servers, we used a least-loaded load balancing scheme, where load was defined as the number of connections open to each Web server. We used such a load balancing scheme as opposed to locality-aware schemes such as LARD [23], because our workload consisted predominantly of dynamic requests and was thus less sensitive to caching effects. In addition to distributing incoming sessions equally among the back-end servers, the least-loaded scheme has the additional advantage of quickly diverting load to a newly allocated Web server, thus bringing it online fast. We show experimentally that this indeed was the case and the load balancer did not affect the reallocation results adversely (for instance, by causing hot spots among the allocated servers).

3.3 Hosted Application and Workload generator

Having described the components of the data center prototype, we now describe the application we hosted on the data center and the workload generator we used on the clients to generate the Web workload.

As our hosted application, we used *RUBiS* [2] — an open-source Web application benchmark that emulates an auctioning website like Ebay [11]. RUBiS is a multi-tiered application that uses a Web server, an application server and a database server tier (as described in Section 3.1), and hence, is fairly representative of common Internet applications. We used the PHP version of RUBiS that implicitly combines the Web server and the application server physically on the same machine. RUBiS has the notion of user sessions, where each user session emulates the actions of a user accessing an auctioning website. These actions include accessing the home page, registering as a user, placing a bid on an item, selling an item, etc. Each session consists of a sequence of requests separated by user think-times. These requests are a combination of static as well as dynamic requests that are serviced using the RUBiS application data maintained on a back-end database.

In our experiments, we first generated offline traces of RUBiS user sessions. The user think-times were generated using a negative exponential distribution with a mean of 2 seconds. This think-time distribution is specified in the TPC-W specifications [29] (We employed this distribution as TPC-W is a popular e-commerce benchmark specification and emulates closely the user behavior for e-commerce applications).

Finally, we replayed these traces using *httperf* [21] — an open-source Web workload generator — running on the client machines, to generate the workload during the experiments. Httperf allows varying various workload parameters such as the rate of session generation, number of requests per session, the request timeout values, etc. With httperf, we were able to generate different kinds of flash crowd conditions using the RUBiS session traces.

4 Experimental Study

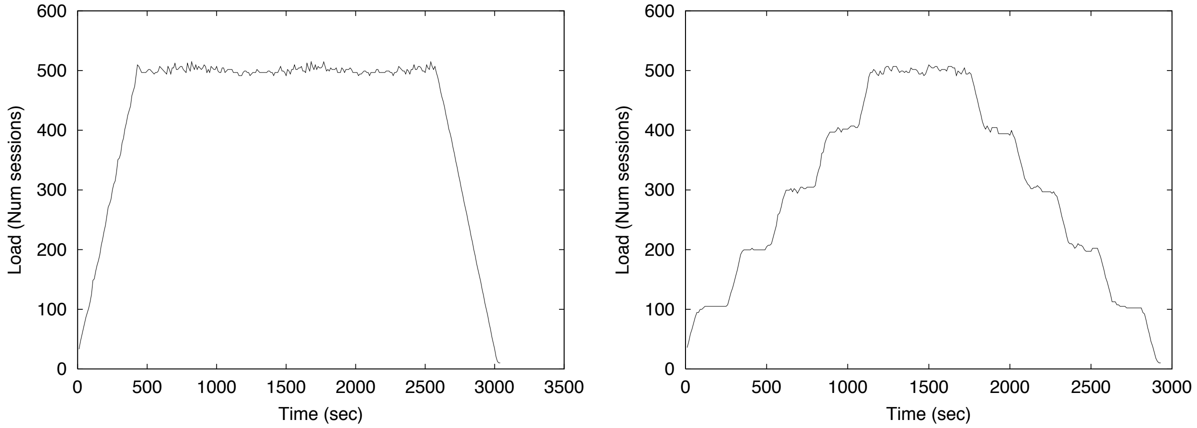
In this section, we present the results of our experimental study to measure the effectiveness of various dynamic allocation parameters on flash crowds with varying characteristics. We begin by describing our experimental methodology, followed by the results from our study.

