Measurement Tools and Modeling Techniques for Evaluating Web Server Performance

John Dilley, Rich Friedrich, Tai Jin — Hewlett-Packard Laboratories, Palo Alto, CA, USA
Jerome Rolia — Carleton University, Ottawa, Ontario, Canada

ABSTRACT

The past few years have seen a rapid growth in the popularity of the Internet and the usage of the World Wide Web in particular. Thousands of companies are deploying Web servers and seeing their usage rates climb dramatically over time. Our research has focused on analyzing and evaluating the performance of Internet and intranet Web servers with a goal of creating a Layered Queueing Model to allow capacity planning and performance prediction of next generation server designs. 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 custom instrumentation we developed and deployed to collect workload metrics and model parameters from several large-scale, commercial Internet and intranet Web servers over a time interval of many months. We describe an object-oriented tool framework that significantly improves the productivity of analyzing the nearly 100 GBs of collected measurements. Finally, we describe the Layered Queueing Model 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.

1.0 INTRODUCTION

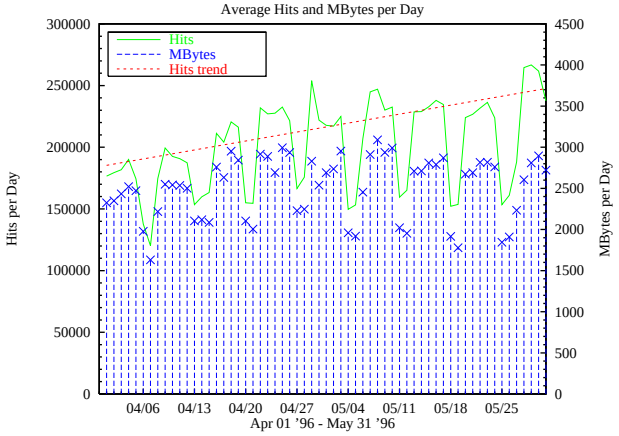
The World Wide Web [1] is a phenomenon which needs little introduction. Very briefly, the WWW employs a client/server architecture for access to a variety of information resources: a client **browser** application locates a server host and, using the HyperText Transfer Protocol (HTTP) sets up a network connection using TCP/IP with the HTTP **server** on that host. The client then requests one or many documents, images, 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 [2] is illustrated in Figure 1. It shows the periodicity of requests with the day of 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 37MB 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 parameterize an analytic model for capacity planning and experimental design analysis purposes.

Figure 1 Daily Traffic Trends



1.1 Related Work

Previous studies of World Wide Web performance have provided many useful insights into the behavior of the Web. At the University of Saskatchewan, Arlitt and Williamson [3] 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 [4]. One result of their work is a proposed set of workload invariants for Web traffic. We have corroborated several of these findings with our own data.

Cunha, Bestavros, and Crovella at Boston University also instrumented a Mosaic browser and studied the characteristics of the resulting client traces [5]. 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 disk requests (serving hot content directly out of memory).

Almeida, Bestavros, Crovella, and de Oliveira published a paper characterizing reference locality in the WWW [6]. 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 time scales.

Kwan, McGrath, and Reed at NCSA studied user access patterns to NCSA's Web server complex [7]. 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 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 several 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.

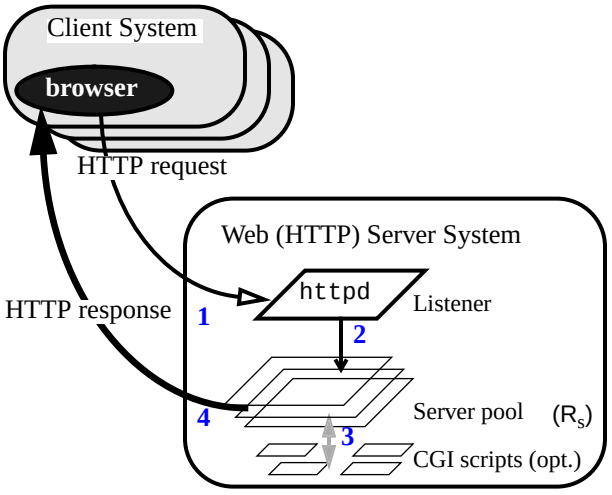
The workload characteristics of the Web [2][8] are fundamentally different from other well studied information systems, such as the Network File System (NFS) and On-Line Transaction Processing (OLTP) systems. In OLTP environments the median request and response message sizes tend to be on the order of 1KB; by contrast Web responses are substantially larger, on the order of 4KB, 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 servers must take into account the differences in workload and application architecture.

