ELSEVIER Performance Evaluation 33 (1998) 5–26

PERFORMANCE EVALUATION An International Journal

Web server performance measurement and modeling techniques

John Dilley [a,1], Rich Friedrich [a,*], Tai Jin [a,2], Jerome Rolia [b,3]

[a] *Hewlett-Packard Laboratories, Hewlett-Packard Company, 1501 Page Mill Road, Mailstop 1U17, Palo Alto, CA 94304-1126, USA*

[b] *Systems and Computer Engineering, Carleton University, Ottawa, Ont., Canada*

Abstract

The popularity of the Internet, and the usage of the world wide web in particular, has grown rapidly in recent years. Thousands of companies have deployed Web servers and their usage rates have increased dramatically. Our research has focused on measuring, analyzing and evaluating the performance of Internet and Intranet Web servers with a goal of creating capacity planning models. We have created layered queuing models (LQMs) and demonstrated their superiority to traditional queuing network models since they incorporate layered resource demands. Along the way we built a tool framework that enables us to collect and analyze the empirical data necessary to accomplish our goals.

This paper describes the custom instrumentation we developed and deployed to collect workload metrics and model parameters from large-scale, commercial Internet and Intranet Web servers. We discuss the measurement issues pertaining to model parametrization and validation. We describe an object-oriented tool framework that significantly improves the productivity of analyzing the nearly 100 GBs of measurements collected during this workload study interval. Finally, we describe the LQM we developed to estimate client response time at a Web server. The model predicts the impact on server and client response times as a function of network topology and Web server pool size. We also use it to consider the consequences of server system configuration changes such as decreasing the HTTP object cache size. © 1998 Published by Elsevier Science B.V. All rights reserved.

Keywords: World wide web; Performance model; Capacity planning; Workload characterization

1. Introduction

The world wide web (WWW) [1] is a phenomenon which needs little introduction. Very briefly, the WWW employs a client/server architecture for access to a variety of information resources: a client *Browser* application locates a server host, and using the hyper text transfer protocol (HTTP) sets up a network connection using TCP/IP with the HTTP *server* on that host. The client then requests one or many documents, images,

* Corresponding author. Tel.: +1 650 857 5024; fax: +1 650 857 5100; e-mail: richf@hpl.hp.com.
[1] E-mail: jad@hpl.hp.com.
[2] E-mail: tai@hpl.hp.com.
[3] E-mail: jar@sec.carleton.ca.

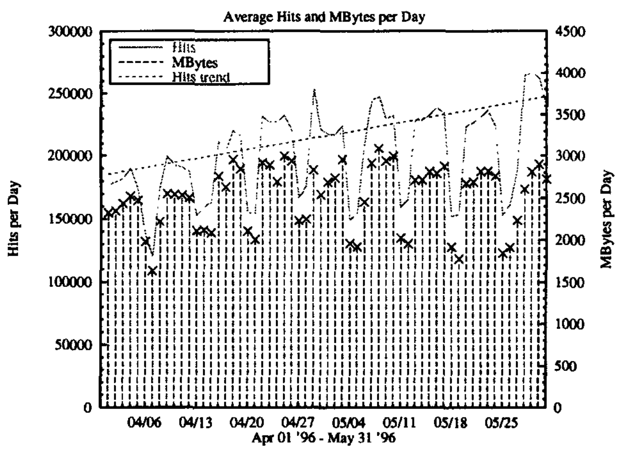

Fig. 1. Daily traffic trend for an Internet Web server.

or other media. The server returns the data to the client, which then displays it to the end user using whatever method is appropriate for the content. The primary driving factors for the rapid growth of the WWW include rich content types, ease of use, prevalence of TCP/IP, and ease of publishing.

Network administrators and webmasters at professionally managed business sites need to understand the ongoing behavior of their systems. They must assess the operation of their site in terms of the request traffic patterns, analyze the server's response to those requests, identify popular content, understand user behavior, and so on. A powerful set of analysis tools working in tandem with good modeling techniques are an invaluable assistance in this task.

Results from a two-month period at one of our commercial Internet measurement sites is illustrated in Fig. 1. This figure plots the daily number of HTTP hits at the server, and the amount of data transferred, and the hits per day trend line. It shows the periodicity of requests with the day of the week, and documents a dramatic increase in the number of hits per week – a linear regression shows traffic increasing at a rate of 7000 hits per week and 37 MB per week. This is a powerful motivator for capacity planning of these services. Effective performance management requires systematic measurement and efficient modeling. Consequently, we defined a set of metrics necessary to characterize Web server workloads and then developed custom instrumentation to collect these measures from a set of operational Web servers. The measurements are used to parametrize an analytic model for capacity planning and experimental design analysis purposes.

1.1. Related work

Previous studies of WWW performance have provided many useful insights into the behavior of the Web. At the University of Saskatchewan Arlitt and Williamson [2] examined Web server workloads through an in-depth study of the performance characteristics of six web server data sets with a range of time periods, from one week to one year. In addition that group has studied and modeled the workload from the client perspective by instrumenting the Mosaic browser [3]. One result of their work is a proposed set of workload common characteristics for Web traffic. We have corroborated several of these findings with our own data.

Cunha et al. [4] at Boston University also instrumented a Mosaic browser and studied the characteristics of the resulting client traces. They observe that the distribution of document sizes and document popularity profiles often follow a power-law distribution. Under a power-law distribution, popular content is very popular, hence server side caching can be effective at reducing server requests (serving hot content directly out of memory).

Almedia et al. [5] published a paper characterizing reference locality in the WWW. The paper examines spatial and temporal locality of reference in WWW traces exploring for evidence of long-range dependence and self-similarity in the traffic patterns. That paper supported the power-law distribution findings of the previous paper and found evidence of long-range dependence in Web traffic: Web server request traffic is bursty across varying timescales.

Kwan et al. [6] at NCSA studied user access patterns to NCSA's Web server complex. This paper presents an in-depth study of the traffic at their web server over a five-month period. The NCSA site was the busiest of the early sites (since most Mosaic browsers when they started made a request to the site), although traffic has since declined. Their findings illustrate the growth of traffic on the WWW, and describe the request patterns at their server.

Our study has examined logs from very busy, large-scale open Internet and private Intranet commercial sites over a period of serveral months. We have focused on understanding the fundamental workload characteristics and capacity planning techniques for these large Intranet (10 000 + users) and Internet sites (100 000 + hits per day). In particular we have developed new techniques to visualize workload parameters from extremely busy sites, and to model traffic at these sites.

The workload characteristics of the Web are fundamentally different from other well-studied information systems, such as the network file system (NFS) and on-line transaction processing (OLTP) systems. In OLTP environments the median response message size is on the order of 1 KB; by contrast Web responses are substantially larger, on the order of 4 KB, often with a heavy tail depending upon site content. In distributed file systems the data requests are for one or more fixed sized blocks of file system data; but Web servers return data that is variable in size, and increasingly requires CPU processing (for dynamic content and processing forms).

Web workloads often require significant network protocol processing time and have high communication latency. In addition, links in the Web tend to be much wider area in scope, meaning that a browser may visit several different sites possibly separated by a large geographical distance in a relatively short period of time. By contrast in OLTP and distributed file system workloads a client tends to exercise the same server or relatively small set of local servers. Techniques used to study Web serves must take into account the difference in workload and application architecture.