4.1 Experimental Methodology

As described in the previous section, we use a set of client machines running httperf to generate the workload for RUBiS application. This application is hosted on a multi-tiered data center which employs online monitoring, load detection and reallocation mechanisms.

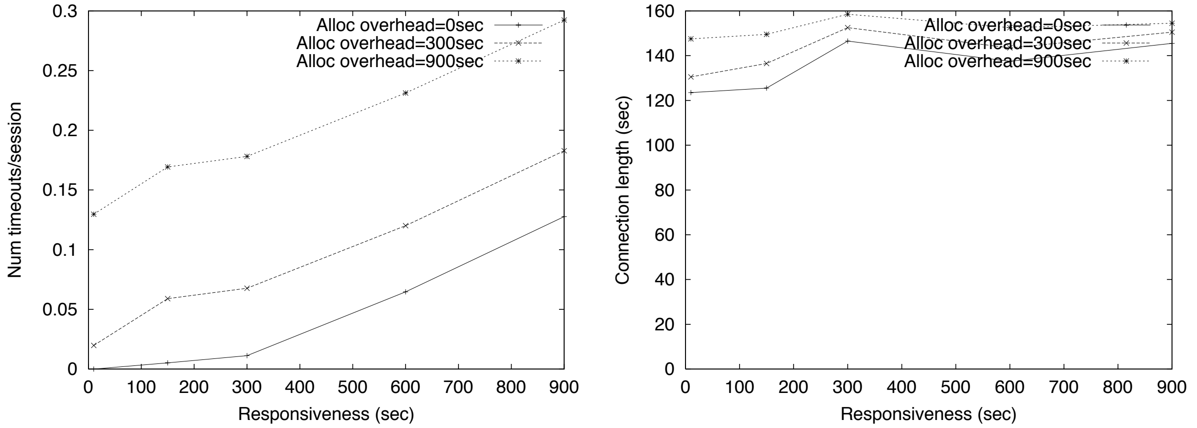
In our experiments, we vary the flash crowd characteristics by controlling httperf parameters such as the total number of user sessions and the rate of session generation. Since we are investigating the effect of dynamic allocation on application performance, we study only a single hosted application, but the same scenario could be recreated for multiple applications. The most important flash crowd characteristic for studying the effect on a single application is the load growth rate (λ) described in Section 2.3. This parameter determines the effectiveness of allocation parameters such as responsiveness and allocation overhead. We use λ values of 20 and 60 sessions/minute in our experiments to measure the effect on these allocation parameters. On the other hand, the peak load P only plays a role in determining the maximum number of servers that need to be allocated, and the duration T is important in determining the amount of overlap with other co-hosted applications in the data center. In our experiments, we use a peak load value of 500 sessions that corresponds to an ideal requirement of 4 servers in our data center implementation. Finally, we keep the total time T of the flash crowd fixed at one hour.

To study the effectiveness of different allocation schemes, we vary the parameters of responsiveness (R) and allocation overhead (A). for each flash crowd, as described in the previous section. The values of responsiveness were varied between 10 seconds (representing a highly responsive allocation scheme) and 15 minutes (representing a slowly responding scheme). Similarly, the values of allocation overhead were varied between 0 and 15 minutes, where a value



(a) Flash crowd with λ=60 sessions/min (b) Flash crowd with λ=20 sessions/min

Figure 4: Flash crowds generated with varying rates of workload increase.



(a) Number of timeouts/session (b) Connection length (95th percentile)

Figure 5: Application performance with varying responsiveness and allocation overhead, flash crowd λ=60 sessions/min

of 0 represents a hot spare allocation, a value of 1-5 minutes represents allocation overhead of a clean spare, while that of 5-15 minutes was used for a dirty spare allocation.