1.2 Web Server System Modeling

The architecture of a typical WWW service environment is shown in Figure 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 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

Figure 2 Processing an HTTP Request at a Web Server



`httpd` 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 of these processes execute 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 relationships between Web servers, clients, and the Internets and intranets that connect them.

Layered Queueing Models (LQM) have been proposed to study distributed application systems [9][10][11] and we extend and apply them in this paper to address Web servers. They are extended Queueing Network Models (QNM) [12] 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 of capacity planning questions as QNMs, but also estimate client queueing delays at servers. Figure 2 illustrates clients competing for access to a server. When the number of clients is large client queueing 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 World Wide Web performance evaluation and is structured as follows.

- Section 2.0 discusses the metrics and instrumentation required to capture response time and service demand at WWW server sites.
- Section 3.0 presents the performance analysis toolkit that supports rapid data reduction and visualization for large measurement logs.
- Section 4.0 describes Layered Network Queueing Models and presents an LQM for one of the sites we measured. The model defines the relationship between the number of `httpd` processes, the number of CPUs, network delays, and client response times.
- Section 5.0 summarizes our contributions and outlines areas of future research.

2.0 METRICS AND INSTRUMENTATION

This section describes the workload metrics 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.

- **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 queueing delays at physical resources in order to complete the request; it does not include queueing delays in the network or at the server prior to the request reaching the Server pool process. It consists of the following sub-components.
  - **$R_{S,Parse}$—Server parse time** is the time that the server spends reading the client request.
  - **$R_{S,Proc}$—Server processing time** is the time that the server spends processing the request.
  - **$R_{S,Net}$—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 queueing 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).
- **D_S—Server service demand** is the amount of system resources consumed by each client HTTP request. It consists of the following sub-components.
  - **$D_{s,CPU}$** is the average CPU time for the request.
  - **$D_{s,Disk}$** is the average disk demand for the request.
  - **$D_{s,Net}$** is the average network delay for the request.

R_S is distinct from the classical "response time" metric in several 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 actually 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` directives) 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 Res_C because R_S does not include client queueing 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 overestimates Res_C). Res_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 instrumentation we use LQMs and analytic performance evaluation techniques to estimate it based upon the measured values R_S.

The commercial sites we studied used a variety of Web server implementations from NCSA, OpenMarket, and Netscape. All of 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 $D_{s,CPU}$ 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 Figure 3, and show that R_S does not measure the network and queueing delays from the time the client made the request (the click of the hypertext link) and when the request arrived at the `httpd` process ($Q_{Net,Client}$ and Q_S in Figure 3). R_S also does not measure the network queueing and transmission delay for the last window of TCP data, which is queued in the server's network buffers after the final `write` system call completes ($Q_{Net,Server}$ 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.

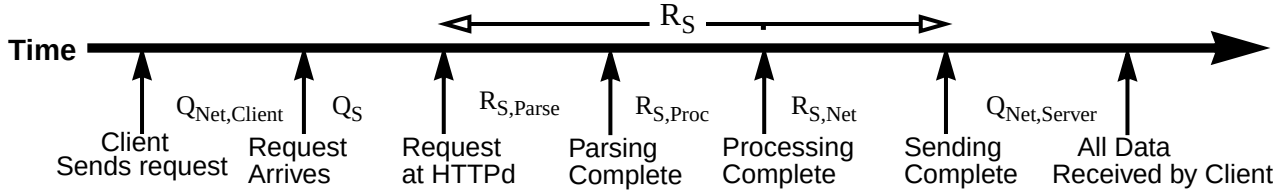
2.2 NCSA `httpd` Instrumentation

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

$D_{s,CPU}$ 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 milliseconds).