1.2. Web server system modeling

The architecture of a typical WWW service environment is shown in Fig. 2. Our focus is on the Web server system (hardware and software). The Web server receives requests for its content from one or more client browser applications. Requests arrive at the `httpd` Listener process on the server (1) and are dispatched to one of a pool of `httpd` Server processes (2). The number of servers in the pool can vary between a fixed lower and an elastic upper bound; each process can serve one request at a time. If the request is for HTML (hypertext markup language) or image content the `httpd` Server retrieves the content and returns it directly to the client. If the request is for dynamic (common gateway interface or CGI) content the `httpd`

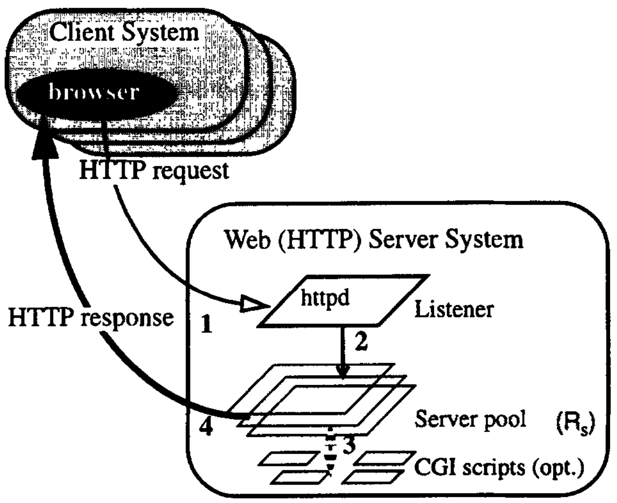

Fig. 2. Web server HTTP request processing.

server creates a child process which runs a CGI script to compute the requested information (3) and return output for the server to send back to the client. All these processes excecute in user space on the Web server.

Our research goal was to construct a model of commercial Web servers with parameters that could be measured from a real system, allowing us then to apply the model to help understand the relationship between Web servers, clients, and the Internets and Intranets that connect them.

Layered queuing models (LQMs) have been proposed to study distributed application systems [7–9] and we extend and apply them in this paper to address Web servers. They are extended queuing network models (QNMs) [10] that consider contention for processes as well as physical resources such as processors and disks. An LQM can be used to answer the same kinds to capacity planning questions as QNMs, but also estimate client queuing delays at servers. Fig. 2 illustrates clients competing for access to a server. When the number of clients is large client queuing delays at the server can increase much more quickly than server response times and are a better measure of end-user quality of service (including client response time).

This paper documents our methodology for WWW performance evaluation and is structured as follows.

- Section 2 discusses the metrics and instrumentation required to capture response time and service demand at WWW server sites.
- Section 3 presents the performance analysis toolkit that supports rapid data reduction and visualization for large measurement logs.
- Section 4 motivates the need for LQMs by first developing a traditional queuing network model and then illustrating the weaknesses of this approach. We then describe LQMs and present an LQM for one of the Web sites we measured. The model defines the relationship between the number of `httpd` processes, the number of CPUs, network delays, and client response times. We use it to explore the effects of different network technologies, Web server pool size, and HTTP object cache size.
- Section 5 summarizes our contributions and outlines areas of future research.

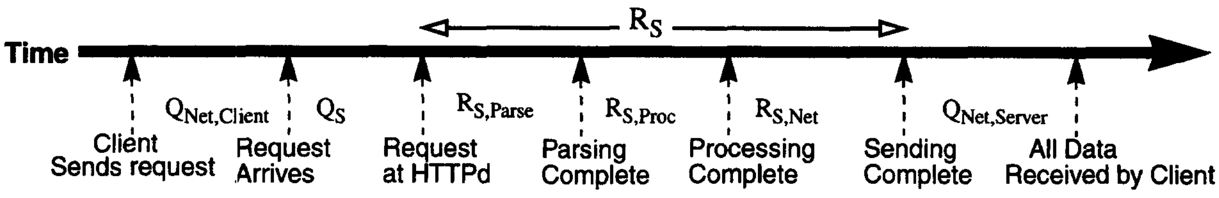

Fig. 3. HTTP request timeline illustrating performance instrumentation intervals and related metrics.

The characterization results of the Web server workload itself is not a goal of this paper. It is the subject of a future paper.

2. Metrics and instrumentation

This section describes the workload metrices of interest for capacity planning, the custom instrumentation that we incorporated into the NCSA WWW server process, and the measurement challenges that remain unsolved.

Our study focused on the following metrics: server response and service demand, and client residence time at the server. The specific metrics are described in detail below and are illustrated in Fig. 3 using a timeline of an HTTP request by a client to a Web server.

- R_S – *Server response time* is the time that a single HTTP request spends at the Server pool process. It includes its service time and queuing delays at physical resources in order to complete the request; it does not include queuing delays in the network or at the server prior to the request reaching the Server pool process. It consists of the following subcomponents.
  - $R_{S,\text{Parse}}$ – *Server parse time* is the time that the server spends reading the client request.
  - $R_{S,\text{Proc}}$ – *Server processing time* is the time that the server spends processing the request.
  - $R_{S,\text{Net}}$ – *Server network time* is the time that it takes for the server to reply to the client's request.
- Res_C – *Client residence time* is the queuing delay plus the Server response time for one visit to the web server (i.e. a single HTTP request).
- R_C – *Client response time* is the network communication latency plus the client residence time (i.e., end-to-end or last byte latency for one request).
- X_{Server} – *Server throughput* is the number of completed HTTP requests per unit time processed by the server.
- D_S – *Server service demand* is the amount of system resources consumed by each client HTTP request. It consists of the following subcomponents.
  - $D_{\text{s,CPU}}$ is the average CPU time for the request.
  - $D_{\text{s,Disk}}$ is the average disk demand for the request.
  - $D_{\text{s,Net}}$ is the average network delay for the request.

R_S is distinct from the classical "response time" metric in seveal ways.

- Server response time does not include all of the network time of the request. It does not capture the time between the user's initial click at the client browser, the latency in the network, or the server time to dispatch the request to the `httpd`; nor does it record the time between when the server writes the last byte to the network stack and the time the last byte accutually arrives at the client. Even if the end-to-end network delay is accurately measured, there may be additional time for the client browser to display the information to the user.

- The server response time metric is recorded for each *hit* at the server, i.e., for a single browser HTTP request and not for the end user's request, which may require several hits (i.e., HTTP requests) in the case of an HTML page or CGI request that contains inline images. When a browser receives a request with inline images (HTML `IMG SRC` derectives) it usually attempts to retrieve and display each of those images as part of the page the user selected. A more precise response time metric must take into account the aggregate latency of retrieval for all of the images visible to the user.

R_{S} is important for understanding server utilization and hence for capacity planning. R_{S} underestimates $\mathrm{Res}_{\mathrm{C}}$ because R_{S} does not include client queuing delays at the server. R_{C} on the other hand includes latencies beyond the control of the Web server so it is not a good measure of client quality of service at the server (R_{C} is always larger than $\mathrm{Res}_{\mathrm{C}}$). $\mathrm{Res}_{\mathrm{C}}$ is a better measure of end-user quality of service for a Web server. Since this value cannot be measured directly with server-side intrumentation we use LQMs and analytic performance evaluation techniques to estimate it based upon the measured values R_{S}.

