SHRiNKing Web Server Farms: A Method for Scaleable Performance Prediction and Measurement

Konstantinos Psounis
Departments of Electrical Engineering and Computer Science
University of Southern California
email: kpsounis@usc.edu
phone: 001-213-7404453
address: 3740 McClintock Ave., EEB 304, Los Angeles, CA 90089

Abstract

The increasing size of web-server farms and the sheer volume of HTTP requests, makes hard to collect performance measurements and monitor the state of a farm in real time. Further, it increases the cost of a bad algorithmic or architectural decision, while predicting the performance of new algorithms and architectures is also hard.

We propose a way to side-step these problems, by intelligently combining small-scale experiments and analysis. Our hypothesis is this: if we take a sample of the incoming requests, and feed it into a suitably scaled version of the web farm, we can extrapolate from the performance of the scaled system to that of the original.

Our main finding is that when we suitably scale a web-server farm, then performance measures such as mean response time and throughput are left virtually unchanged. We show this using experiments and simple analysis.

Keywords: Performance prediction, Web Farms.

1 Introduction

Monitoring the performance of a large system like a web-server farm, and predicting its behavior under novel algorithms, architectures and load conditions are important research problems. Operators find it useful to look at a live representation of their entire farm: this lets them identify potential problems at a glance, ensure reliable operation, and monitor their service level agreements. Researchers like to evaluate the effect of design changes on performance. As a result, these problems have inspired research [2, 4] and industrial products [10].

Large web-server farms consist of thousands of servers and may handle millions of HTTP requests per second. While operators are just about able to collect detailed traffic statistics, the very detail and volume make them nearly impossible to analyze. As a result, performance prediction, monitoring, and performance measurement are rendered increasingly complicated. To keep the size of web traces manageable, researchers record traces corresponding only to a couple of minutes real time. Hence, trace-driven simulations cannot accurately predict the performance of novel algorithms.

To solve these problems, in this paper we explore a new method which combines sampling, small-scale experiments, and simple analysis. Our basic hypothesis, which we call SHRiNK[1], is this: if we take a *sample* of the traffic, and feed it into a *suitably scaled* version of the system, we can *extrapolate* from the performance of the scaled system to that of the original. Note that SHRiNK may apply to other large-scale systems as well, e.g. to IP networks [8, 9].

This has two benefits. First, by relying only on a sample of the traffic, SHRiNK reduces the amount of data we need to work with. Second, by using samples of actual traffic, it short-cuts the traffic characterization and model-building process while ensuring the relevance of the results.

This approach also presents challenges. At first glance, it appears optimistic: might not the behavior of a large farm with many servers be intrinsically different to that of a smaller one? Somewhat surprisingly we find that one can mimic a large farm using a suitably scaled-down version. The key is to find suitable ways to sample the traffic, scale down the farm, and extrapolate performance. In particular, we find that when we suitably scale a web-server farm, then performance measures such as mean response time and total throughput are left virtually unchanged.

[1]SHRiNK: Small-scale Hi-fidelity Reproduction of Network Kinetics

Practitioners sometimes use such scaling techniques in an ad-hoc manner when they want to test a novel architecture or new software in a large farm. However, to the best of our knowledge there is no systematic study of the scaling properties of todays web-servers and web-server farms. We attempt to do such a study, give guidelines on how to scale a farm, and explain the reasons why scaling works using simple analysis.

2 Scaling Web Server Farms

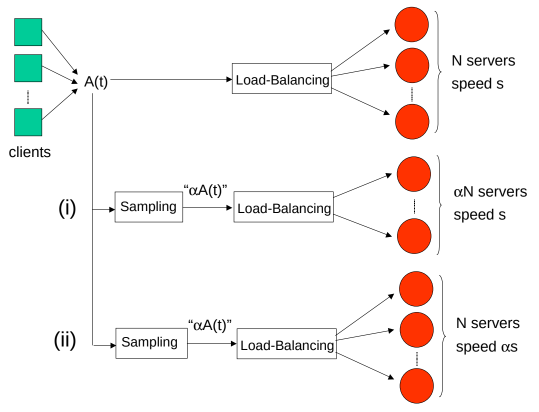

Figure 1: Scaling a web server farm:[2] (i) scaling the number of servers, (ii) scaling the speed of the servers.