The instrumentation captures the three separate R_S intervals which when combined comprise the overall server response time, R_S. $R_{S,Parse}$ 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. $R_{S,Proc}$ 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. $R_{S,Net}$ begins with the sending of the response and ends with the final send of data from the `httpd` Server pool process. $D_{s,CPU}$ is measured over the same time period as R_S.

Figure 3 HTTP Request Timeline with Instrumentation Intervals



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 msec/msec/msec. 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 system 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.

3.0 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 two to twelve 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) [13][14][15][16] 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 int`, the URL 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 repeated 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 IO 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 [17][18]) **generic programming** paradigm. Each log record is viewed as an element in a **container** (log file) with an associated **iterator** type; an instance of the iterator points to each record in turn, starting with the element returned by the container's `begin` method, incrementing using the iterator's `operator++` method until reaching the container's `end` iterator 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 command 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 Figure 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.

Figure 4 Sample Data Reduction Application

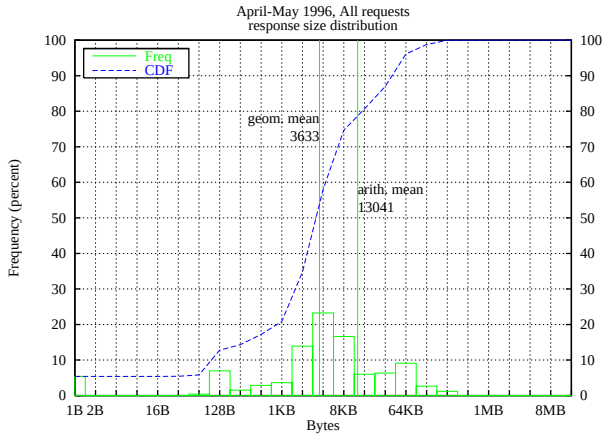
```
// Compute the response size distribution

#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;
}
```

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 if 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 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. An example of the output can be found in Figure 5.

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: total hits, bytes, and duration of the file
- distributions: of response sizes, response times, etc.
- summaries of traffic in hits and bytes by various time periods (hourly, daily, weekly)

Figure 5 Example Distribution



3.3 Visualization

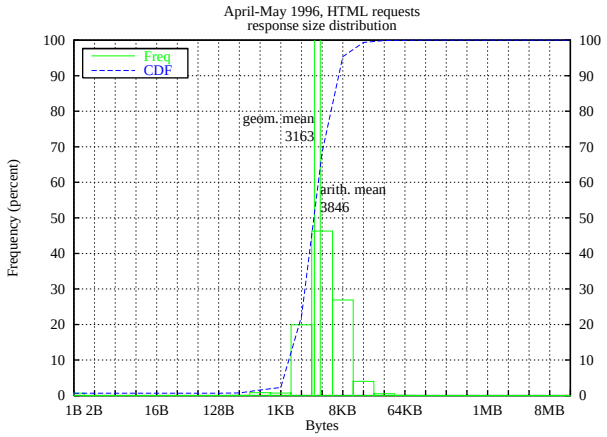
The output of the reduction tools is typically some form of tabular ASCII file, often summarized in the form of a frequency distribution. In order to make sense of this information a visualization tool sends the tabular output into `gnuplot` or some other graphics or spreadsheet tool that converts the raw data into more meaningful charts and graphs. The reduced data formats are sufficiently flexible that various graphics or spreadsheet packages can be used to allow flexibility for performance analysts. We tried using a number of the popular and powerful PC-based statistical analysis tools (SigmaPlot, Origin, Excel) but discovered that they cannot handle this reduced data: they insist upon doing the reduction and statistical analysis themselves. Unfortunately they cannot process the volume of data we possess and thus have not proved particularly useful.