Based on our experience the metrics described in this section are the minimal number needed for modeling a Web server. However, we found little use for $R_{\mathrm{S,Parse}}$ and have concluded that it is not necessary to measure it for the purpose of Web server capacity planning.

The commercial sites we studied used a variety of Web server implementations from OpenMarket, NCSA, and Netscape. All these systems record incoming requests but for varying metrics and with various precision.

- The OpenMarket server records its timestamps with microsecond precision and records both request arrival and completion times. The server's response time (R_{S}) is the difference between these times.
- The standard (public domain) NCSA server records request completions with one second precision but does not measure the server response time.
- The Netscape server provides some ability to customize the logging detail level, but does not offer high precision timestamps or server response time measurements.

Two of our most important sites were running the NCSA implementation during the period of our study. Since the source code was available we added custom instrumentation to measure the R_{S} and $D_{\mathrm{S,CPU}}$ values for each request at the server. This data was written in extra fields in the `httpd` logs. The following section describes this in more detail.

2.1. Measurement issues

Our choice of metrics leads to some measurement issues due to the instrumentation being in the application and not on the network or in the server host's network stack. The measurement intervals are identified in Fig. 3, and show that R_{S} does not measure the network and queuing delays from the time the client made the request (the click of the hypertext link) and when the request arrived at the `httpd` process ($Q_{\mathrm{Net,Client}}$ and Q_{S} in Fig. 3). R_{S} also does not measure the network queuing and transmissions delay for the last window of TCP data which is queued in the server's network buffers after the final `write` system call completes ($Q_{\mathrm{Net,Server}}$ in the figure). Our technique does measure the delay due to network congestion for all packets except for those in the last window since they must be received and acknowledged before the last window can be queued for delivery to the server's network stack.

While this metric does not provide an accurate indication of client response time at the server, it does measure the time a Server pool process spends servicing each request. This is the data we use to construct our analytic models from which we estimate server queueing delay and client response time.

2.2. *NCSA* `httpd` *instrumentation*

The NCSA 1.5 `httpd` was instrumented to provide server response time (R_S) and CPU service demand ($D_\mathrm{s,CPU}$). R_S is measured as the real or wall-clock time spent processing the request using the `gettimeofday(2)` system call rounded to millisecond (MS) precision. We distinguish between different HTTP request types (e.g., HTML, Image, CGI, etc.) and further subdivide the metrics.

$D_\mathrm{s,CPU}$ is measured using the `times(2)` system call, which indicates the user and system CPU time consumed by the process and its children with millisecond precision (the accuracy is system dependent but is typically 10 ms).

The instrumentation captures the three separate R_S intervals which when combined comprise the overall server response time, R_S. $R_\mathrm{S,Parse}$ measures the length of time from the client connection to the `httpd` server process to the reading of the first line of the HTTP request. $R_\mathrm{S,Proc}$ measures the time spent by the server in processing the request, including reading in the rest of the request and internal processing up to the point of sending the response. $R_\mathrm{S,Net}$ begins with the sending of the response and ends with the final send of data from the `httpd` Server pool process. $D_\mathrm{s,DPU}$ is measured over the same time period as R_S.

The source code changes to implement this instrumentation were minimal. Calls to `gettimeofday(2)` and `times(2)` were added to the beginning of the `get_request` routine. This is the entry point into the request processing: a connection has already been established with the client; `get_request` then reads the first line of the HTTP request. At the end of the routine another `gettimeofday(2)` call is made to mark the end of the first interval and the beginning of the second interval.

To measure the end of the second interval and the beginning of the last interval, a `gettimeofday(2)` call was added to the `begin_http_header`, `error_head`, and `title_html` routines. These routines begin the sending of the response to the client. The final change is to the `log_transaction` routine. This is the routine which logs the request to the access logfile. At the beginning of the routine calls are made to `gettimeofday(2)` and `times(2)` to measure the end of the last server response time interval and the total CPU time, respectively.

R_S and D_S are logged as two additional fields appended to the end of the common log format entries in the access logfile. Both fields have the format ms/ms/ms. For R_S each millisecond value represents each of the three intervals of the server response time. For D_S the first and second values represent the CPU time spent by the server process in the kernel and in user spaces, respectively; the third value represents the CPU time spent by any child processes in both kernel and user spaces (child processes satisfy CGI requests).

When used in analysis the three intervals of R_S are typically combined to obtain a single server response time value since the individual intervals are not precise measures (the point at which the three intervals end is imprecise due to the structure of the code where processing and sending data overlap).

The overall impact of the code changes was slight; only a few lines were changed in the daemon and the performance impact of these additional systems calls was insignificant when compared with the amount of work done by an HTTP connection request. The additional information allowed us to develop predictive models for the system with minor perturbation on the running system.

3. Log analysis framework and tools

The process of analyzing HTTP server performance begins with the collection of log files spanning some analysis time period. To understand traffic trends we collected logs over a period of 2–12 months at our instrumented sites. All of our sites compressed their log files with `gzip` prior to retrieval. At our busiest site the *compressed* logs were on the order of 30–40 MB per day (240 + MB per day uncompressed); our second busiest Internet site's logs were 6–8 MB per day (30 MB per day uncompressed). The logs are kept compressed on an HP 9000/755 workstation with a modest disk farm.

Analysis of these logs presented a challenge: we could not hold all of them uncompressed, yet uncompressing them each time we wished to perform some analysis was intractable considering the number of days we wished to examine. Even if we could store the log files uncompressed, the state of the art tools (most of them implemented as `perl` scripts) [11–14] often have to re-parse every line of the log file each time that file is examined. So in order to achieve "same day service" when analyzing months of data we constructed an object-oriented software framework, the log analysis framework, and a set of custom data reduction, analysis, and visualization tools built atop the framework.

By converting the raw ASCII log files into a record-oriented binary format and mapping them into system memory our tools are able to operate with a 50x speedup as compared with previous perl tools while saving 70–80% of the disk space required by the ASCII logs. Plus the common library facilities make writing new tools almost trivial.

3.1. Data conversion

The first step in the process is to convert the compressed or uncompressed `httpd` log files into a reduced binary representation in which the request timestamp is stored as a UNIX `time_t` or `timeval` structure, the number of bytes returned is stored as an unsigned integer, the URL (uniform resource locator, or content location) is stored as an index into a string symbol table, and so on. The conversion utility creates two output files for each log file: the binary compressed data (with fixed size log records) and a symbol table containing the variable-length strings from the original log file (client addresses, URLs user IDs). One symbol table can be used by multiple log files since there tend to be relatively few unique URLs requested many times, and few clients making repreated requests.

The conversion utility reads Web server logs in any of the NCSA, Netscape, and OpenMarket formats. By reading multiple data file formats we are able to use the same reduction tools regardless of the type of Web server used at the site.

3.2. Data reduction

Once the files are compressed into their binary format a set of tools reduce the data by extracting various data fields from the logs. The log analysis framework provides access to the log data as an array of fixed-length records in memory. By mapping the converted data files into memory the framework saves I/O overhead at application start-up time and uses the operating system to manage data access; furthermore use of memory mapping avoids a copy of data onto the heap.