Consider a web server farm with N servers each having speed s, as in Figure 1. Sample in an i.i.d. fashion with probability α the requests destined for the original farm. (We elaborate more on sampling in the following sections, where for every experiment we discuss whether sampling takes place at the session, document, or embedded request level. A web browsing session is a series of document requests initiated by a user, with small silent periods between consecutive requests. Each document request usually initiates a bunch of embedded requests. These requests are usually for images embedded in the document.)

Feed the sampled traffic into a farm consisting of either (i) a fraction α of the original web servers, or (ii) the same number of servers each having speed αs (see

[2]This is a simplified picture of a farm, since the application-servers, the databases, and the switches used to interconnect the various components are absent.

(i) and (ii) of Figure 1). We wish to investigate how close the average response time, the server throughput, and the capacity (maximum throughput) in the scaled system are to those in the original system.

3 Scaling the Speed of Servers

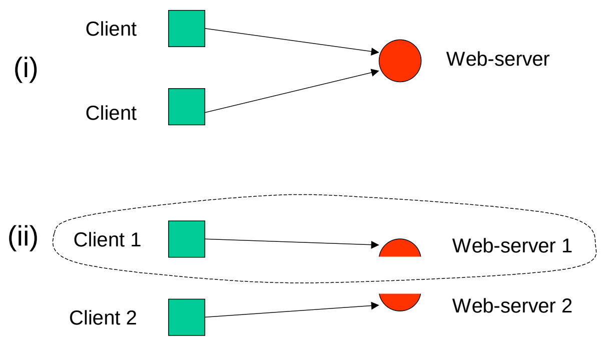

Figure 2: Experimental setup: (i) the unscaled system, and (ii) effectively running a server at a fraction $\alpha = .5$ the speed, by running two servers on a single machine.

In practice, it is not straightforward to simply run a server at a slower speed. Instead, we use the setup shown in Figure 2: Three Linux machines, configured with a Pentium II at 300MHz and 256MB of RAM, are connected to a 100Mbits/sec switch. A variable number of Apache 1.3.9 [1] web servers is hosted at one of the machines, and the other two machines run Surge [3] HTTP traffic generators. Running two web servers on a single machine has the effect of halving the speed of each: when the load is light, speed is not a constraint either for the two web servers or for one at half the speed; when load is heavy, each of the two servers runs most of the time with its maximum number of concurrent httpd processes, so they share the speed of the machine between them equally.

We compare the average response time of the following two systems:

I. Original system, illustrated in Figure 2(i): The hosting machine hosts a single web server, which serves requests from both client machines.

II. Scaled system, illustrated in Figure 2(ii): The hosting machine hosts two web servers, each listening on a different IP address. Each of the client machines generates requests destined for only one of the servers. One client-server pair is used only to consume half of the resources while the other pair is used to record the results for the scaled system.

The web files of both systems are generated using Surge, configured with the default parameters. Surge generates web traffic by creating a number of *userequivalents*, each of which generates HTTP requests in the following manner: First, an HTTP object is requested. Objects consist of a set of embedded requests and model a user's "click", i.e. a request for a web document. Then, the user equivalent waits till it receives the responses for all the embedded requests. After that, it sleeps for some random time. Finally, a new object is requested and so on and so forth. [3] Notice that in the scaled system described above, we essentially sample user equivalents.

We have performed experiments with HTTP/1.0, HTTP/1.1 without pipelining, and HTTP/1.1 with pipelining. The results are similar, so we only show results for the latter case. Further, we show results for the cases where the bottleneck is the maximum number of concurrent httpd processes, or the server CPU. Unless otherwise stated, the Apache servers use the default configuration parameters.

3.1 Experimental Results

Apache's default limit on the number of concurrent httpd processes is 150. In the scaled system, we set a limit of 75 for each of the two servers. Figure 3 plots the mean response time of HTTP objects as a function of the normalized load, that is, the load multiplied by α^{-1}. The load is measured in user equivalents. Figure 4 is the same plot zoomed in. We see that scaling the system leaves the mean response time virtually unchanged. (So, for example, the mean response time for the original system serving 200 user equivalents is the same as that for the scaled system serving 100 user equivalents.) Figure 5 shows the normalized server throughput and capacity, that is, the server throughput and capacity multiplied

[3]User equivalents are like web sessions with an important difference: they are all created at the beginning of the experiment and terminate at the end, instead of arriving and terminating at random times.

