System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace

by

Steven D. Gribble

B.Sc. (University of British Columbia) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Eric A. Brewer, Chair
Professor Randy H. Katz

1997

The dissertation of Steven D. Gribble is approved:

Chair Date

Date

University of California at Berkeley

1997

System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace

by

Steven D. Gribble

Abstract

System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace

by

Steven D. Gribble

Master of Science in Computer Science

University of California at Berkeley

Professor Eric A. Brewer, Chair

In this thesis, we present the analysis of a large client-side web trace gathered from the Home IP service at the University of California at Berkeley. Specifically, we demonstrate the heterogeneity of web clients, the existence of a strong and very predictable diurnal cycle in the clients' web activity, the burstiness of clients' requests at small time scales (but not large time scales, implying a lack of self-similarity), the presence of locality of reference in the clients' requests that is a strong function of the client population size, and the high latency that services encounter when delivering data to clients, implying that services will need to maintain a very large number of simultaneously active requests. We then present system design issues for Internet middleware services that were drawn both from our trace analysis and our implementation experience of the TranSend transformation proxy. We also describe two generations of a trace playback tool that allowed us to replay the Home IP traces back at our prototype middleware service for the sake of stress testing.

To that which makes life

come to life.

Contents

List of Figures

List of Tables

Acknowledgements

I want to thank my colleagues, specifically Armando Fox, Ian Goldberg, David Wagner, and Yatin Chawathe, who have made the past two years such a great experience. I'd also like to thank Randy Katz for his feedback on earlier drafts of this Master's report, and Cliff Frost and Vern Paxson for their feedback and help during the tracing process. Finally, I'd like to thank my supervisor, Eric Brewer, who has given me his unwavering support, vision, and trust.

1 Introduction

The recent explosive growth of the Internet [32] has resulted in a similar explosion of web-oriented middleware services ([15], [3], [6]). These middleware services, particularly the more popular services that experience extremely high load, must overcome a number of challenging system design issues in order to maintain fast response time, constant availability, and capacity. Services must accommodate a client population with wide variation in hardware, software, and network connectivity. They must be able to handle offered loads of hundreds of requests per second, and because of the often slow connectivity to clients and the implied lengthy delivery times, they must be able to handle hundreds of simultaneously outstanding tasks.

Previous work has explored the performance of operating system primitives and their relationship to architecture ([30], [2]), and operating system design issues for busy Internet services ([19], [27]). In contrast, this paper raises a number of system design issues specifically for Internet middleware services that we encountered during two separate but related efforts: the analysis of a set of extensive client-side HTTP [5] traces that we gathered from the University of California at Berkeley's dial-in modem banks during October and November of 1996, and the implementation and deployment experience we gained from the TranSend Internet middleware service [15].

Since nearly 70% of all Internet clients use dial-in modems of speeds of 28.8 Kb/s or less [18], we use the traces to make a number of observations about the Internet user population and the services with which they communicate. Section 2 discusses the gathering of the traces, and section 3 performs a detailed analysis of these traces, both in terms of observations made about the client population and the services themselves. In section 4, we discuss the system design issues drawn from our experience with TranSend, and in section 5 we discuss a tool that we designed to replay the captured traces at a live proxy. In section 6 we present related work, and in section 7 we conclude.

2 Home IP Trace Gathering

During October and November of 1996, we gathered over 45 consecutive days worth of HTTP traces from the Home IP service offered by UC Berkeley to its students, faculty, and staff available to researchers. (Two and a half weeks worth of anonymized versions of these traces have been made available at http://www.acm.org/ita.) Home IP provides dial-up PPP/SLIP connectivity using 2.4 kb/s, 9.6 kb/s, 14.4 kb/s, or 28.8 kb/s wireline modems, or Metricom Ricochet wireless modems, which achieve approximately 20-30 kb/s throughput with a 500 ms round-trip time (RTT).

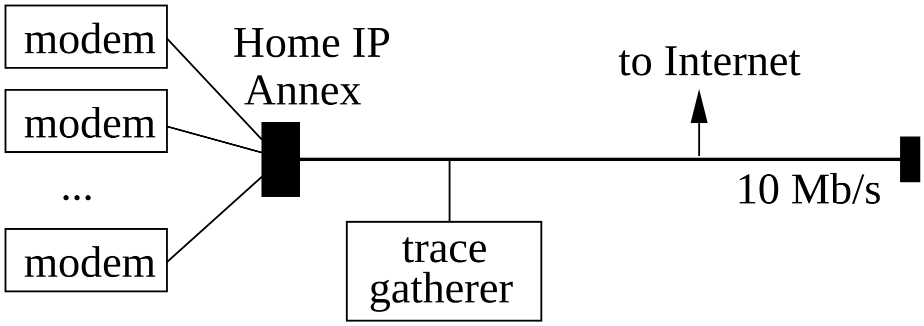

Figure 1: **The Home IP Tracing Environment**

2.1 IPSE

The HTTP client traces were unobtrusively gathered through the use of a packet sniffing machine placed on a 10 Mb/s Ethernet segment at the head-end of the Home IP modem bank through which all IP traffic flows (Figure 1). The trace gathering program that we used was a custom HTTP module written on top of the Internet Protocol Scanning Engine (IPSE)[17]. IPSE is a user-level packet filter that runs on Linux; IPSE allows filter modules to capture TCP segments and recreate the TCP streams observed by the endpoints of the TCP connection. The custom module was therefore able to parse each HTTP request as it was happening, and write out the salient features of each HTTP request to a log file on-the-fly. We only traced traffic destined for port 80, ignoring all non-HTTP protocols and HTTP connections to other ports. Each user of the Home IP service is assigned a static IP address, so we could track individual users over the entire duration of the tracing experiment.