Application access this data using the C++ standard template library (STL [15,16]) *generic programming* paradigm. Each log record is viewed as an element in a *container* (log life) with an associated *iterator* type; an instance of the interator points to each record in turn, starting with the element returned by the container's

```
// Compute the response size distribution

#include <stdio.h>
#include <stdlib.h>
#include "memLogFile.H"
#include "command.H"
#include "distribution.H"

int main(int argc, char * argv[])
{
  Options opt(argc, argv);

  try {
    Distribution respSize(opt);
    const char * arg;
    for (; arg = *opt; ++opt) {
      MemLogFile    log(arg);

      MemLogIter it = log.begin(opt);
      for (; it != log.end(); ++it) {
        respSize.add((*it).respBytes);
      }
    }
    respSize.printDist("size", ".dist");
  }
  catch (int err) {
    fprintf(stderr,
         "Exception %d: exiting", err);
    return 1;
  }
  return 0;
}
```

Fig. 4. Sample data reduction application.

`begin` method, incrementing using the iterator's `operator++` method until reaching the container's `end` interator position. At each record the iterator is *dereferenced* via its `operator*` method; the application can access each of the elements of the log record (i.e., `(*iterator).requestUrl`).

Using framework capabilities an application can visit each log record in all log files specified on the common line using about a dozen lines of boilerplate code (including exception handling); this leaves the developer to focus on the custom logic to analyze the data in the log files rather than writing more parsing code. The code sample in Fig. 4 demonstrates the ease of creating new data reduction tools using the library framework. This code examines one or more log files and creates a file with the response size distribution of all requests logged in those files.

The code uses class `Options` to parse the command line flags and pass them to instances of class `MemLogFile` and class `Distribution` to control their behavior. Then for each argument left on the command line, the application constructs an instance of class `MemLogFile` to refer to that file; it maps the data file into memory and throws an exception of anything goes wrong. Next an instance of class `MemLogIter` is created to iterate through all records in the log file. The iterator only pauses at log records

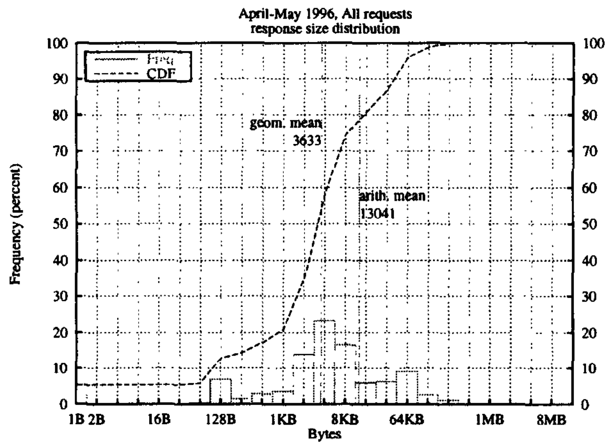

Fig. 5. Example response size distribution.

that match the filter options specified on the command line. After each increment the iterator is dereferenced to access the underlying struct `LogRecord`, and in this application to extract the response size in bytes from that record. After all matching log records have been examined (i.e., the iterator has reached the end of the container) the `Distribution` object is printed to a file "`size.dist`". This file can then be given to a visualization tool.

We built a set of standard reduction tools which perform common tasks that someone viewing an HTTP log file might wish to know, including:

- overall statistics for total hits (client accesses), bytes transferred, and percent of HTTP objects requested;
- distributions and means of response sizes, response times and interarrival times;
- summaries of traffic in hits and bytes by various time periods (hourly, daily, weekly).

3.3. *Visualization*

The output of the reduction tools is typically a tabular ASCII file often summarized in the form of a frequency distribution. In order to make sense of this information a visualization tool sends the tabular output into `gnuplot` or some other graphics or spreadsheet tool that converts the raw data into more meaningful charts and graphs. The reduced data formats are sufficiently flexible that various graphics or spreadsheet packages can be used for performance analysis.

An example of our visualization tool's output can be found in Fig. 5. In this figure the response size distribution of all requests during the measurement interval of April–May 1996 are plotted. We define the *response size* as the size in bytes of the HTML object returned by the Web server. Two plots are graphed: a histogram and a cumulative distribution function (CDF). For this Web server the median response size was slightly less than 4 KB. The 90th percentile size is greater than 32 KB.

3.4. *Uniform filtering capability*

When analyzing a response size distribution some questions can arise of the form, "What traffic is responsible for the peak at 4 KB, or, "What is the server response time of requests from the AOL domain?"

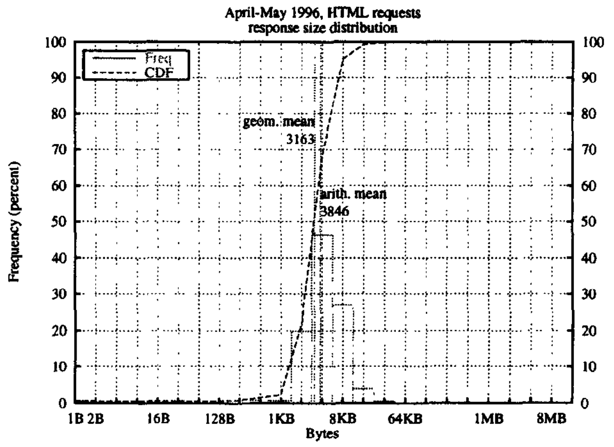

Fig. 6. Example response size filtered distribution.

To answer these quenstions requires reexamination of a subset of the data in the log files that match some criteria, or pass some *filter*. Due to the size of the data, extracting each subset and placing it in a separate file is impractical (because of both size and time concerns). In addition, reuse of existing software is essential to maximize developer and user productivity.

The uniform filtering capability allows logs to be filtered transparently to the accessing code; therefore a single tool can operate on entire logs or, with a set of predefined command line arguments, only on the subset of the log entries matching the user-specified filter criteria. The standard filters include client host ID, timestamp (day of week, or month and day), size (range or exact value), and many more. Fig. 6 plots the response size distribution from the same Web server and measurement period plotted in Fig. 5 but filtered to plot only HTML requests. Comparing the two figures, the distribution for HTML objects is much narrower with a similar median size but with a 90th percentile value of less than 8 KB.

One of the most powerful filters is the user request filter, which transparently aggregates one or more log records (individual browser *hits*) into a single logical *user request* consisting of the URL the user visited and all inlined images and forms that were returned as a result of the user's click. The user request filter is most useful in environments where users are uniquely identified by IP address. When many users visit a site through a firewall or proxy cache they appear to the server to come from the same IP address; in this case the filter often cannot distinguish between the hits belonging to one user and those of another user coming through the same proxy or firewall. On our busy Internet servers, a majority of the hits appear to come through firewalls at large corporate sites or Internet service providers (ISPs). The presence of these firewalls and proxies reduces the ability of our current technique to aggregate individual hits into a single user request.

3.5. Statistical analysis

Now that the response size for each request is known how should the data be visualized? There are several possibilities, but the most concise way to view the data distribution is through a histogram and cumulative distribution function (CDF) plot. A histogram displays the data summarized into *buckets*; each bucket contains the proportion of observed values that fell into the range of that bucket. The *CDF* plot indicates the proportion of the observed values whose value was less than that bucket's value. The point at which the CDF plot crosses the 50% mark indicates the median of the observed sample. We also typically indicate the arithmetic and geometric means using vertical lines in the distribution.