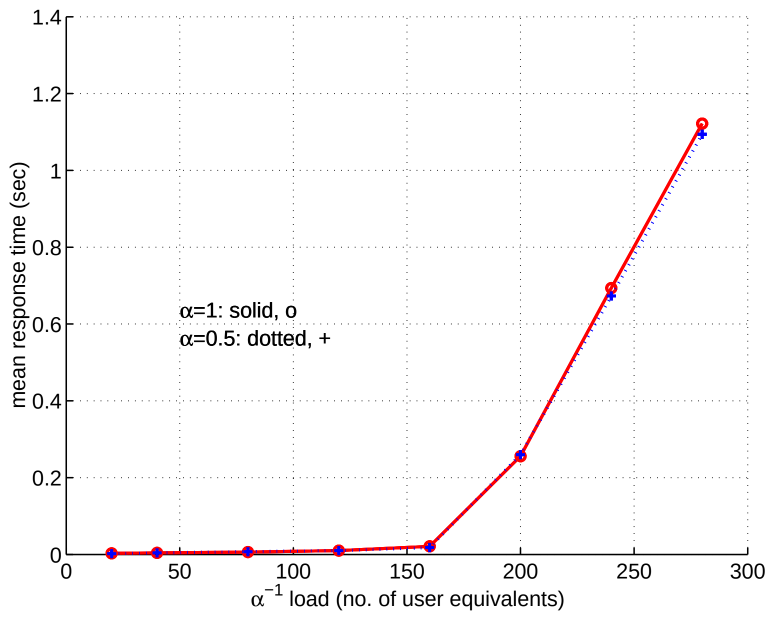

Figure 3: Average response time, when the bottleneck is the number of concurrent processes.

by α^{-1}, as a function of the normalized load. Again, scaling the system changes these quantities by very little.

To make the CPU the bottleneck, we recompile Apache to allow a large number of concurrent httpd processes, and set their maximum number to 800 in the original server and to 400 in each of the scaled servers. Figures 6 and 7 show the average response time and the normalized server throughput as a function of the normalized load. Again, scaling the system leaves these quantities virtually unchanged. Notice that, as expected, the server capacity is increased due to allowing more concurrent connections.

An intuitive way to see why performance scaling occurs in these experiments is the following: Let s be the speed of the server and n be the number of concurrent httpd processes in the original system. Then, since in the scaled system there are two web servers hosted on the same machine that receive half of the traffic each, each server has speed $s/2$, and the number of concurrent httpd processes on each server is $n/2$. Hence, since CPU operates in a processor sharing manner, each job gets the same service rate in the two systems and mean response times remain unchanged.

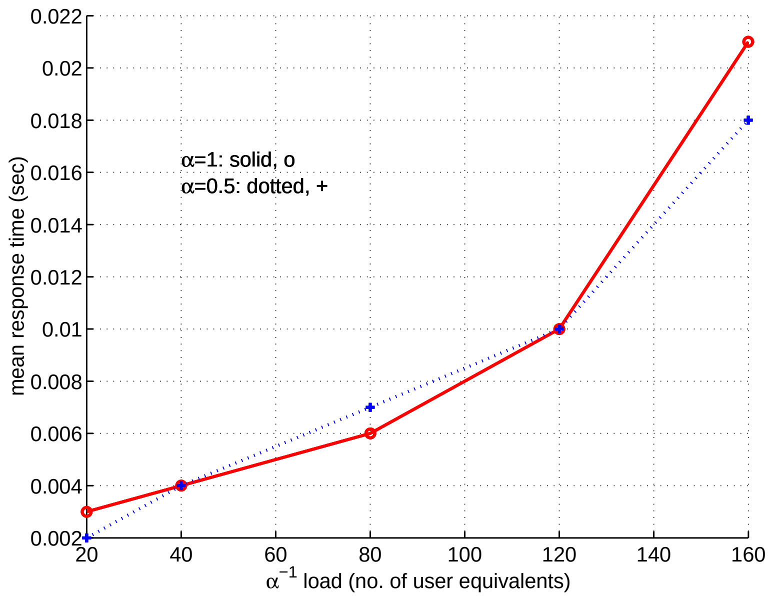

Figure 4: Average response time, when the bottleneck is the number of concurrent processes (zoomed in).