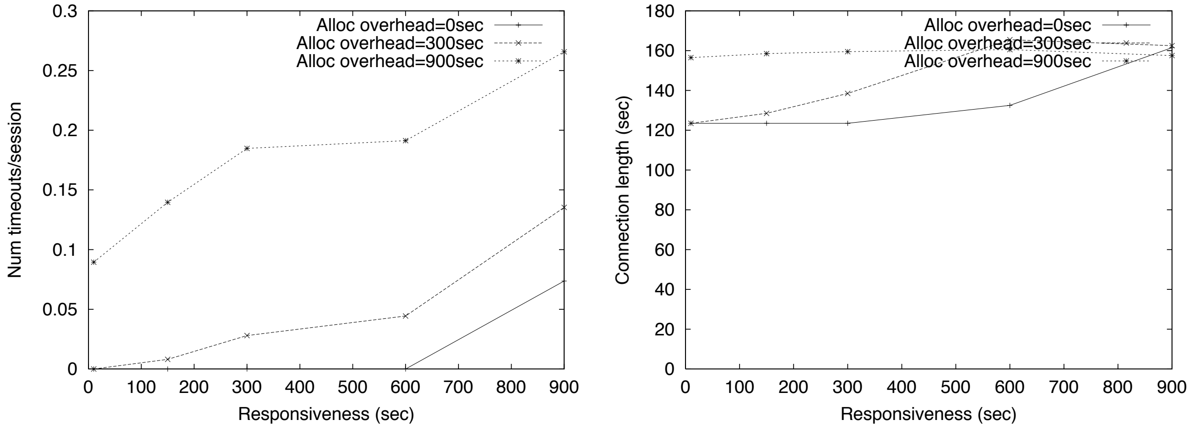
Using these varying parameter values, we then measure various application performance metrics in our experiments. The performance metrics we use are session lengths, reply rates and number of connection timeouts. While the number of connection timeouts and reply rates represent the performance of the application in terms of admitting clients, session lengths measure the performance in terms of the service received by the admitted clients. We use these metrics to measure the application performance under different allocation schemes.

4.2 Experimental Results

4.2.1 Effect of responsiveness and allocation overhead

Figures 4 (a) and (b) show the workload of two flash crowds with mean peak growth rates of 60 and 20 sessions/min respectively. Each of these flash crowds has a total duration of 1 hour, and a peak load of 500 sessions.

In Figure 5, we plot the effect of varying the allocation scheme's responsiveness (R) and allocation overhead (A) on



(a) Number of timeouts/session

(b) Connection length (95th percentile)

Figure 6: Application performance with varying responsiveness and allocation overhead, flash crowd λ=20 sessions/min

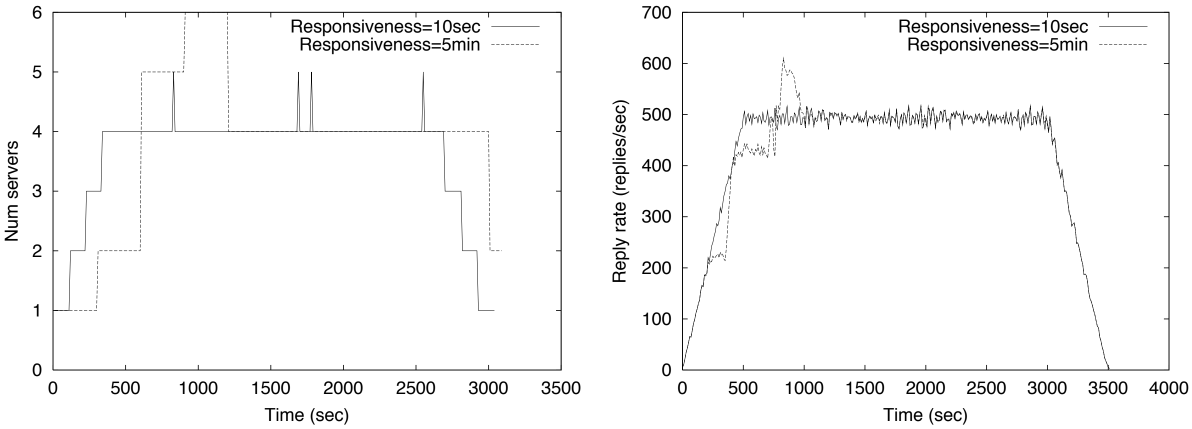
the application performance metrics. Figure 5 (a) shows the number of session timeouts per session as we vary responsiveness on the x-axis. Different curves on the graph correspond to different values of allocation overhead. The value of session timeouts corresponds to the number of user sessions *rejected* by the application. This figure shows that as the value of responsiveness increases from 0 to 15 minutes, the timeout rate steadily increases by about 0.1-0.15 timeouts/session for each value of allocation overhead. Figure 5 (b) plots another application performance metric, namely, the 95th-percentile value of session length. The session lengths are computed only over the *successful* connections, hence this metric corresponds to the performance of the serviced requests. As the plot shows, the 95th-percentile value of session length remains nearly steady between 120 and 160 seconds (with some fluctuations), and there is only a small relative increase in its value as we increase the value of either responsiveness or allocation overhead. These results together imply that *while the admission rate of user sessions decreases as we make the allocation scheme less responsive or increase the overhead of allocation mechanisms, the performance of the successfully serviced sessions is largely unaffected.*