Most data from Web servers tend to have a very wide dynamic range, and approximates a log-normal distribution. Therefore we usually apply a logarithmic transformation to the data prior to placing it into buckets. Since this is such a prevalent component of data analysis we created the class `Distribution` to assist with the collection and reporting of data. Each observed data value is added to a `Distribution` class instance; after all files have been processed the distribution is printed to a file. The distribution provides a concise representation of the data population, including its arithmetic mean and geometric mean (which can be useful with a log-normal distribution), standard deviation, minimum and maximum values, and the bucket populations. Visualization tools read the distribution output and create the charts and graphs used for our analysis.

4. Layered queuing models for Web server capacity planning

Measurement tools are essential for characterizing the current and past behavior of a system. However it is also important to be able to anticipate future behavior with different hardware and software configurations or under different workloads. For this reason we developed an analytic queuing model for the system. Analysis is chosen over simulation because of speed and ease in interpreting results. We rely on mean value analysis (MVA [10,17]) based techniques and queuing network models (QNMs) for our analysis.

In the following sections we describe a QNM for a Web server. Deficiencies in the model motivate the need for a more complex queuing model technology and we utilize layered queuing models (LQMs) [7]. LQMs are introduced and a brief outline is given for the method of layers performance evaluation method for LQMs. An LQM and its parameters are given for the Web server considered in the study. The parameters were gathered using the instrumentation and tools described in Sections 2 and 3. Last, we use the model to predict client response response times under different Web server and network configurations.

4.1. A queuing network model for a Web server

Queuing network models (QNMs) have been in use for over 25 years as a model of computation for capacity planning in mainframe environments and are available as commericial products [22,23]. Mean value analysis (MVA) has been used as a performance evaluation method for these models providing fast yet accurate estimates of system performance measures. This section describes the parameters for open QNMs and then develops a simple open QNM for an example Web server. Open models assume a very large number of customers with an arrival rate that is not influenced by server response times and where the arrival rate equals the completion rate. These assumptions are reasonable for larger commercial sites with low measures of connection failure.

The purpose of the predictive model is to capture those features of the system that affect performance most. In a QNM these features are assumed to be the queues for devices. The parameters for a single class product-form open QNM are as follows:

- λ the arrival rate of the class,
- D_k the mean demands of the class at each device k in the model.

Table 1
QNM model parameters for an example Web server processing a CGI script

CGI processing step	CPU (ms)	Disk (ms)
Accept a TCP/IP connection	1.0	
Pass TCP/IP connection to HTTP server process	1.0	
Fork a child process for CGI processing	2.0	
Application processing	138.0	8.0
Pass results back to HTTP server process	4.0	
Pass results back to client via TCP/IP connection	4.0	
Total resource demands	150.0	8.0

Table 2
Mean response time of the server as a function of arrival rate λ

λ Server arrival rate (s^{-1})	R_{server} (s)
1	0.18
2	0.22
3	0.28
4	0.38
5	0.61
6	1.51
6.5	6.01

The results of an MVA include:

- R_k the mean response time of the class at each device k,
- R the total mean response time of the class.

In this section we consider a QNM for an example Web server that services a single class of users that only issue CGI requests. To build the model we enumerate the steps involved in serving a CGI request, associate them with service times, and then summarize the service times to get the parameters for the QNM model. Note that the service times used here are only estimates for the purpose of this example and do not correspond to any particular system. Table 1 summarizes the processing steps and service times for our example CGI request.

The formula for mean response time for a single class open QNM with devices having processor sharing (round-robin) scheduling is as follows [10]. For the purpose of clarity we include the workoad class name *server* in the subscripts of the model paramters.

$$\begin{aligned} R_{\text{server}} &= R_{\text{server,cpu}} + R_{\text{server,disk}} \\ &= D_{\text{server,cpu}}/(1 - \lambda_{\text{server}} D_{\text{server,cpu}}) + D_{\text{server,disk}}/(1 - \lambda_{\text{server}} D_{\text{server,disk}}) \end{aligned} \quad (1)$$

In this equation λ_{server} is the arrival rate of customers making CGI requests at the server and $D_{\text{server,cpu}}$ and $D_{\text{server,disk}}$ are the aggregate mean demands of each request.

Table 2 shows the mean response time as the arrival rate increases from 0 to 6.5 requests per second. The model suggests that up to six requests can be handled per second before response times start to grow rapidly. At six requests per second the mean aggregate response time of a CGI request at the CPU and Disk is approximately 1.5 seconds.

These results can be used to estimate how much memory should be set aside for server processes that support client connections and the CGI requests. If too few resources are allocated, clients may suffer queuing delays before accessing the site. Little's law [10] provides the average number of server processes N_{server} that are needed:

$$N_{\text{server}} = \lambda_{\text{server}} R_{\text{server}}. \tag{2}$$

With an arrival rate of six requests per second N_{server} is equal to 9. Thus, the site will need a mean of nine server processes.

A problem with this approach is that we have not taken into account the nested delay for the network time required to convey results to the client, namely $R_{\text{S,Net}}$. If we add an Internet delay of $R_{\text{S,Net}}$ equal to 2 s to R_{server}, then the extimate for N_{server} increases to a mean of 21 processes. The greater the duration of Internet delays the more server processes that are needed. The actual number of processes needed to avoid queuing delays will be higher and is a nonlinear function with respect to the mean number of processes.

The layering of demands for $R_{\text{S,Net}}$ is similar to the problem of simultaneous resource possession [24] but can be much more complex. For example, a CGI process may visit some other Web server or a database, either on the current node or on another node in the network. The nested request can suffer queuing delays waiting for access to this other server, network latencies, and delays for further nested requests while in service.

As demonstrated by the inclusion of the Internet delay $R_{\text{S,Net}}$ such contention can significantly impact server and client performance and thus must be reflected in the performance model. LQMs, introduced in the next section, provide a more appropriate model of computation for these kinds of systems than QNMs.

4.2. Layered queuing network models

In this section we describe layered queuing models (LQMs) and briefly outline the method of layers solution algorithm for LQMs. We then develop an LQM for the Web server and identify its parameters. These parameters are gathered using the instrumentation and tools described in Sections 2 and 3. Last, we use the model to predict client response times under different Web server and network configurations.

LQMs and QNMs extended to reflect interactions between client and server processes. The processes may share devices and server processes may also request services from one another. LQMs are appropriate for describing distributed application systems such as CORBA [18], DCE [19], and Web server applications. In these applications a process can suffer queuing delays both at its node's devices and at its software servers. If these software delays are ignored, response time and utilization estimates for the system will be incorrect.

The method of layers (MOL) [7,21] and stochastic rendezvous network (SRVN) [9] techniques have been proposed as performance evaluation techniques that estimate the performance behavior of LQMs. Both evaluation techniques are based on approximate MVA.

The MOL is an iterative technique that decomposes an LQM into a series of QNMs. Performance estimates for each of the QNMs are found and used as input parameters of the other QNMs. The purpose of the MOL is to find a fixed point where the predicted values for mean process response times and utilizations are consistent with respect to all of the submodels. At that point the results of the MVA calculations approximate the performance measures for the system under consideration. Intuitively, this is the point at which predicted process response times and utilizations are balanced so that each process in the model has the same throughput whether it is considered as a customer in a QNM or as a server: the rate of completions of the server equals the rate of requests for its service according to the flow balance assumption, and the average service time required by callers of a process equals its average response time.