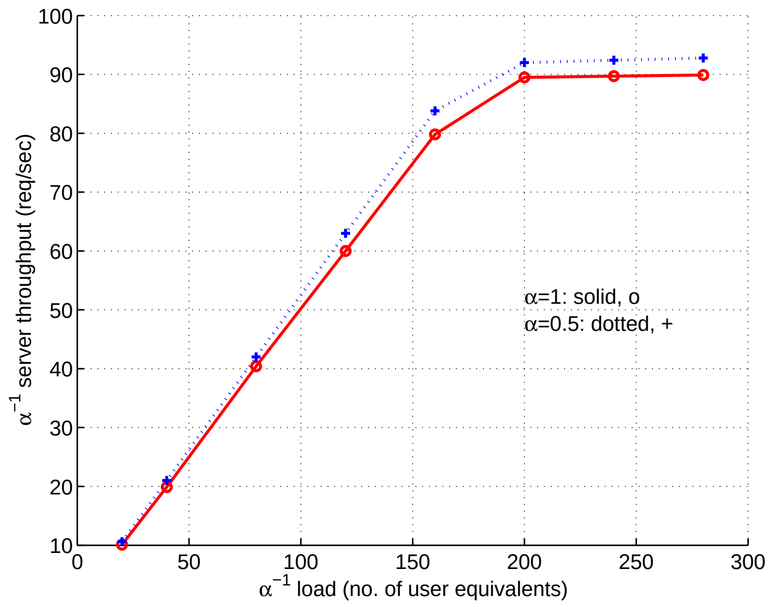

Figure 5: Server throughput, when the bottleneck is the number of concurrent processes.

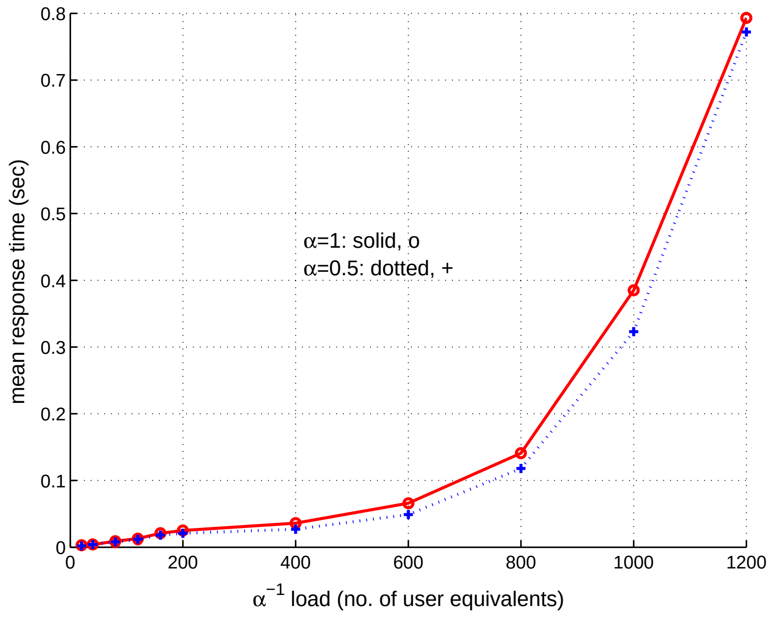

Figure 6: Average response time, when the CPU is the bottleneck.

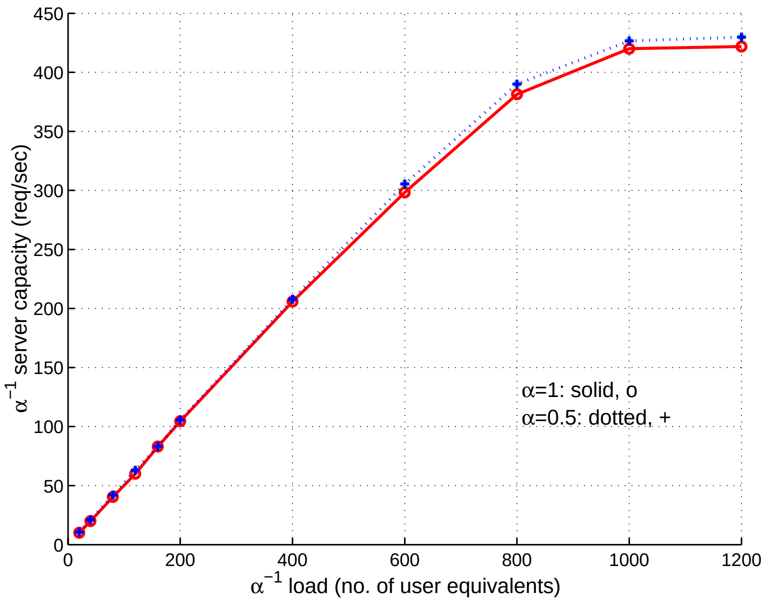

Figure 7: Server throughput, when the CPU is the bottleneck.