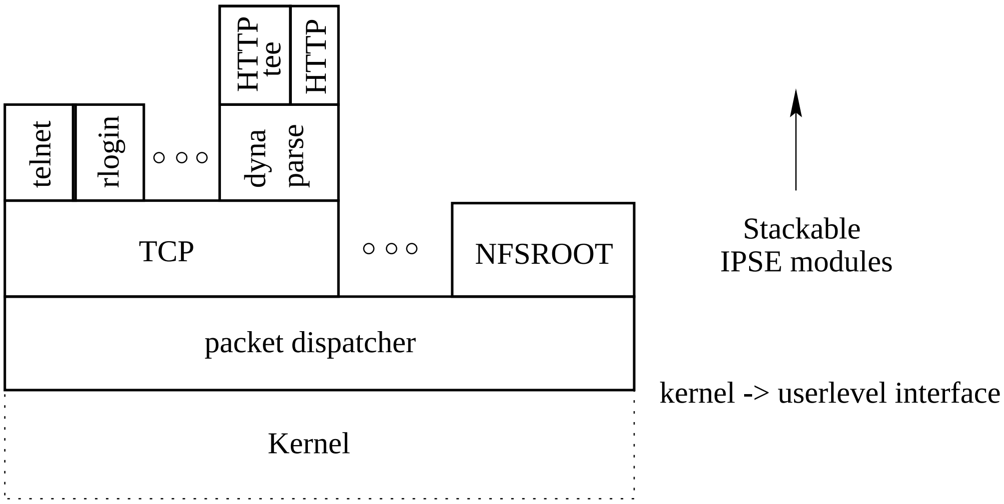

Figure 2: **The Internet Protocol Scanning Engine (IPSE)**

Figure 2 illustrates the design of IPSE. The lowest layer of IPSE, the packet dispatcher, accepts link-level packets either directly from the kernel[1] or through the LIBPCAP [29] library. In the first case, packet filtering is performed at the user level, and in the second case in-kernel filtering is performed if available.

The packet dispatcher sends these packets to the appropriate IPSE module, which is loaded dynamically when IPSE starts up. One such module, the generic TCP module, maintains state for each active TCP connection, and multiplexes the TCP data up to a higher level set of TCP modules. Because IPSE was initially designed to be run only for short sessions, the state maintenance and lookup mechanism relied on a linked-list of TCP

[1]Using the Linux SOCK_PACKET interface.

connections that were never purged. As we needed to run IPSE for hours at a time on a network with hundreds of simultaneously active connections, we rewrote this portion of IPSE to use a hash table instead of linked lists, and to throw away connection state information when either the TCP connection was properly shut down (through FIN or RST packets), or if the TCP connection remained unused for long periods of time.

We implemented an additional TCP module, *dynaparse*, which reconstructs TCP streams and allows a third level of module to perform data parsing operations on these recontructed streams. Our HTTP module is an example of such a dynaparse-dependent module: it extracts critical information from HTTP sessions, and writes this information out to a trace file. The HTTP module is able to handle both persistent and non-persistent HTTP connections. All protocol information is gathered through on-the-fly protocol analysis: the HTTP headers are parsed as the TCP segments arrive and are appended to the TCP stream reconstruction. Only the desired information is saved to the trace file, saving both time and disk space. Except for anonymization, no postprocessing was performed on these traces.

We implemented a second dynaparse-dependent module called HTTP-tee. HTTP-tee encapsulates the client request HTTP headers in a UDP packet, and sends this packet to remote process that extracts the HTTP headers and uses them to send an HTTP request to either the end server or to a proxy. HTTP-tee thus transparently forks off a stream of HTTP requests, allowing forked stream to be directed at a target that will then receive an identical series of requests as was observed on the tracing network.

2.2 The Trace Files

The 45-day trace contains approximately 24,000,000 HTTP requests, representing the web-surfing behaviour of over 8,000 unique clients.[2] The trace capture tool collected the following information for each HTTP request seen:

[2]Although there are approximately 20,000 registered users of the Home IP service, only 8,000 of them actually used the service during the tracing period.

- the time at which the client made the request, the time that the first byte of the server response was seen, and the time that the last byte of the server response was seen,
- the client and server IP addresses and ports,
- the values of the **no-cache**, **keep-alive**, **cache-control**, **if-modified-since**, **useragent**, and **unless** client headers (if present),
- the values of the **no-cache**, **cache-control**, **expires**, and **last-modified** server headers (if present),
- the length of the response HTTP header and response data, and
- the request URL.

For example, the following is an ASCII representation of one of the unanonymized trace entries:

```
862265301:723492 862265301:729372 862265301:730384 136.152.117.53:1040
128.32.25.12:80 10 0 855810957 4294967295 4294967295 80 1 43
GET /assets/images/departments.gif HTTP/1.0
```

Here, we see that client IP address 136.152.117.53 contacted the server 128.32.25.12 to request the URL "/assets/images/departments.gif". The request was made at epoch time 862265301.7 (Monday, April 28th, 1997 at 3:08:21pm), the first byte of the server response was seen 6ms later, and the last byte of the server response was seen 1ms after that. The response consisted of 80 bytes of header and 1 byte of data. This probably means that the response was a redirect or error, and the data simply contained a single newline character.

The HTTP module wrote this information to disk in a compact, binary form. Every four hours, IPSE was shut down and restarted, as its memory image would get extremely large over time.[3] This implies that there are two potential weaknesses in these traces:

1. Any connection active when the engine was brought down will have a possibly incorrect timestamp for the last byte seen from the server, and a possibly incorrect reported size.

2. Any connection that was forged in the very small time window (about 300 milliseconds) between when the engine was shut down and restarted will not appear in the logs.

We estimate that no more than 150 such entries (out of roughly 90,000-100,000) are misreported for each 4 hour period.