The following are the parameters for an LQM:

- process classes and their populations or threading levels;
- devices and their scheduling policies;
- for each service s of each process class c:
  - $V_{c,s,k}$ the average number of visits to each device k;
  - $S_{c,s,k}$ the average service time per request at each device k;
  - $V_{c,s,d,s2}$ the average number of visits to each service $s2$ for each server process class d.

Note that the client's service times at the services it visits are not specified. They are the response times of the services R_s and must be estimated by performance evaluation techniques.

Services identify a resource visit specification that characterizes the types of work supported by a server pool. A visit specification describes the physical resource demands required by a type of work within a server and its requests for service from other servers. With this extra degree of detail client and server service response times, and response times for specific client/server interactions can be estimated.

In addition to the results from MVA for QNMs, LQM output values include for each process class c:

- $R_{c,s}$ the average response time of each service s;
- $R_{c,s1,d,s2}$ the average response time of each client of service $s1$ at each service $s2$ of its serving process class d;
- Q_c the total average queue length of process class c;
- $Q_{c,s}$ the total average queue length of process class c and its service s;
- U_c the total utilization of process class c;
- $U_{c,s}$ the total utilization of process class c and its service s.

Residence time expressions have been derived by MOL for LQMs modeling several types of software interactions, including synchronous RPC and rendezvous, multi-threaded servers [20] and asynchronous RPC [21]. As we will show next, the resulting abstraction can be used to describe Web servers and other distributed applications.

4.3. An LQM for a Web server

An LQM for the Web server is shown in Fig. 7. The client class generates the workload for the server. We give it a single service with the name *Request*. It is used to generate the visits to all of the Web server's services. The Listener process has a single service named *Accept* that accepts client requests and forwards them to the server pool. This is reflected in the LQM as a visit by the client class to the Listener process and then a visit to the server pool. With this approach we maintain the visit ratio and blocking relationships present in the real system.

The server pool offers many services to clients. However, our workload analysis indicated that three services constitute the majority of the resource demand and that each requires significantly different resource demands. These three services supply *Image, HTML,* and *CGI* objects to the client. The *Image* and *HTML* requests use processor and disk resources of the server process. The *CGI* service spawns another process to execute a corresponding CGI program. The server pool process waits for the CGI program to complete so it can return results to the client. In general, these CGI programs could exploit middleware platforms such as CORBA, DCOM, RMI or DCE to interact with other layers of servers. However, this was not the case for this application. For this reason we included the spawned processes CPU demand in the *CGI* service.

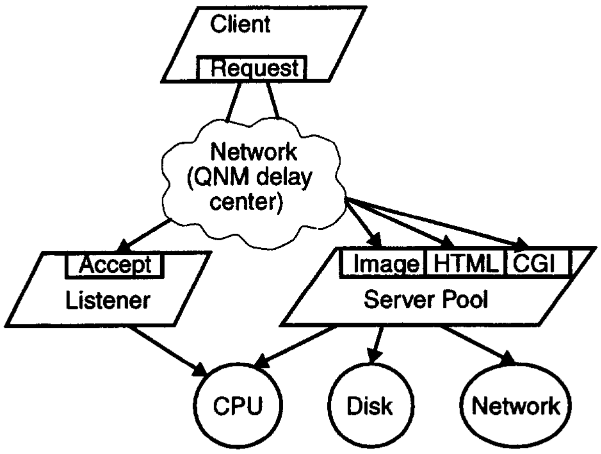

Fig. 7. Layered queuing model for a Web server.

To complete the request using the HTTP protocol, the server process must wait for all but the last TCP/IP window to be forwarded to the client over the network and acknowledged. The time to send results was measured as $R_{\mathrm{S,Net}}$. Since it includes relatively little CPU demand it gives a good estimate for the network delay; we use this value as an input parameter for our model. The network was represented as a delay center.The CPU and Disk had processor sharing disciplines.

The measurement tools were used to estimate the following LQM model parameters:

- for the *request* service of the client class *c*:
  - $\lambda_{\mathrm{Client}}$ the arrival rate of requests;
  - the average number of visits to Listener per request; $V_{\mathrm{c,request,Listener,accept}} = 1$;
  - the average number of visits to each Web service per request:
    - $V_{\mathrm{c,request,Pool,Image}} = 57\%$;
    - $V_{\mathrm{c,request,Pool,Html}} = 30\%$;
    - $V_{\mathrm{c,request,Pool,Cgi}} = 13\%$;
- N_{Pool} the number of processes in the server pool;
- for each service *s* of the server pool class of processes:
  - $D_{\mathrm{s,CPU}}$ the average CPU time of each service;
  - $D_{\mathrm{s,Disk}}$ the average disk demand of each service;
  - $D_{\mathrm{s,Net}}$ the average network delay.

Table 3 gives the parameters for the LQM used for validation and experimentation.

For clients the arrival rate of requests was 11 404 hits per hour. This is a low estimate because requests will be dropped when it is not possible for a client to establish connection. To emulate an open class we set the very large customer population of 1 00 000 each with a think time of 31567.871 seconds. For each request, the client visits the Listener once to get forwarded to a server pool process. The measured fraction of each type of hit gives the probability and hence average number of visits to each service.

The $D_{\mathrm{s,CPU}}$ values were captured for each request then aggregated by service type. The $D_{\mathrm{s,Disk}}$ could not be captured on a per service basis so they had to be estimated. Because of file system caching the $D_{\mathrm{s,Disk}}$ quantities were quite small compared to the measured response times for the services. As a result we believe our analysis is not sensitive to these estimates.

Table 3
Parameters for a Web server LQM

Parameter	Value (ms)
$D_{\text{image,cpu}}$	12.65
$D_{\text{html,cpu}}$	228.9
$D_{\text{cgi,cpu}}$	829.9
$D_{\text{image,disk}}$	18.4
$D_{\text{html,disk}}$	3.74
$D_{\text{cgi,disk}}$	2.76
$R_{\text{image,net}}$	14 234
$R_{\text{html,net}}$	626
$R_{\text{cgi,net}}$	2329

The $R_{\text{S,Net}}$ were captured for each request and were also aggregated by service types to estimate $D_{\text{S,Net}}$. They are large compared to CPU and disk times and have a significant impact on server behavior.

4.4. Server capacity planning using an LQM

In this section we use the LQM to consider trade-offs in server configuration for several different network topologies. We show that server configuration depends greatly on networking characteristics. For each experiment we consider the client response time which includes queuing for access to a server in the server pool, the time needed for the service, and the time up to the point the last portion of the request's results are submitted to TCP/IP.

We consider:

- Three types of network delay characteristics:
  - *Internet*: using data from our measurement site;
  - *WIntranet*, wide area Intranet: using a delay value of 1000 ms;
  - *LIntranet*, local area Intranet: using a delay value of 10 ms.
- Server pool sizes of between 1 and 60.
- Either one or two CPUs on the server node, (labeled in the figure with the suffix 1 or 2).

A second set of experiments revisits the three networking cases with one processor for the server: Intranet 1, WIntranet 1, and LIntranet 1. For these cases we decrease the amount of memory used to cache frequently requested pages thereby increasing the otherwise low utilization of the disk subsystem.