4 Scaling the Number of Servers

We investigate the validity of SHRiNK in server farms when one scales the number of servers of the farm.

Consider the farm illustrated in Figure 1. When servers are identical, it is straightforward to scale their number. If there are K types of servers, we simply scale down by the same proportion the servers of each type.

Sampling is more involved. There are two levels at which one may perform the sampling, namely at the object or the user-equivalent level. [4] When HTTP1.1 is used, it is important to sample at the same level at which load balancing takes place. To see this, suppose load balancing takes place at the user-equivalent level, but sampling takes place at the object level (see Figure 1). Then, since HTTP1.1 maintains persistent connections between the clients and the servers, a server will have more concurrent connections in the scaled system than in the original system. (This is because there are more user equivalents per server in the scaled system than in the original.) We perform experiments and verify that SHRiNK is valid when load-balancing and sampling take place at the same level, while when this constraint is not met the mean response time in the two systems differ. This is not an issue when HTTP1.0 is used, since HTTP1.0 does not use persistent connections.

We illustrate the above points by presenting some experimental results. In the experiments we use eight Linux machines configured with a Pentium III at 550MHz and 384MB of RAM, connected to a 100Mbits/sec switch. Machines acting as servers host one Apache 1.3.9 web server, and machines acting as clients run Surge to generate HTTP requests.

4.1 Experimental Results

In the first experiment the clients use HTTP1.1. Load-balancing is a simple round-robin scheme. Both load-balancing and sampling take place at the user-equivalent level. We compare the average response time of the following two systems:

I. Original system, illustrated in Figure 8(i): Four machines act as servers, and

[4]Sampling at the embedded request level would not work well because the sample doesn't capture the structure of the traffic.

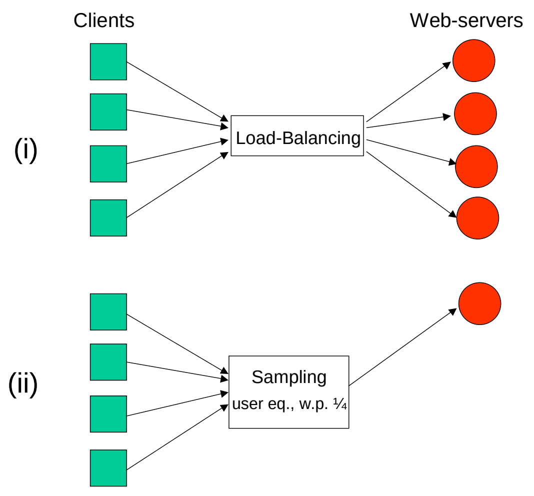

Figure 8: Experimental setup using HTTP1.1: (i) the unscaled system, and (ii) scaling the number of servers and sampling incoming user equivalents, $\alpha = .25$.

four as clients.

II. Scaled system, illustrated in Figure 8(ii): The user equivalents of the four clients are sampled in an i.i.d. fashion with probability 1/4 and are directed to a single server machine.

Figures 9 and 10 show the average response time and the normalized server throughput as a function of the normalized load. Again, scaling the system leaves these quantities virtually unchanged. Note that we treat the farm of the four servers as a single entity. The normalized load is the total normalized load directed into the farm, and the normalized throughput is the sum of the normalized throughputs of the servers of the farm.

Intuitively, performance scaling occurs in this experiment because the total number of user equivalents directed to the stand-alone server of the scaled system is the same with the total number of user equivalents directed to each of the servers of the original farm.