Once per day (late at night to avoid interfering with the Home IP traffic), these 4 hour trace files were bulk transferred from the tracing machine to another post-processing machine, on which trace anonymization was performed. Anonymization consisted of replacing the source IP, destination IP, and requested URL with keyed hashes. Because hashes are one-way functions, it is difficult for anyone to recover IP addresses, but since hashes are deterministic and probabilistically unique, they still maintain enough information to distinguish individual sites, users, and URLs. We also preserved filename extensions in URLs in order to deduce data type information.

David Wagner [33] pointed out that if either the client or server population space is small, or if the hash digest size is small, then the hash function could be reversed through an exhaustive search. Also, the preserved filename extensions in our anonymized traces allow an antagonist to easily find correlations between requests in the traces and requests observed in publically available HTTP server log files, giving her an opportunity to perform

[3] We believe there was a memory leak that was either in IPSE, or in the kernel and was tickled by IPSE. This leak couldn't be resolved even after many days of code inspection, so we chose to work around the bug by periodically restarting IPSE.

traffic analysis based on timing coincedences between server log entries and our Home IP trace entries and therby identify servers and users in the Home IP traces.

3 Trace Analysis

In this section, we present the results of our analysis of the Home IP traces. In section 3.1, we demonstrate the heterogeneity of the observed client population. Section 3.2 discusses the request rates and interarrival times generated by the client population. In 3.3, object type and size distributions are presented. Section 3.4 demonstrates the existence of locality of reference within the traces through the use of a number of cache simulations driven by trace entries. Finally, section 3.5 presents distributions of service response times, and argues that at any given time, a very large number of outstanding requests will be flowing through middleware or end services.

3.1 Client Heterogeneity

Table 1 lists the most frequently observed "UserAgent" HTTP headers observed within the Home IP traces. From this table, it is easy to make a common misconclusion about web clients, namely that the set of web clients in use is extremely homogeneous, as nearly all browsers observed in our traces are either the Netscape Navigator [28] or Microsoft Internet Explorer (MSIE) [26] browsers running on the Windows or Macintosh operating systems. However, there is significant heterogeneity arising from the many versions of these browsers and their widely varying feature sets. Furthermore, we observed a total 166 different UserAgent values within the traces, representing a wide range of desktop systems (MacOS, Win16, NetBSD, Linux, etc.) More significantly, however, we saw requests from a number of exotic clients such as Newton PDAs running the NetHopper [1] browser.

Internet services that do not want to limit the effective audience of their content must therefore be able to deliver content that suits the needs of all of these diverse clients. Either the services themselves must adapt their content, or they must rely on the emergence of middleware services (such as in [13], [14], and [7]) to adapt content on the fly to better suit the clients' particular needs.

Browser	OS	% Seen
Netscape	Windows 95	55.1
	Macintosh	19.7
	Windows (other)	8.8
	Windows NT	3.5
	Linux	2.2
	Other	0.4
MSIE	Windows 95	7.6
	Macintosh	0.6
	Windows NT	0.7
	Windows (other)	0.1
Other		1.3

Table 1: **UserAgent HTTP headers:** this table lists the 10 most frequent UserAgent headers observed in the traces. "Other" browsers observed include PointCast, Cyberdog, Mosaic, Opera, Lynx, JDK, and NetHopper.

3.2 Client Activity

As seen in Figure 3, the amount of activity seen from the client population is strongly dependent on the time of day. The Berkeley web users were most active between 8:00pm and 2:00am, with nearly no activity seen at 7:00am. Services that receive requests from local users can thus expect to have widely varying load throughout the day; internationally used services will most probably see less of a strong diurnal cycle. Other details can be extracted from these graphs. For example, there is a decrease of activity at noon and at 7:00pm, presumably due to lunch breaks and dinner breaks, respectively.

The diurnal cycle is largely independent of the day of the week, but there are some minor differences: for instance, on Fridays and Saturdays, the traffic peaks are slightly higher than during the rest of the week. However, the gross details of the traces remain independent of the day of the week. We calculated the average daily cycle observed by averaging the number of events seen per minute for each minute of the day across 15 days of traffic. For our calculation, we picked days during which there were no anomalous trace effects, such as network outages. Figure 4 shows this average cycle, including a polynomial curve fit that can be used to calculate approximate load throughout a typical day.