The client and server response times for the Image, HTML, and CGI workload classes are outputs of the model.

The wide area Intranet and local area Intranet examples make use of the same measured data as the Internet case for their workload characterization but use estimated network delay times for $R_{\text{S,Net}}$.

In Fig. 8 the response time is plotted as a function of the network type, workload class, and whether measured on the client or server. From these results we deduce that:

- Network delay has a dramatic impact on both server and client response times with the Internet having the worst response time characteristics.
- The addition of a second CPU on the server significantly improves the response time for the CGI workload but has less of an effect for the HTML case and very little for the Image case.

In Fig. 9 the response time is plotted for the Internet case as a function of the workload class and varying server pool size. From the results from the model we deduce that:

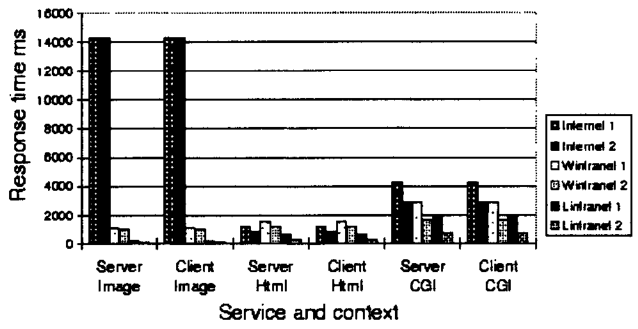

Fig. 8. Estimated client and server response times with thread pool size of 60.

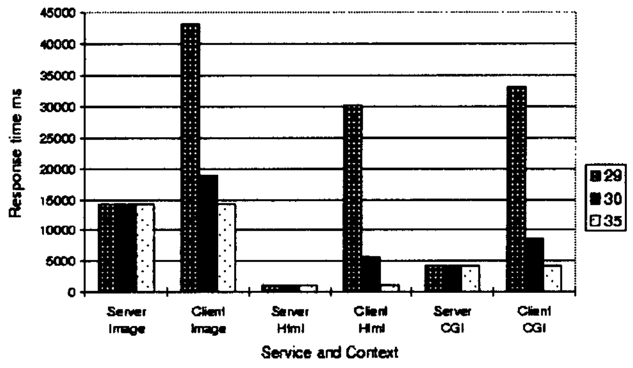

Fig. 9. Internet case with measured network delay and server pool sizes of 29, 30, and 35.

- Client response times can be much larger than server response times if the pool size is not sufficient.
- Increasing the size of the server pool significantly decreases client response times.
- Larger network delays require more servers in the server pool.

When the servers in the server pool approach full utilization, the estimated client response times increase very quickly. This is the case in Fig. 9. The CPU and Disk utilizations were approximately 60% and 3%, yet with low pool size the server processes were nearly fully utilized. This is because the Internet acts as a delay center that holds a server process until the last TCP/IP window worth of data is submitted. This causes the server pool to become a software bottleneck. In Fig. 9, the network delays were between 625 and 14 000 ms for the different types of services (and hence result sizes). If result sizes were smaller, for example on the order of 1 KB (which is less than an Ethernet frame size and typical for transaction processing requests), we would not expect to see such a large impact on system behavior. Next we consider the second set of experiments by revisiting the Intranet 1, WIntranet 1, and LIntranet 1 cases. We reflect a decrease in the amount of memory used to cache frequently requested HTTP objects by causing an increase in disk utilization from 3% to 20% and 40%. Table 4 shows the performance impact on client response times for these three different network topologies; as in Fig. 8 each topology has 60 server threads. Image requests have the greatest input-output requirements; they are the only class of requests to incur any significant degradation. The smaller the network delay the greater the relative degradation. From the table, we conclude that for systems with larger Internet delays (Internet) very large caches have little impact on

Table 4
Client response times (in ms) as a function of disk utilization (which is a function of HTTP object cache miss rate)

Disk utilization	3%	20%	40%	Impact on results
Internet Image	14 282	14 384	14 578	Small
HTML	1152	1172	1211	Small
CGI	4225	4239	4265	Small
WInternet Image	1048	1151	1344	Large
HTML	1526	1547	1585	Small
CGI	2897	2912	2838	Small
LIntranet Image	148	251	445	Most significant
HTML	628	649	688	Small
CGI	2004	2019	2047	Small

Table 5
Model validation values

Metric	Value (s)
$R_{\mathrm{S,Image}}$	13.0
$R_{\mathrm{S,Html}}$	1.0
$R_{\mathrm{S,CGI}}$	3.1

customer response times. This is because the Internet delays dominate access time. For local area networks (LIntranet) input–output system caches can significantly reduce client response times.

These results assume a very large population of clients with each arriving client completing its requests for service. In a real system calls will simply be lost from the socket queue when the server becomes too busy. However, when the system is well behaved then few calls will be lost. For these cases we believe the model captures the trend in client response times at the server.

This model and analysis describes a server with a fixed upper limit on the number of server pool processes. Some server implementations allow additional processes to be spawned when their configured upper limit is reached. However there is an eventual upper limit (e.g., the size of the process table or amount of physical memory). We can increase the workload and encounter this same effect at that limit. The key point is that the number of available processes in the pool must be sized based upon the expected workload and network delay.

4.5. Model validation

We validated the model using the measurements collected and described in Section 2. The model used as input data from six hours of data early in the two month investigation period. Using the full two month time period we computed the average response times as shown in Table 5.

These values agree closely with those predicted by the model as illustrated in Fig. 9 (compare these values with the corresponding 35-process server columns in the figure). This is not a full validation of the model but does indicate that the model is able to predict server performance with a limited amount of parametrization.

Finally, we consider how well the interarrival time distribution matches the QNM assumptions of exponentially distributed. Fig. 10 graphs the interarrival time of all HTTP requests to the Web server over the

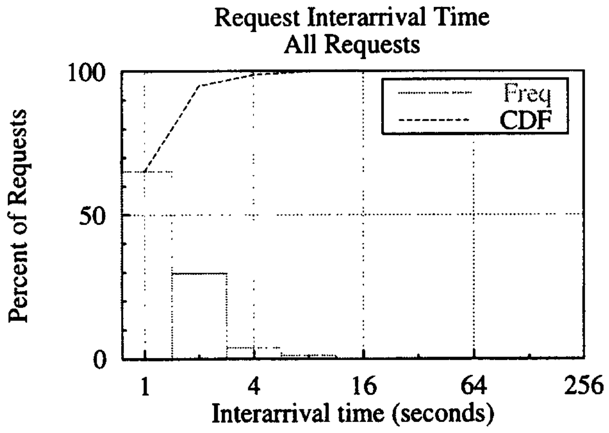

Fig. 10. HTTP request interarrival time distribution.

duration of the workload analysis period. Thus all hits are represented here and not individual local user requests (that, as we have described earlier, can consist of many separate HTTP requests). Unfortunately, the granularity of the timers was on the order of 1 s; consequently all of the hits less than 1 s are plotted in the first bucket. The result is an approximate exponential curve without a heavy tail. Note that the x-axis is logarithmic.