Figure 6 shows similar results for the flash crowd with λ=20 sessions/minute. However, the values of timeout rates are smaller for the corresponding values of responsiveness and allocation overhead in this case compared to those for the flash crowd with λ=60 sessions/min. In fact, as can be seen from Figure 6 (a), the number of timeouts is 0 for responsiveness values as large as 10 minutes when allocation overhead=0, and similarly for allocation overhead values as large as 5 minutes for responsiveness=10 sec. This implies that *with a more slowly growing flash crowd, an allocation scheme that is not very responsive or uses large overhead mechanisms can also perform well.* However, we still need to ensure that at least one of the two parameters of responsiveness and allocation overhead is small.

4.2.2 Variation in application performance with time

To understand our results better, we also look at the variation in the application performance with time for given combinations of allocation parameters. Figures 7(a) and (b) plot time series showing the data center server allocation and the application performance for different values of responsiveness. This figure shows the results for the flash crowd with λ=60 requests/min. We keep the allocation overhead value fixed at 0 (that corresponds to a hot spare allocation), and compare the allocation schemes with responsiveness values of 10 seconds and 5 minutes.

Figure 7(a) shows the number of servers allocated for each responsiveness value, while Figure 7(b) shows the reply rates for the corresponding responsiveness values at corresponding times. As can be seen from the figures, the more responsive scheme (responsiveness=10 sec) is able to allocate the requisite number of servers quickly, and is able to match the workload requirements. On the other hand, the slower responding scheme (responsiveness=5 min) lags in



(a) Number of servers allocated

(b) Client reply rate

Figure 7: Time series of server allocation and application performance with different allocation schemes

the allocation of servers. For instance, at time 500 seconds, while the faster scheme is able to allocate 4 servers, the slower scheme is able to allocate only 2 servers. As a result, the slower scheme shows a degradation in the application performance during the transient stage of the flash crowd. For instance, the slower scheme is able to service only 400 requests/second between time 500 and 700 seconds as compared to 500 requests/second for the faster scheme during the same period.