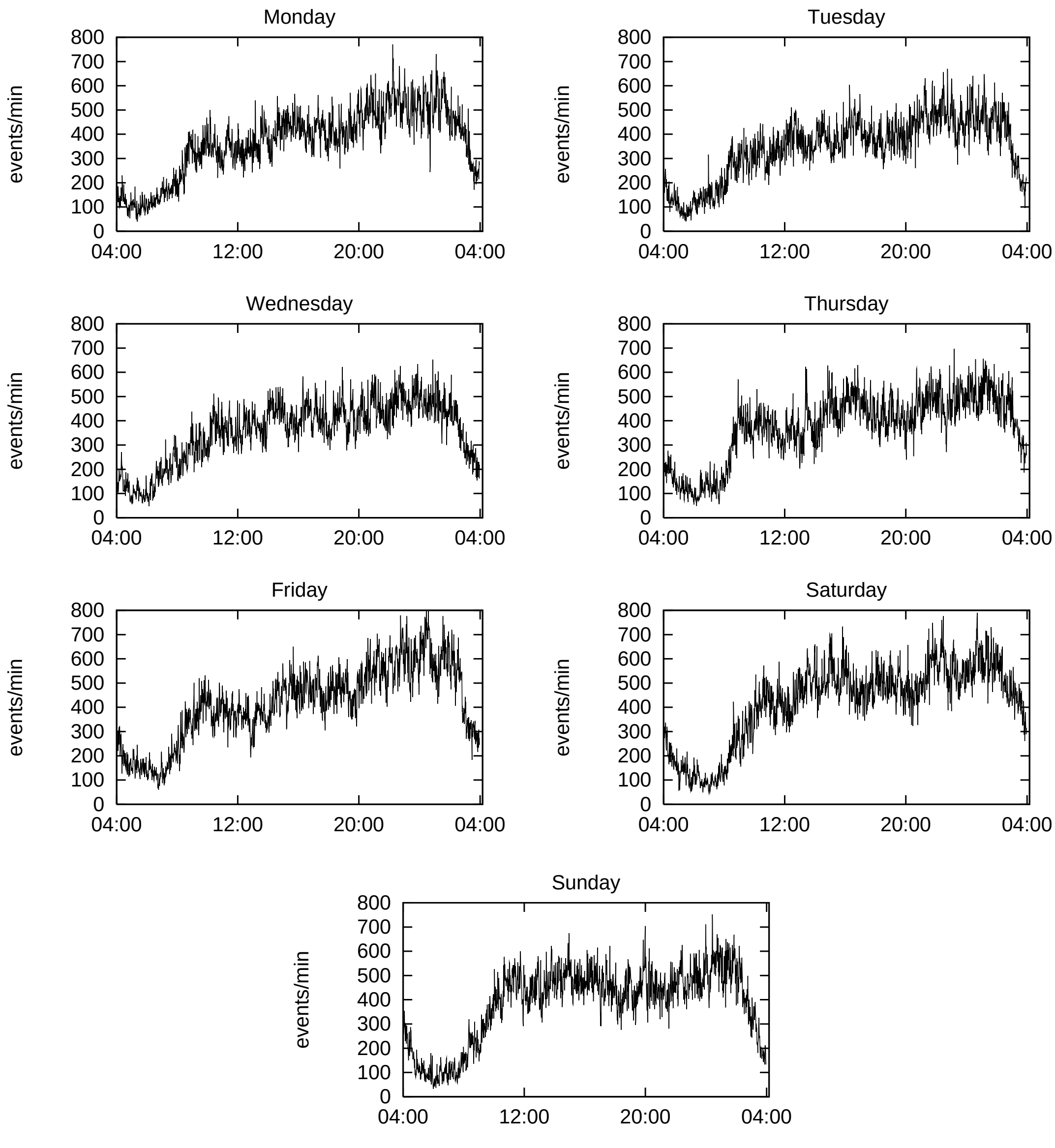

Figure 3: **Diurnal cycle** observed within the traces - each graph shows 1 day worth of trace events. The y-axis shows the number of observed requests per minute.

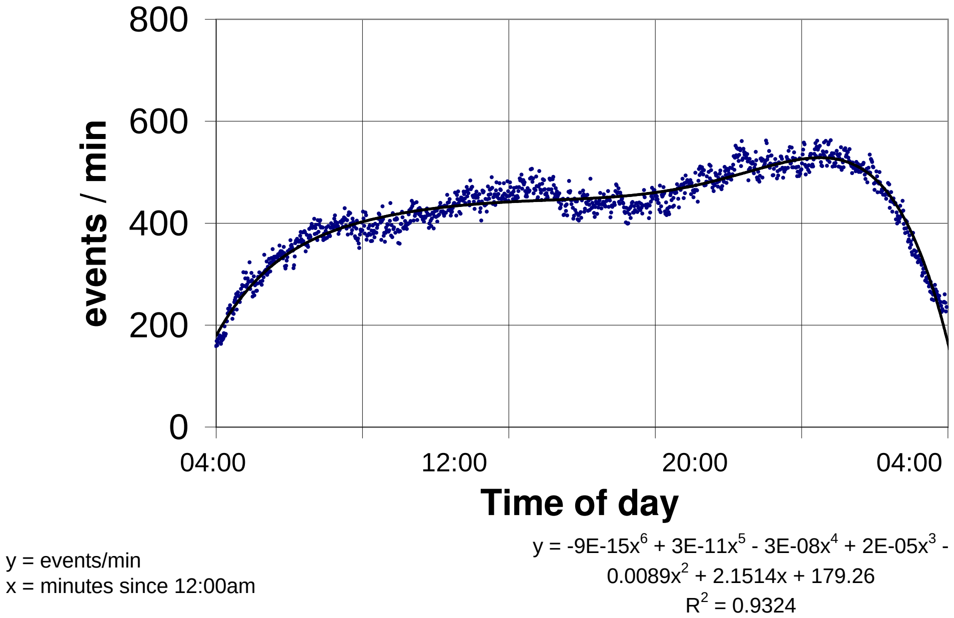

Figure 4: **Average diurnal cycle** observed within the traces - each minutes worth of activity shown is the average across 15 days worth of trace events. The x-axis is broken down into minute-long buckets, beginning and ending at 12:00am. The y-axis shows the average number of observed requests per minute for each minute-long bucket.