5. Summary and contributions

This paper described our contributions in measurement tools and modeling techniques for evaluating Web server performance. Specifically, we created custom instrumentation and collected representative workload data from several large-scale, commerical Internet and Internet Web servers over a time interval of many months. We developed an object-oriented tool framework that significantly improved the productivity of analyzing the hundreds of GBs of measurement data. The framework's binary data format allows a common set of tools to analyze data from diverse HTTP server implementations.

Metrics were chosen that allowed us to construct an LQM. We used the model to predict client response time at the server, a value which could not be measured directly but is a good measure of client quality of service at the Web server. We show that client response times are quite sensitive to the number of servers in the server pool and are more sensitive in environments with high network latency such as the Internet.

Another insight of our workload characterization analysis is the realization that Web workloads depend on a server's content and usage and both are evolving rapidly. This suggests that an automated process is necessary for collecting, analyzing and modeling the data required for effective Web server performance management.

6. Future research

Our future research is divided into two areas: measurement and modeling. Future measurement research will focus on characterizing the workloads of Web servers and proxy caches especially when connected to clients using high speed cable modems. We are researching techniques to aggregate individual HTTP requests into a single logical user request even when proxies and firewalls are located betwen client and server. We have also collected request traces from many heavily loaded Web servers and proxy caches and will analyze this data to determine the degree of self-similarity [2,5] in this traffic.

Our modeling research will focus on validating and extending this model beyond a single Web server to Web server clusters; integrating proxy caches into the network topology; aggregating hits into user requests and using the model to predict user request behavior; and using the model to explore Web server performance in the new broadband networks being deployed to the home via cable modem and digital subscriber line (ADSL, HDSL, etc.) technologies.

Acknowledgements

The authors would like to thank Martin Arlitt, Sailesh Chutani, Gita Gopal, Gary Herman, and Jim Salehi for their review of this paper. An earlier version of this work was published in Lecture Notes in Computer Science 1245 [25].

References

[1] T. Berners-Lee, R. Cailliau, J. Groff, B. Pollermann, World wide web: The information universe, in: Electronic Networking: Research, Applications and Policy, vol. 2, no. 1, Meckler, New York (1992) pp. 52–58.
[2] M. Arlitt, C. Williamson, Internet web servers: Workload characterization and performance implications, IEEE Trans. Networking 5 (5) (1997) 631–645.
[3] M. Arlitt, C. Williamson, A synthetic workload model for internet mosaic traffic, Proc. 1995 Summer Computer Simulation Conference, July, 1995.
[4] C. Cunha, A. Bestavros, M. Crovella, Characteristics of WWW client-based traces, Boston University Technical Report BU-TR-95-010, July 1995.
[5] V. Almeida, A. Bestavros, M. Crovella, A. de Oliveira, Characterizing reference locality in the WWW, Proc. Parallel and Distributed Information Systems, 1996.
[6] T. Kwan, R. McGrath, D. Reed, User access patterns to NCSA's World Wide Web server, IEEE Computer 28 (11) (1995).
[7] J. Rolia, K. Sevcik, The method of layers, IEEE Trans. Software Engrg. 21 (8) (1995) 689–700.
[8] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, C.M. Woodside, A toolset for performance engineering and software design of client–server systems, Perform. Eval. 24 (1–2) 117–135.
[9] C.M. woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, The stochastic rendezvous network model for performance of synchronous client–server-like distributed software, IEEE Trans. Comput. 44 (1) (1995) 20–34.
[10] E. Lazowska, J. Zahorjan, G. Graham, K. Sevcik, Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice-Hall, Englewood Cliffs, NJ, 1984.
[11] getstats: Web server log analysis utility. URL: `www.eit.com/software/getstats/getstats.html`
[12] WWWstat: HTTPd logfile analysis software. URL: `www.ics.uci.edu/pub/websoft/wwwstat/`
[13] Wusage 4.1: a usage statistics system for web servers. URL: `www.boutell.com/wusage/`
[14] Analog: a WWW server logfile analysis program. URL: `www.lightside.net/analog/`
[15] A Stepanov, M. Lee, The Standard Template Library, ANSI Document No. X3J16/94–0030 WG21/N0417, March 1994.
[16] Object Space, Inc., STL<Tool Kit> users Guide , C++ Component Series, Version 1.1, May 1995.
[17] M. Reiser, A queueing netwrok analysis of computer communication networks with window flow control, IEEE Trans. Comm. (1979) 1201–1209.
[18] Object Management Group, Common Object Request Broker Architecture and Specification (CORBA), Technical Document 91.12.1, Revision 1.1.
[19] H. Lockhart, OSF DCE, McGraw-Hill, New York, 1994.

[20] J. Rolia, Predicting the performance of software systems, University of Toronto CSRI Technical Report 260, January 1992.
[21] F. Sheikh, J. Rolia, P. Garg, S. Frolund, A. Sheppherd, Layered performance modeling of a large scale distributed application, Proc. 1st World Congress on Systems Simulation, Quality of Service Modeling, September 1997, pp. 247–254.
[22] BGS System, Inc., Best/1 User's Guide, 1982.
[23] Quantitative System Performance, Inc., MAP User's Guide, 1982.
[24] P. Jacobson, E. Lazowska, Analyzing queueing networks with simultaneous resource possession, Comm. ACM (1982) 141–152.
[25] J. Dilley, R. Friedrich, T. Jin, J. Rolia, Measurement tools and modeling techniques for evaluating web server performance, in: R. Marie, B. Plateau, M. Calzarossa, G. Rubino (Eds.), Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, June 1997, St. Malo, France, Lecture Notes in Computer Science, vol. 1245, Springer, Berlin, pp. 155–168.

John Dilley is a research engineer with Hewlett-Packard Laboratories, Palo Alto, California, USA. He has worked at Hewlett-Packard for 12 years in various product development, advanced technology and research positions in the areas of networking and distributed computing. His research interests include architecture and design of distributed applications, object location and distributed directory services, web server performance analysis, and object-oriented design and development. He received degrees in Mathematics and in Computer Science from Purdue University. He is a member of the ACM and the IEEE Computer Society.

Rich Friedrich is a Senior Researcher at Hewlett-Packard Laboratories in Palo Alto, California, USA. He has held several research and product development positions within Hewlett-Packard including leading the system performance engineering team that developed and optimized the first commercial RISC based systems in the mid-1980s and the design of a distributed measurement system for the OSF DCE in the early 1990s. His current interests are in QoS control mechanisms for Internet services, distributed systems and the visualization of large data sets. He is the program co-chair for the IEEE Sixth International Workshop on Quality of Service. He attended Northwestern University and Stanford University.

Tai Jin is a research engineer at Hewlett-Packard Laboratories in Palo Alto, California, USA. He was a key contributor to the HP-UX networking projects in the late 1980s and was involved in the creation of the HP intranet. During that time he developed a tool which revolutionized network software distribution within the company. He has also created several useful services accessible through the World Wide Web. His interests include networked systems, exploiting the World Wide Web, performance tuning, creating useful tools, and the stock market. He graduated from Columbia University in 1984.

Jerome Rolia is an Associate Professor in the Department of Systems and Computer Engineering at Carleton University in Ottawa, Canada. His interests include software performance engineering, analytic modeling, and the measurement, model building, and management of distributed application systems. The work focuses on tools and methods to support the management of performance from a system's requirement stage through to maintenance. He received his Ph.D. from the University of Toronto in 1992.