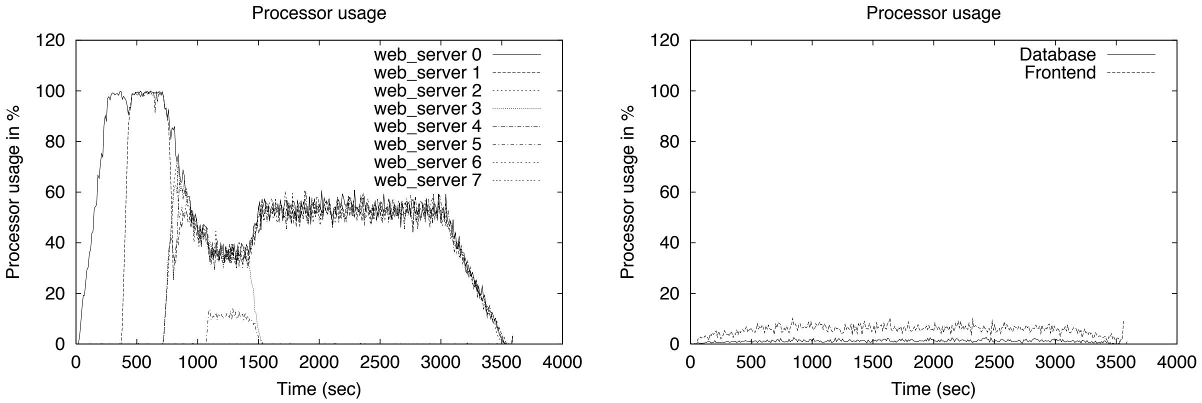
Another interesting region in the figures is the time period between 750 and 1250 seconds. Here, the less responsive scheme ends up over-allocating resources by 2 servers and its reply rate also increases upto 600 requests/sec briefly. This happens due to the user session backlog that accumulates in the transient period. This illustrates that the less responsive scheme is also more unstable than the faster responding scheme in terms of the application performance. Finally, both schemes perform equally well once the fluctuations die down and the flash crowd settles down in a steady state.

These results imply that *faster responding and low overhead allocation schemes can respond better to the transient periods* of flash crowd load increases as compared to slower schemes. Moreover, *the less responsive schemes are less stable* in application performance due to the backlog created during the transient period resulting in *temporary overallocation of resources*.

4.2.3 Variation in server load with time

Having looked at the application performance as the workload and the server allocation change with time, we now examine the load on the servers as a function of time. In Figure 8, we plot the load variation on the various data center servers for the flash crowd with λ=60 sessions/minute. The responsiveness and allocation overhead of the allocation scheme were each set to 5 minutes in this experiment (corresponding to the dotted curve in Figures 7(a) and (b)). Figure 8(a) plots the CPU usage of Web servers in the system as they are allocated one-by-one with the load variation, while Figure 8(b) plots the CPU usage on the load balancer and the database server.

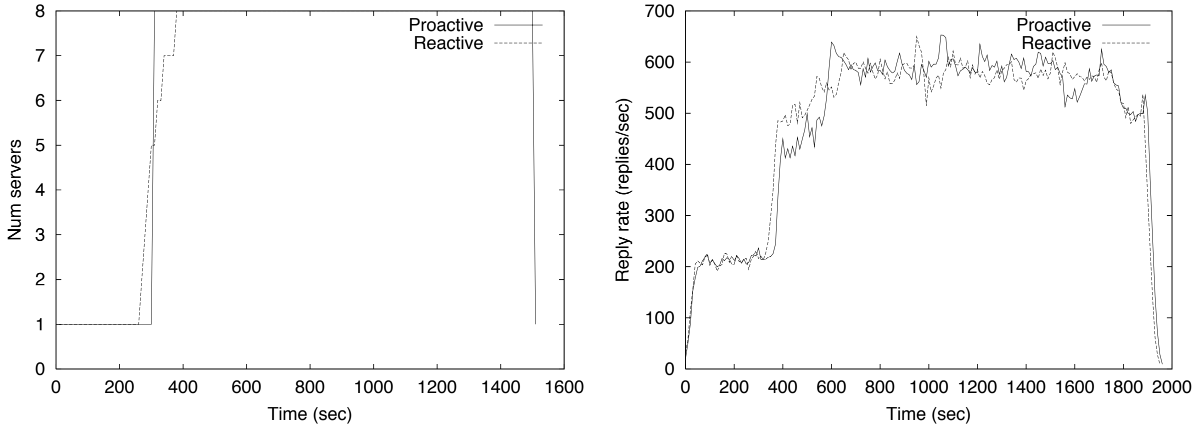
As can be seen from Figure 8(a), the load on the initially allocated Web servers grows to almost 100% by time 750 seconds. This is due to the lag in allocation of new servers. However, 3 new servers are started at time 700 seconds, which quickly brings down the load on the individual servers to about 40%, and finally, in the steady state, all servers maintain a load of about 50%. Note that 2 servers were deallocated at time about 1500 seconds, as the servers had become underloaded. The interesting point to be noted from this figure is the effectiveness of the load balancing. Recall from Section 3.2.2 that we used a least-loaded load balancing strategy. This figure confirms the effectiveness of this strategy in the presence of dynamic workload when caching effects are not very important. Also, Figure 8(b) shows that the load on the load balancer and the database server are negligible. This confirms our assertion