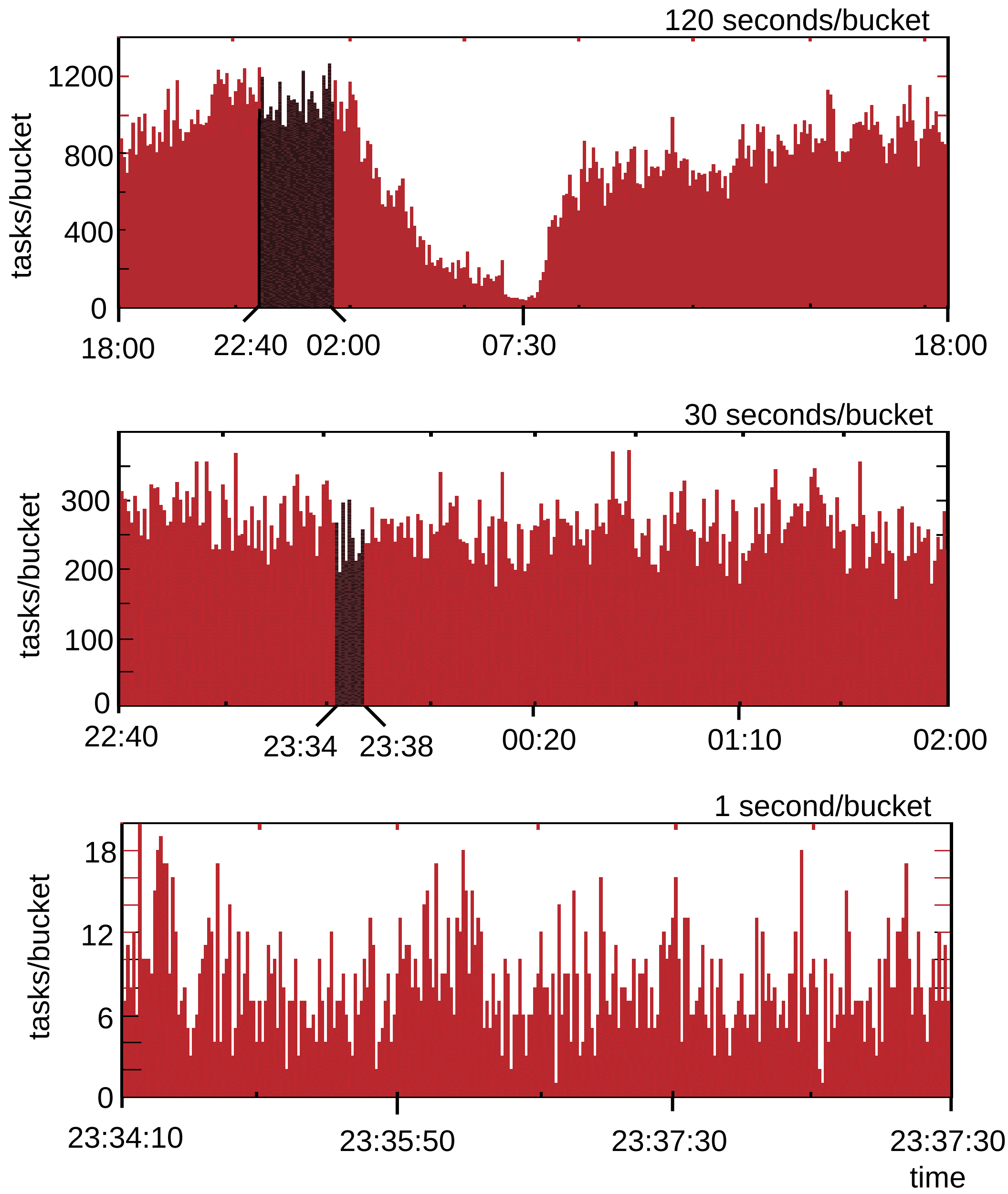

Figure 5: **Request rate** observed over a 24 hour, 3 hour, and 3 minute period of the traces.

On shorter time scales, we observed that client activity was less regular. Figure 5 illustrates the observed request rate at three time scales from a one-day segment of the traces. At the daily and hourly time scales, traffic is relatively smooth and predictable - no large bursts of activity are present. At the scale of tens of seconds, very pronounced bursts of activity can be seen; peak to average ratios of more than 5:1 are common.

Many studies have explored the self-similarity of network traffic ([4], [16], [21], [22], [24], [31]), including web traffic [9]. Self-similarity implies burstiness at all timescale, which is **not** compatible with our observations. One indicator of self-similarity is a heavy-tailed interarrival process. As Figure 6 clearly shows, the interarrival time of GIF requests seen within the traces is exponentially distributed, and therefore not heavy tailed. We saw similar exponential distributions for other data types' request processes, as well as for the aggregate request traffic. These observations correspond to requests generated from a large population of independent users.

Internet services must be able to handle rapidly varying and bursty load on fine time scales (on the order of seconds), but these bursts tend to smooth themselves out on larger time scales (on the order of minutes, hours, or days). The provisioning of resources for services is therefore somewhat simplified and more predictable. This is extremely important, because capacity planning for services is difficult but must be accurate to adequately support a user population.

3.3 Reference type and size distributions

Section 3.1 answered the question of who requests data, and section 3.2 discussed how often the data gets requested. In this section, we inspect the nature of the data that is requested. Figure 7a shows the MIME type breakdown of the transferred data in terms the number of bytes transferred, 7b shows this breakdown in term of files transferred.

From Figure 7a, we see that most of the bytes transferred over the Home IP

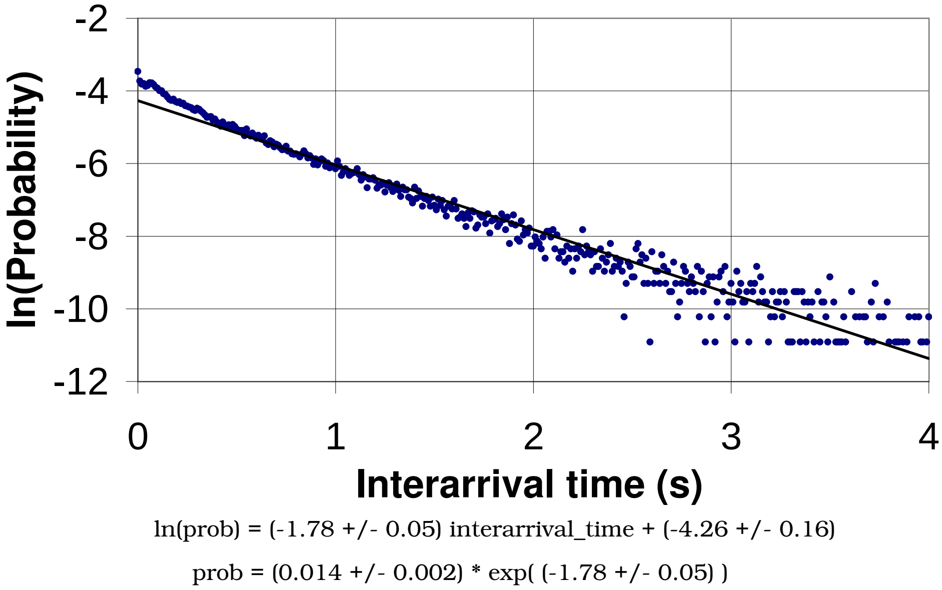

Figure 6: **Interarrival time distribution** for GIF data type requests seen within a day-long trace portion. Note that the Y-axis is on a logarithmic scale. The upper of the two curve fit equations represents the least-squares linear fit in the logarithmic domain, and the bottom of the two equations represents the exponential curve in the linear domain. The error bound on the exponential multiplicative constant was pessimistically calculated as the larger of the intercept error bounds from the logarithmic domain converted to the linear domain.