In the second experiment the servers receive HTTP1.0 requests. In practice, this happens either when clients use HTTP1.0., or more frequently due to a proxy that

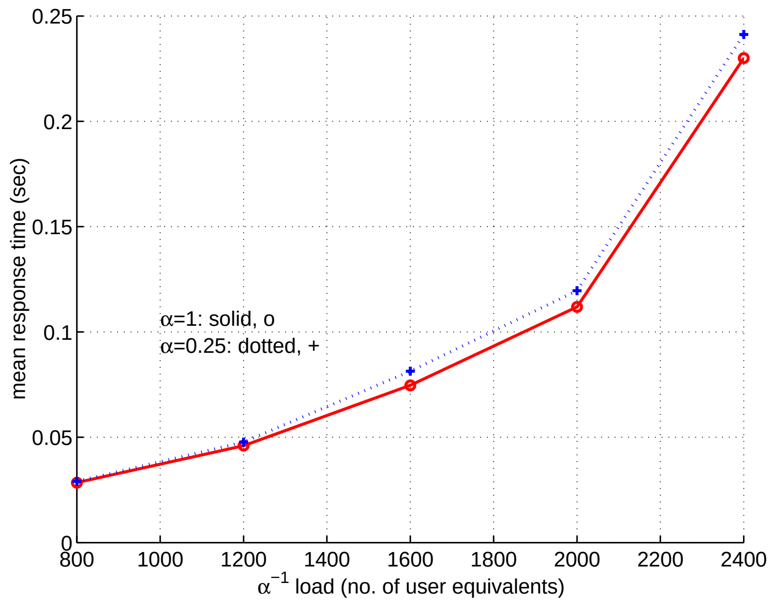

Figure 9: Average response time, when sampling user equivalents.

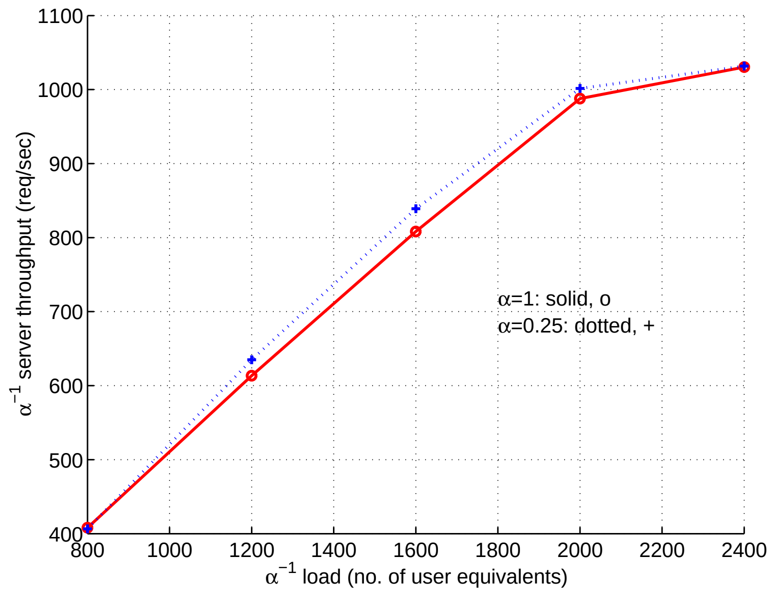

Figure 10: Server throughput, when sampling user equivalents.

may reside at the entry point of a farm and convert HTTP1.1 requests to HTTP1.0 requests.

(The rational of using this proxy is the following: The persistent connections that HTTP1.1 maintain reduce the average response time, when the network delay between the client and the server is large. However, if the client and the server are connected at the same LAN, it is faster to use HTTP1.0 because as the number of clients increases the overhead of persistent connections is large. Hence, using a proxy at the entry point of a farm to convert HTTP1.1 requests to HTTP1.0 requests and visa versa, exploits the advantages of both versions of the HTTP protocol.)

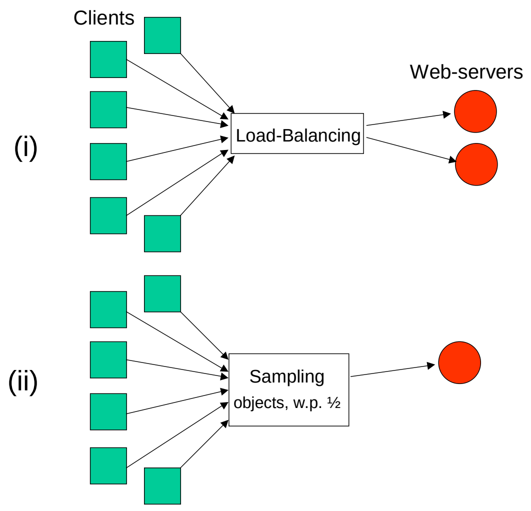

Figure 11: Experimental setup using HTTP1.0: (i) the unscaled system, and (ii) scaling the number of servers and sampling incoming HTTP objects, $\alpha = .5$.