(a) Load on the Web servers

(b) Load on the load balancer and database servers

Figure 8: Variation in the load on various data center servers with change in allocation



(a) Number of servers allocated

(b) Client reply rate

Figure 9: Comparison of a slow proactive and a fast reactive allocation scheme in the presence of high allocation overhead.

in Section 3.1 that Web server tier is the bottleneck tier for our experimental workload.

4.2.4 Comparison of Slow Pro-active and Fast Reactive Allocation

Another aspect of resource allocation we consider is the ability to allocate servers pro-actively as against allocating them reactively. In the case of a proactive scheme, there is an early allocation of servers in anticipation of load increase. This could be based on some kind of short-term prediction using recently observed workload metrics. On the other hand, a reactive scheme would allocate resources solely on the basis of current load on the system. The advantage of using a pro-active allocation scheme would be most pronounced when the allocation overhead of new servers is high. In that case, multiple servers can be pre-allocated and would be expected to be ready to handle the load by the time it increases to higher levels. Here, we conduct an experiment to study the conditions under which a reactive scheme might be able to emulate the performance of a pro-active scheme.

To study this aspect, we generated a flash crowd with peak growth rate of 120 sessions/minute and peak load of about 1800 sessions. The duration of the flash crowd was 30 minutes. These parameters emulated a fast-growing flash crowd requiring large number of servers to handle the peak load.

In Figure 9(a) and (b), we plot the server allocation and the application performance using two allocation schemes. One is a pro-active allocation scheme with a responsiveness of 5 minutes. It accurately predicts the expected load during this time period and allocates enough servers to handle this load. On the other hand, the second scheme allocates individual servers as the load increases based on the current load. This scheme is more responsive, with a responsiveness of 10 seconds. In this experiment, the allocation overhead is 5 minutes for each scheme, so that the servers take 5 minutes to get ready. As shown in Figure 9(a), the server allocation of the pro-active scheme is more aggressive in that it is able to allocate multiple servers at the same time. On the other hand, the reactive scheme reacts in a more incremental manner allocating 1 server at a time. Despite this difference, their server allocations match closely, and their client reply rates also match closely with a few fluctuations, as shown in Figure 9(b).

These results imply that in case of rapidly increasing loads, a highly responsive reactive scheme can perform as well as a less responsive pro-active scheme, so that it is possible to meet similar application performance by *either allocating multiple servers simultaneously or a few servers frequently.*

4.3 Implications of our results

In this section, to summarize the experimental results we presented in Section 4.2. Our results indicate that

1. Using a more responsive allocation scheme, or using less overhead allocation mechanisms such as using hot spares or clean spares is preferable.

2. Slower allocation mechanisms and schemes are susceptible to instability in application performance and can result in temporary over-allocation.

3. Even with slower mechanisms, the application performance is degraded only during the transient period of flash crowd load growth, and in steady state, the performance is comparable to that of faster mechanisms.

4. A highly responsive reactive allocation scheme can match the performance of a less responsive pro-active scheme in the presence of fast-increasing workloads.

5. A simple least-loaded load balancing strategy can work well when the workload is primarily dynamic and caching effects are not predominant.