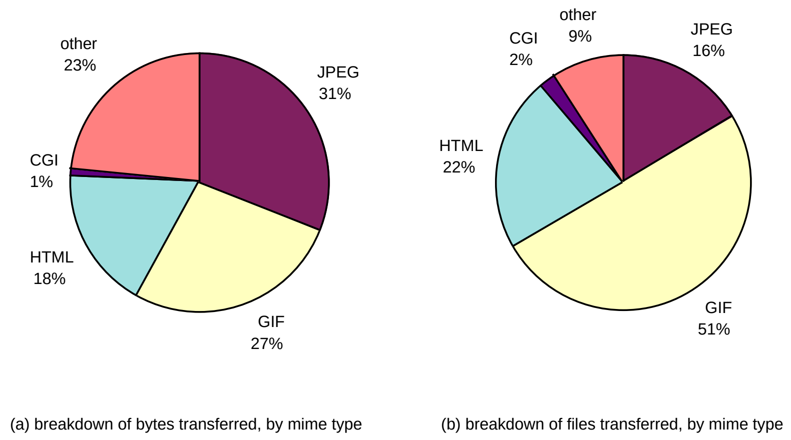

Figure 7: **Breakdown of bytes and files transferred by MIME type**

modem lines come from three predominant MIME types: **text/html**, **image/gif**, and **image/jpeg**. Similarly, Figure 7b shows that most files sent over the modem lines have the same three predominant MIME types. Interestingly, however, we see that although most bytes transferred correspond to JPEG images, most files transferred correspond to GIF images. This means that, on average, JPEGs are larger than GIFs.

The fact that nearly 58% of bytes transferred and 67% of files transferred are images is good news for Internet cache infrastructure proponents. Image content tends to change less often than HTML content; images are usually statically created and have long periods of stability between modifications, in comparison to HTML which is often dynamically generated. There are counterexamples, such as dynamic images generated by devices like “web cams”, but these are very rare relative to static images.

In Figure 8, we see the distribution of sizes of files belonging to the three most common MIME types. Two observations stand out: most Internet content is less than 10 kilobytes in size, and data type size distributions are heavy-tailed, meaning that there is a non-trivial number of large data files on the web. Looking more closely at individual distributions, we can confirm our previous hypothesis that JPEG files tend to be larger

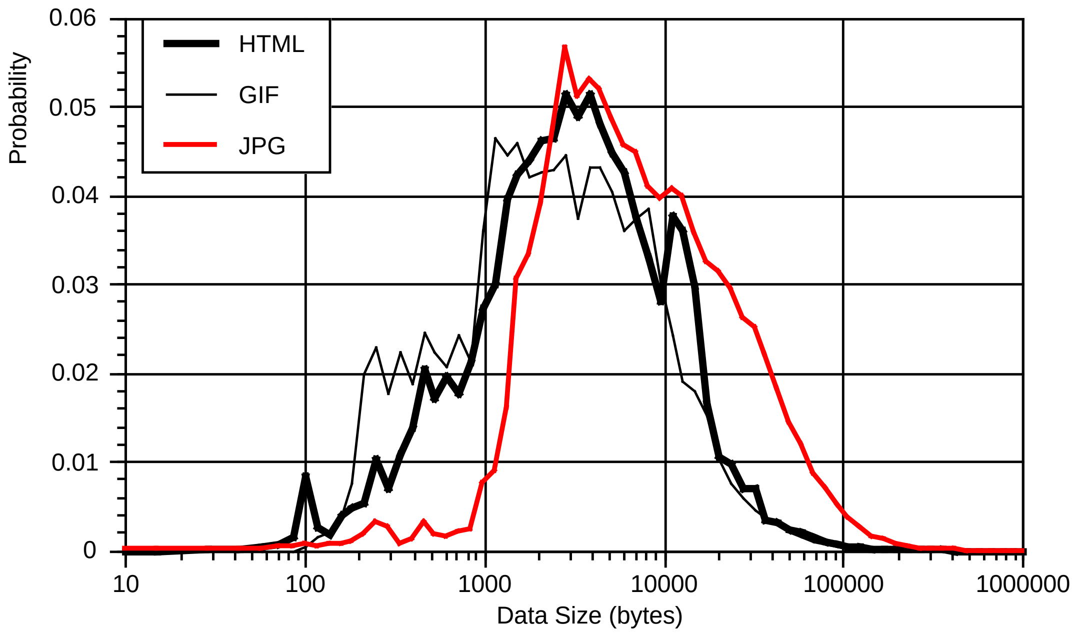

Figure 8: **Size distributions by MIME type**, shown on a logarithmic scale. The average HTML file size is 5.6 kilobytes, the average GIF file size is 4.1 kilobytes, and the average JPEG file size is 12.8 kilobytes.

than GIF files. Also, the JPEG file size distribution is considerably more heavy-tailed than the GIF distribution. There are more large JPEGs than GIFs, perhaps in part because JPEGs tend to be photographic images, and GIFs tend to be cartoons, line art, or other such simple, small images.

There are other anomalies in these distributions. The GIF distribution has two visible plateaus, one at roughly 300-1000 bytes, and another at 1000-5000 bytes. We hypothesize that the 300-1000 byte plateau is caused by small "bullet" images or icons on web pages, and the 1000-5000 byte plateau represents all other GIF content, such as cartoons, pictures, diagrams, advertisements, etc. Another anomaly is the large spike in the HTML distribution at roughly 11 kilobytes. Investigation revealed that this spike is caused by the extremely popular Netscape Corporation "Net Search" page.