Load-balancing is again a simple round-robin scheme at the user-equivalent level, while sampling takes place at the object level. [5]

We compare the average response time of the following two systems:

I. Original system, illustrated in Figure 11(i): Two machines act as servers,

[5]Due to the close-loop manner by which Surge generates requests, when an object is not sampled, we make the corresponding user equivalent to sleep for some time to account for the object service time and the sleep time that it would wait if the object were requested.

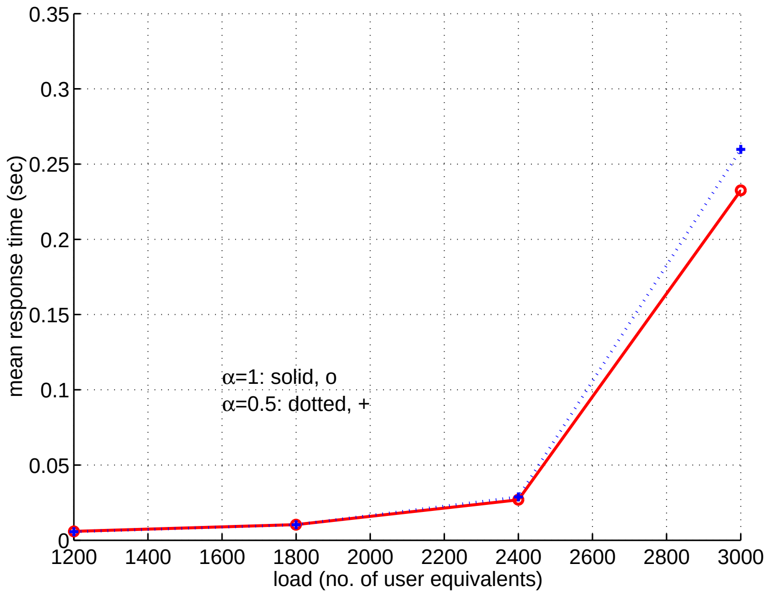

Figure 12: Average response time, when sampling HTTP objects.

and six machines act as clients. (We want to be able to saturate the servers. In the previous experiment four client machines are enough to achieve that, since HTTP1.1 is used and we artificially saturate the farm by setting the maximum number of concurrent connections to 500. Here, since HTTP1.0 is used, we need to saturate the CPU utilization of the server machines and hence we need more client machines.)

II. Scaled system, illustrated in Figure 11(ii): The HTTP objects of the six clients are sampled in an i.i.d. fashion with probability 1/2 and are directed to a single server machine.

Figures 12 and 13 show the average response time and the normalized server throughput as a function of the load. [6] Again, these quantities remain virtually unchanged after scaling. Note that we treat the farm of the two servers as a single entity.

[6]The number of user equivalents sending requests at the two systems is now the same hence the horizontal axis is not multiplied with α^{-1} as before. It is the number of objects and requests directed at the two systems that differ due to object sampling.

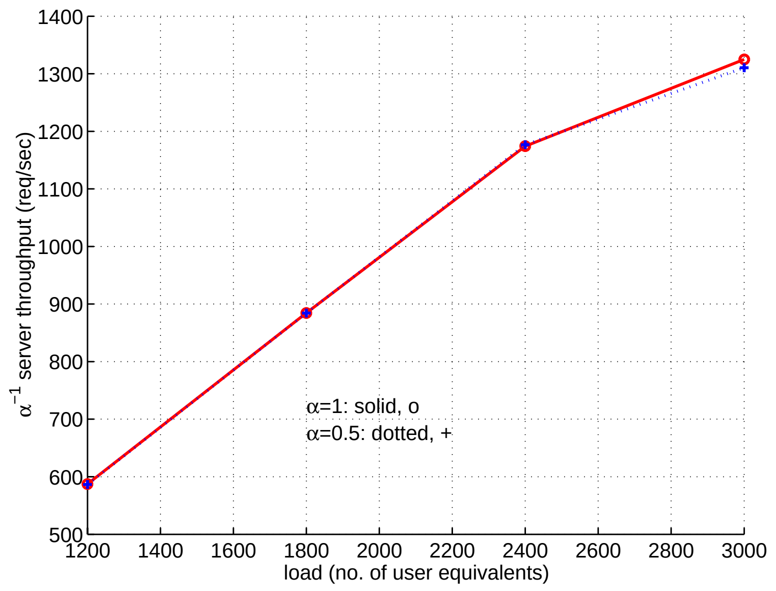

Figure 13: Server throughput, when sampling HTTP objects.