The above results (particularly result 3) indicate that it is possible to use slower mechanisms such as dirty spares in flash crowd conditions, if the scheme can maintain a few hot spares to service the load during the transient period of server preparation. This is an important observation, because in many flash crowd conditions, the allocation scheme might have to fall back on clean and dirty spares after a while, if the flash crowd is large in magnitude or duration. Under such conditions, it is possible to pro-actively start allocating the slower spares, and press the more readily available servers into service for the intervening duration. Further, result 4 implies that for a purely reactive scheme, it is important to be highly responsive, in which case it is able to emulate a less responsive pro-active scheme. Moreover, what this result indicates is that even with large allocation overhead, it is possible to meet application requirements by either allocating multiple servers simultaneously (as done by a slow pro-active scheme) or by allocating servers often (as done by a fast reactive scheme).

5 Related Work

A dynamic allocation technique to handle unexpected workload surges has been proposed in a recent work [17]. This technique performs dynamic allocation using short-term load prediction coupled with early pre-allocation of resources. Our work differs from this work in the following aspects. First of all, we examine not just a single allocation technique, but a range of allocation parameters to understand their utility under different scenarios. Also, this technique assumes fixed server allocation overheads of about 30 seconds, while we consider a range of values ranging from very small (0) to large (15 minutes) for the allocation overhead that correspond to servers being in different states at the time of allocation. This is an important parameter, as flash crowds, by their very definition, can have large magnitudes necessitating the need to allocate servers with high allocation overheads.

There have been several proposals for managing overload in Web workloads. Many of these approaches employ different forms of admission control [9, 13, 15, 31], traffic shaping [14, 30] and scheduling [28]. These approaches use the techniques of workload shaping and service differentiation to meet the application performance needs. Our approach of using dynamic allocation under overload scenarios is complementary to these approaches, and can be used in conjunction with them to achieve good application performance while admitting large number of requests. For instance, techniques like admission control or service differentiation could be employed during the transient allocation period, and they can be relaxed once the requisite number of servers become available for service.

Recent studies [3, 7] have examined the effect of allocation parameters on resource provisioning gains in data center environments. These studies are different from our work in several respects. First of all, our work focuses on the impact of dynamic allocation on *application performance* as opposed to resource savings that was the main focus of these studies. Secondly, these studies obtain their results by performing data analysis of real data center traces. On the other hand, we use an experimental methodology, wherein we employ a prototype data center implementation with a real Internet application benchmark to conduct an experimental study. Such a methodology provides us with the ability to monitor server resources and application performance under varying allocation parameters, while also allowing us to systematically vary the workload parameters.

In this paper, we focused on dedicated data center architectures [5, 24]. However, several other data center architectures have been proposed. These include shared architectures such as MUSE [8] that can host multiple applications on shared servers, and hybrid architectures like Cluster-on-Demand [20] that hierarchically allocate virtual clusters to a group of applications. Another architecture makes use of virtual clusters to allocate resources across globally distributed data centers [25]. Some of these architectures like the hybrid one share some aspects of dedicated architectures (for instance, with respect to allocating servers across virtual clusters), and hence, some results of our study are directly applicable to them. Shared architectures allow very fast re-provisioning of resources withing servers and thus display a high degree of responsiveness and low allocation overhead. However, other aspects of allocation in such architectures, such as work-conserving nature of server resource schedulers, server capacity, etc., need to be examined in greater detail.

Several research efforts have also proposed prediction and workload characterization techniques that work well for long-term seasonal trends such as time-of-day-effects [12, 26, 32]. However, these techniques are not directly applicable for flash crowd scenarios, as flash crowds are inherently unpredictable, and these techniques also require access to a large data record to make their predictions.

Several dynamic resource allocation techniques have been proposed for data centers that use modeling techniques to achieve resource guarantees [1, 6, 10, 19]. These techniques use measurement and prediction techniques to reallocate resources among applications based on their varying workload. Such techniques are complementary to our work and can be employed on a data center implemented with appropriate allocation mechanisms.