3.4 Locality of Reference

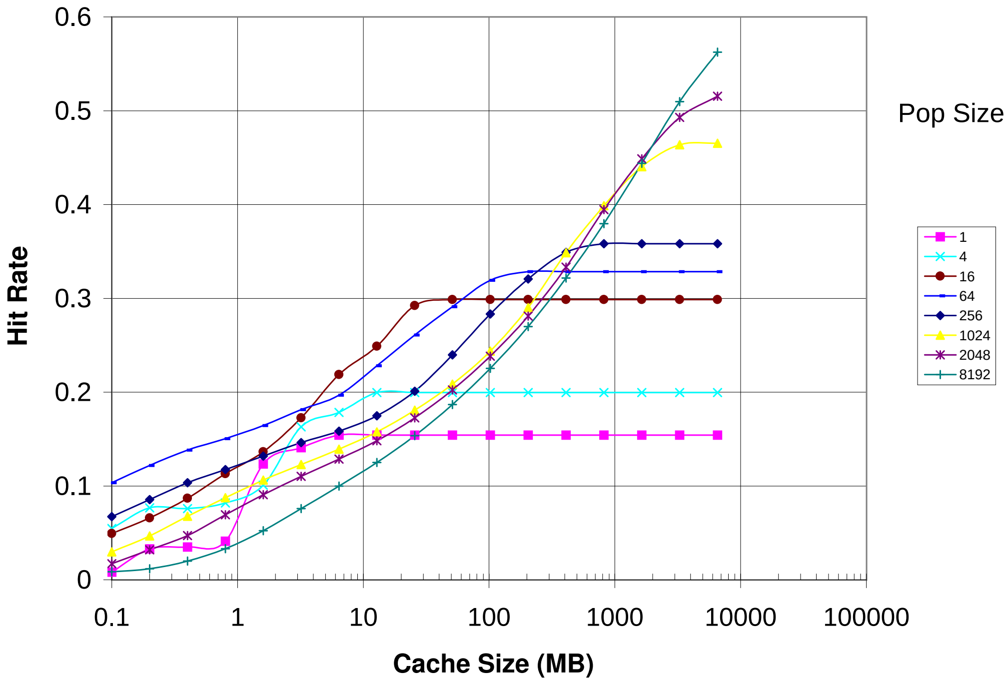

Figure 9: **Hit rate vs. Cache size** for a number of different user population sizes.

A near-universal assumption in systems is that of locality of reference, and the typical mechanism used to take advantage of this locality of reference is caching [11, 8]. The effectiveness of caching depends upon a number of factors, including the size of the user population that the cache serves and the size of the cache.

To measure the effectiveness of infrastructure caching (as opposed to client-side or server-side caching) with respect to the HTTP references captured from the Home IP population, we implemented a cache simulator and played segments of the traces at these

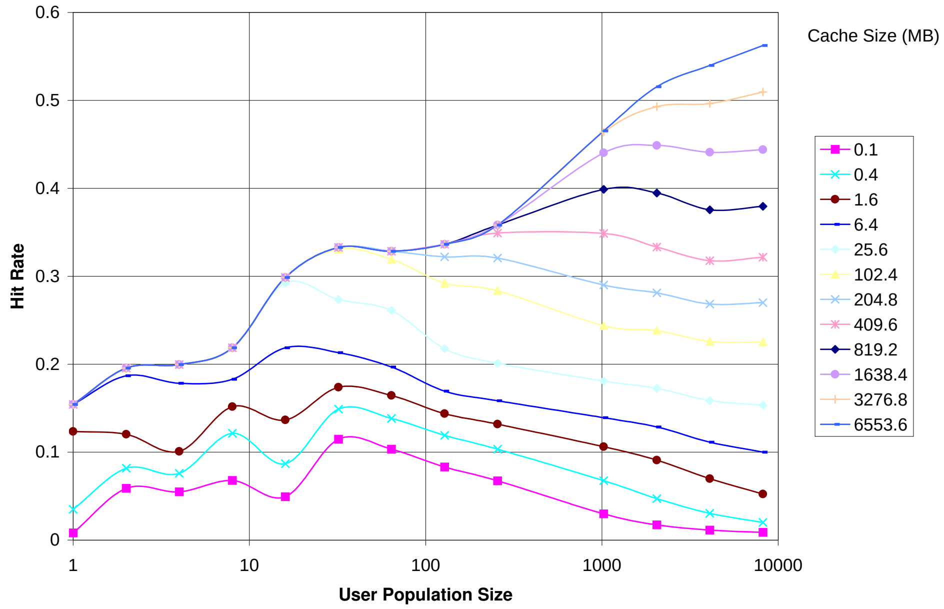

Figure 10: **Hit rate vs. User Population size** for a number of cache sizes.

caches. We filtered out requests from all but a parameterizable set of client IP addresses in order to simulate client populations of different sizes. The cache simulator obeyed all HTTP cache pragmas (such as the no-cache pragma, the if-modified-since header, and the expiry header), and implemented a simple LRU eviction policy. Figure 9 shows measured cache hit rate as a function of cache size for different user population sizes, and Figure 10 shows measured hit rate as a function of user population size for different cache sizes.

Figure 9 shows two trends: the first is that an increasingly large cache size results in an increasingly large cache hit rate. The second trend is that we observed that hit rate is a very strong function of the user population size. As the population gets larger, the locality of reference within that population gets stronger, and caches become more effective. For a given population size, the cache hit rate as a function cache size plateaus at the working set size of that population. In Figure 10, one additional trend can be observed: as the user population size grows, if the cache size does not also pace the increasingly large working set of that population, the cache hit rate will start to drop as the cache effectively begins to thrash from constant evictions.