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 4KB?", or, "What is the server response time of requests from the AOL domain?" To answer these questions 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. Figure 6 shows the file size distribution from the previous figure but filtered to only image requests.

Figure 6 Example Filtered Distribution



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 **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 seem to be coming through firewalls at large corporate sites or Internet Service Providers (ISPs).

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 tends 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.

4.0 LAYERED QUEUEING MODELS FOR 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 develop an analytic queueing model for the system. Analysis is chosen over simulation because of its speed and the easier interpretation of results. We rely on Mean Value Analysis (MVA [19][12]) based techniques and Queueing Network Models (QNMs) for our analysis. In this section we describe Layered Queueing Models (LQMs) and briefly outline the Method of Layers solution algorithm for LQMs. We then give an LQM for the Web server and identify its parameters. These parameters are gathered using the instrumentation and tools described in Section 2.0 and Section 3.0. Last, we use the model to predict client response times under different Web server and network configurations.

LQMs are 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 [20], DCE [21], and Web server applications. In these applications a process can suffer queueing 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) [9] and Stochastic Rendezvous Network (SRVN) [11] 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*:
  - the average number of visits $\mathbf{V_{c,s,k}}$ to each device *k*
  - the average service time $\mathbf{S_{c,s,k}}$ per request at each device *k*
  - and the average number of visits $\mathbf{V_{c,s,d,s2}}$ to each service *s2* of each of server process class *d*

Note that the client's service times at the services it visits are not specified. These values must be estimated by performance evaluation techniques.

Services identify separate visit ratio specifications that characterize the types of work supported by a server pool. A visit ration 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*:

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

Residence time expressions have been derived by MOL for LQMs modeling several types of software interactions, including synchronous RPC and rendezvous, multi-threaded servers [22] and asynchronous RPC [23]. As we will show next, the resulting abstraction can be used to describe Web server and other distributed applications.

4.1 An LQM for a Web Server

An LQM for the Web server is shown in Figure 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

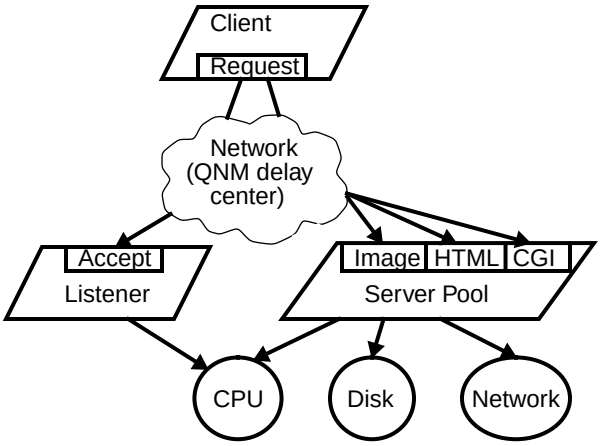
visits to all of the Web server's services. The Listener process has a single service named **accept** that accepts client requests and forwards them to the server pool. This is reflected in the LQM as a visit by the client class to the Listener process and then a visit to the server pool. With this approach we maintain the visit ratio and blocking relationships present in the real system.

The server pool offers three services that require significantly different resource demands. These are **Image**, **Html**, and **CGI**. 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 DCE or CORBA 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. 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 $R_{s,Net}$. 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 measurement tools were used to estimate the following LQM model parameters:

- for the **request** service of the client class *c*:
  - the arrival rate of requests $\lambda_{\mathbf{Client}}$
  - the average number of visits to Listener per request $\mathbf{V_{c,request,Listener,accept}} = 1$
  - the average number of visits to each Web service per request