6 Conclusions

In this paper, we presented a prototype data center implementation that we uses to study the effectiveness of dynamic resource allocation for handling flash crowds with different characteristics. This prototype implements a multi-tiered server architecture along with mechanisms for monitoring, load detection, load balancing and dynamic allocation. Our experiments with this prototype showed that a carefully designed dynamic allocation scheme can be effective for handling flash crowds. We showed that in order to handle very sharp growth in loads, a dynamic allocation scheme must be either extremely responsive or employ low overhead mechanisms such as using hot spare servers. On the other hand, gradually increasing flash crowds can be handled equally well with larger overheads and slower reaction times. We also showed that even in the presence of large allocation overhead, it is possible to achieve the same application performance by either allocating multiple servers simultaneously or allocating a few servers often. Using our results, we conclude that even without large-scale over-provisioning, it is possible to effectively handle flash crowd conditions using a dynamic allocation scheme that responds quickly to workload changes, and that can mask large allocation overheads either by deploying a few ready servers or by allocating multiple servers simultaneously.

As part of future work, we would like to examine application performance in a multiple application setting. In such a scenario, it would be interesting to investigate the effect that the time duration and peak magnitude of flash crowds have on the amount of correlation between multiple application demands. This investigation would also shed a light on how successfully a dynamic allocation scheme can be in preparing servers for re-allocation when the need arises. In addition, we would like to examine more carefully the effects of reallocation algorithms and prediction techniques on the various allocation parameters and the effectiveness of dynamic allocation under flash crowd conditions.

References

[1] T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), January 2002.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *Proceedings of WWC-5*, November 2002.

[3] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the Resource Savings of Utility Computing Models. Technical Report HPL-2002-339, HP Labs, December 2002.

[4] Apache HTTP Server Project. http://httpd.apache.org.

[5] K. Appleby, S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based Management of a Computing Utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May 2001.

[6] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of Eleventh International Workshop on Quality of Service (IWQoS 2003)*, June 2003.

[7] A. Chandra, P. Goyal, and P. Shenoy. Quantifying the Benefits of Resource Multiplexing in On-Demand Data Centers. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[8] 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.

[9] L. Cherkasova and P. Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685, June 2002.

[10] R. Doyle, J. Chase, O. Asad, W. Jin, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of USITS'03*, March 2003.

[11] Ebay. http://www.ebay.com.

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

[13] R. Iyer, V. Tewari, and K. Kant. Overload Control Mechanisms for Web Servers. In *Workshop on Performance and QoS of Next Generation Networks*, November 2000.

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

[15] V. Kanodia and E. Knightly. Multi-class latency-bounded web servers. In *Proceedings of International Workshop on Quality of Service (IWQoS'00)*, June 2000.

[16] Kernel TCP Virtual Server. http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html.

[17] E. Lassettre, D. Coleman, Y. Diao, S. Froelich, J. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that have Lead Times. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[18] W. LeFebvre. CNN.com: Facing a World Crisis, June 2002. Invited Talk, USENIX Annual Technical Conference.

[19] Z. Liu, M. Squillante, and J. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, 2001.

[20] J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase. Managing Mixed-Use Clusters with Cluster-on-Demand. Technical report, Department of Computer Science, Duke University, November 2002.

[21] David Mosberger and Tai Jin. httperf – A Tool for Measuring Web Server Performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[22] MySQL. http://www.mysql.com.

[23] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, , and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[24] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[25] J. Rolia, S. Singhal, and R. Friedrich. Adaptive internet data centers. In *Proceedings of SSGRR 2000*, July 2000.

[26] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical Service Assurances for Applications in Utility Grid Environments. Technical Report HPL-2002-155, HP Labs, December 2002.

[27] Sysstat package. http://freshmeat.net/projects/sysstat.

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

[29] W. Smith. TPC-W: Benchmarking An Ecommerce Solution. http://www.tpc.org/information/other/techarticles.asp.

[30] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differrentiation in overloaded web servers. In *Proceedings of USENIX Annual Technical Conference*, June 2001.

[31] 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.

[32] F. Zhang and J. L. Hellerstein. An approach to on-line predictive detection. In *Proceedings of MASCOTS 2000*, August 2000.