4.2 Simple Analysis

Consider the special case where the servers of the original farm operate independently and one samples web sessions. If the load-balancer assigns sessions to servers uniformly at random then SHRiNK is valid. To see this, note that web sessions are Poisson[5] and let the session arrival rate be λ. Sample a proportion α of the sessions, and keep a proportion α of the N servers in the original web farm. Then the arrival process to each server in the original system is Poisson with rate λ/N, and the arrival process to each server in the scaled system is Poisson with the same rate $\alpha\lambda/\alpha N = \lambda/N$. Hence, mean response times are the same.

Is this analysis relevant to practice? Web servers may not be independent, either by virtue of sharing resources such as databases and bandwidth, or due to load balancing. However, usually databases and bandwidth are not the bottleneck on a farm. Further, the most commonly used load-balancing scheme is round-robin. Hence, the dependencies among web servers are usually insignificant. Another element of real web farms that is missing from our experiments is large network delays and packet losses due to slow or congested connections between the clients and the servers. Recently [7], researchers have shown that these factors may significantly affect server performance and response times. We believe the validity of SHRiNK will not be affected by wide-area conditions, since both the original and scaled web-farms face the same conditions.

The analysis above can be also used to shed light on the experimental results when one samples HTTP objects. First, the round-robin load balancing mechanism that is used does not introduce significant dependencies among the servers. Second, the sampled traffic is close to Poisson. To see this, note that HTTP objects are generated by independent user equivalents. Further, since the sleep time is usually significantly larger than the transfer time, the object interarrival times can be considered more or less independent. Hence, the aggregate object arrival times are close to Poisson, and the analysis goes through.

As a final comment, note that Figure 1 implies that one fast server is equivalent to many slow servers. This is not always the case. For example, if incoming requests are serviced in a FIFO manner, the fast single server can be clogged up by a large job and hence give larger mean transfer times than the many slow servers, especially when service requirements are heavy-tailed [6]. However, web servers service requests in a processor sharing manner and server clogging does not occur.

5 Conclusions and Future Work

In this paper we have presented a method, SHRiNK, to reduce the complexity of web-server farm performance prediction and measurement. Our finding is that when one suitably samples the incoming HTTP requests and appropriately scales the farm resources to create a small-scale replica of the original farm, performance measures such as mean response time, throughput, and the capacity of the farm can be accurately predicted by the small-scale replica. Hence, researchers and practitioners may experiment with smaller-scale prototype farms and extrapolate the performance of larger-scale farms. To address some limitations of our experimental testbeds, we plan to gain access to a real large-scale farm and test SHRiNK there.

References

[1] The Apache web-server. `http://httpd.apache.org`, accessed January 2002.

[2] Paul Barford. *Modeling, Measurement and Performance of World Wide Web Transactions.* PhD thesis, Boston University, 2001.

[3] Paul Bradford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, 1998.

[4] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, 2001.

[5] Anja Feldmann. Characteristics of TCP connection arrivals. In K. Park and W. Willinger, editors, *Self-similar network traffic and performance evaluation.* Wiley, 2000.

[6] M. Harchol-Balter, M. Crovella, and C. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, November 1999.

[7] Erich M. Nahum, Marcel-Catalin Rosu, Srinivasan Seshan, and Jussara Almeida. The effects of wide-area conditions on www server performace. In *Proceedings of ACM SIGMETRICS*, 2001.

[8] R. Pan, B. Prabhakar, K. Psounis, and M. Sharma. A study of the applicability of a scaling-hypothesis. In *Proceedings of ASCC*, 2002.

[9] R. Pan, B. Prabhakar, K. Psounis, and D. Whischik. Shrink: A method for scaleable performance prediction and efficient network simulation. In *Unpublished manuscript*, 2002.

[10] Peakstone. `http://www.peakstone.com/technology.html`, accessed January 2002.