Figure 7 Layered Queueing Model for a Web Server



  - $\mathbf{V_{c,request,Pool,Image}} = 57\%$
  - $\mathbf{V_{c,request,Pool,Html}} = 30\%$
  - $\mathbf{V_{c,request,Pool,Cgi}} = 13\%$
- $\mathbf{N_{Pool}}$ the number of processes in the server pool
- for each service *s* of the server pool class of processes:
  - $\mathbf{D_{s,CPU}}$ the average CPU time of each service
  - $\mathbf{D_{s,Disk}}$ the average disk demand of each service
  - $\mathbf{D_{s,Net}}$ the average network delay

For clients, the arrival rate of requests was estimated as the measured number of hits per measurement period. This is a low estimate, because requests will be dropped when it is not possible for a client to establish a connection. To emulate an open class we set the client population to a very large value with think time equal to the population size divided by the hit rate. 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 $D_{s,CPU}$ values were captured for each request then aggregated by service type. The $D_{s,Disk}$ could not be captured on a per service basis so they had to be estimated. Because of caching, the $D_{s,Disk}$ 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.

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

4.2 Server Capacity Planning

We now use the LQM to consider the performance impact of the server and network configuration on client response times at the server. The results of this modeling are shown in Figure 8. Client response times include queueing 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
  - wide area intranet (labeled in the figure as *Wintranet*): using a delay value of 1000 msec.
  - local area intranet (labeled as *Lintranet*): using a delay value of 10 msec.

- Server pool sizes of between 1 and 60.
- Either one or two CPUs on the server node, (labeled in the figure with the suffix of 1 or 2).

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 $R_{s,Net}$.

In Figure 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 Figure 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:

- 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.

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.

When the servers in the server pool approach full utilization, the estimated client response times increase very quickly. This is the case in Figure 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 Figure 9, the network delays were between 625ms and 14000ms for the different types of services (and hence result sizes). If result sizes were smaller, for example on the order of 1KB, we would not expect to see such a large impact on system behavior.

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.

We validated the model using the measurements collected and described in Section 2.0. 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 following average response times:

$R_{S,Image}$ = 13.0 sec
$R_{S,Html}$ = 1.0 sec
$R_{S,CGI}$ = 3.1 sec

Figure 8 Estimated Client and Server Response Times With Thread Pool Size of 60

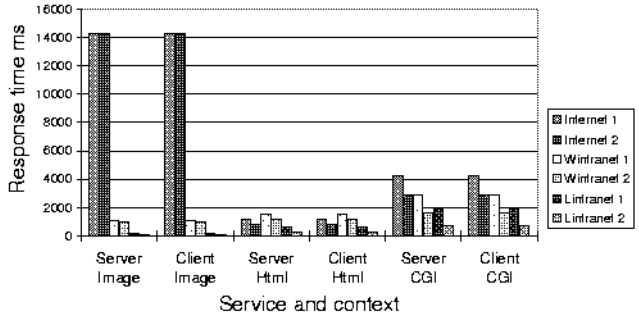
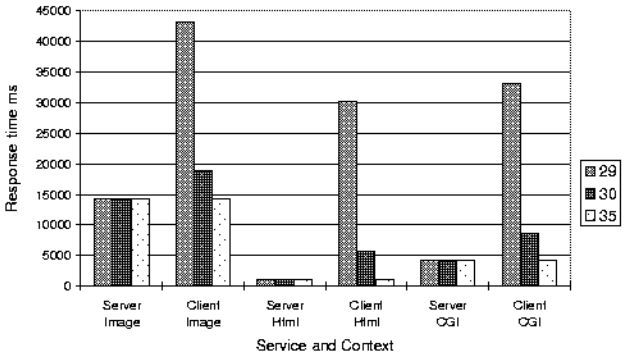


Figure 9 Internet Case With Measured Network Delay and Server Pool Sizes of 29, 30, and 35



These values agree closely with those predicted by the model as illustrated in Figure 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 parameterization.

5.0 Contributions and Future Work

This paper describes 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, commercial Internet and intranet Web servers over a time interval of many months. We developed an object-oriented tool framework that significantly improved the productivity of analyzing the 100s 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 a Layered Queueing Model. We used the model to predict client response times 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.

Our future research will focus on validating and extending this model beyond a single Web server; 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 modems and XDSL.

6.0 Acknowledgments

The authors would like to thank Sailesh Chutani, Gita Gopal, Gary Herman, and Jim Salehi for their review comments and feedback on this paper.

7.0 References

[1] *World-Wide Web: The Information Universe*. Tim Berners-Lee, Robert Cailliau, Jean-Frangois Groff, Bernd Pollermann. In Electronic Networking: Research, Applications and Policy, v2n1, pp 52-58. Meckler, Spring 1992.

[2] *Web Server Workload Characterization*. John Dilley. HPL TR 96-160, December 1996.

[3] *Web Server Workload Characterization: The Search for Invariants*. Martin Arlitt, Carey Williamson, University of Saskatchewan. In ACM SIGMETRICS, Philadelphia, May 1996.

[4] *A Synthetic Workload Model for Internet Mosaic Traffic*. Martin Arlitt, Carey Williamson, University of Saskatchewan.

[5] *Characteristics of WWW Client-based Traces*. Carlos Cunha, Azer Bestavros, Mark Crovella, Boston University. BU-TR-95-010. July 1995.

[6] *Characterizing Reference Locality in the WWW*. V. Almeida, A. Bestavros, M. Crovella, A. de Oliveira. In Proceedings of Parallel and Distributed Information Systems (PDIS), 1996.

[7] *User Access Patterns to NCSA's World Wide Web Server*. Thomas Kwan, Robert McGrath, Daniel Reed, University of Illinois. IEEE Computer, Vol 28, No. 11, November 1995.

[8] *Workload Characterization of Commercial Internet and Intranet Web Servers*. John Dilley, Richard Friedrich, Tai Jin, in preparation, 1996.

[9] *The Method of Layers*, J. Rolia and K. Sevcik, IEEE Transactions on Software Engineering, Vol 21, No. 8, pp 689-700, August 1995.

[10] *A Toolset for Performance Engineering and Software Design of Client-Server Systems*, G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, C.M. Woodside, Special Issue of the Performance Evaluation Journal, Volume 24, Number 1-2, Pages 117-135.

[11] *The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*. IEEE Transactions on Computers, C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar, Volume 44, Number 1, pages 20-34, January 1995.

[12] Q*uantitative System Performance: Computer System Analysis Using Queueing Network Models*. E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, Prentice-Hall, 1984.

[13] *getstats: Web server log analysis utility.* URL: `www.eit.com/software/getstats/getstats.html`

[14] *wwwstat: HTTPd Logfile Analysis Software*. URL: `www.ics.uci.edu/pub/websoft/wwwstat/`

[15] *Wusage 4.1: A Usage Statistics System For Web Servers*. URL: `www.boutell.com/wusage/`

[16] *Analog: a WWW server logfile analysis program*. URL: `www.lightside.net/analog/`

[17] *The Standard Template Library.* Alexander Stepanov, Meng Lee, Hewlett Packard. ANSI Document No. X3J16/94-0030 WG21/N0417. March 1994.

[18] *STL<ToolKit> Users Guide*. ObjectSpace, Inc. C++ Component Series. Version 1.1, May 1995.

[19] *A Queueing Network Analysis of Computer Communication Networks with Window Flow Control*. M. Reiser. IEEE Transactions On Communications, August 1979, pp. 1201-1209.

[20] *Common Object Request Broker Architecture and Specification (CORBA)*. Object Management Group Document Number 91.12.1, Revision 1.1.

[21] *OSF DCE*. Hal Lockhart. McGraw Hill, 1994. ISBN 0-07-911481-4.

[22] *Predicting the Performance of Software Systems*. CSRI Technical report 260, J. Rolia, University of Toronto, Canada, January 1992.

[23] L*ayered Performance Modelling of a Large Scale Distributed Application*. F. Sheikh, J. Rolia, S. Frolund, P. Garg, and A. Shepherd, In preparation to be submitted to SIGMETRICS ‘97.