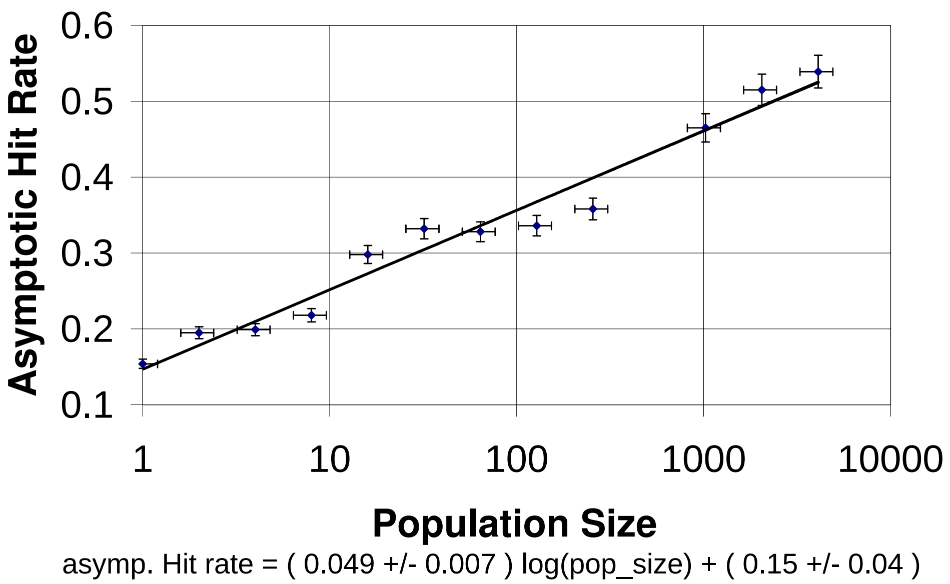

Figure 11: **Asymptotic Hit Rate vs. User Population Size**

An interesting question is: what is the maximum possible cache performance for a given user population size? In Figure 11, we have plotted the asymptotic hit rate achieved in the limit of infinitely large cache size as a function of the user population size. In other words, this graph explicitly shows the cachable working set size of a given user population size. We see that for the range of population sizes that we can model from our traces, the asymptotic hit rate grows logarithmically with population size. Obviously, this logarithmic

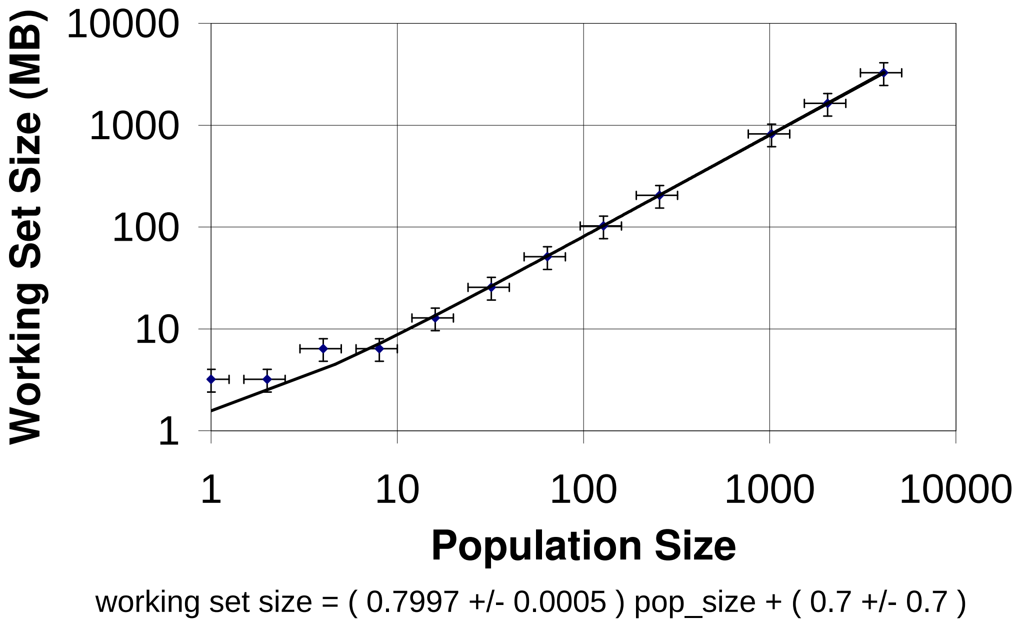

Figure 12: **User Population Size vs. Working Set Size**

increase cannot continue forever, as there is a maximum possible hit rate of 100%; unfortunately, our traces do not contain a large enough population size to see the logarithmic increase tail off.

Another interesting question is of the working set size of a given user population. In Figure 12, we illustrate the working set size size (in MB) as a function of the user population size. We define the working set size as the cache size that gives at least a 0.95 hit rate. As we can see, the working set size is a strongly linear function of the population set size. This implies that each user on balance introduces a new corpus of documents into the total population working set. Figure 12 shows us that each user contributes 0.80 MB to the total working set; given an estimated average 30KB document size (this is HTML plus 5 inline images), we see that each user contributes about 25 documents to the total working set of the population.

A factor that can alter the performance of Internet caches is the increasingly prevalent use of cache pragmas in HTTP headers. To investigate this effect, we measured the percentage of HTTP client requests and server responses that contained relevant headers, namely:

no-cache: This header can be supplied by either the client or the server, and indicates that the requested or returned data may not be served out of or stored in a client-side or proxy cache.

cache-control This is a generic, extensible HTTP header whose value contains the real directive. Cache-control is intended to be used to supply additional caching directives that are interpreted by middleware caches, rather than by the end server or client.

if-modified-since This HTTP header allows a client to specify that a document should be returned only if it has been modified after a certain date. If it hasn't, then the client uses a locally cached version.

expires This HTTP header allows a server to supply an expiration date for returned content. Caches obey this directive by treating cached data as stale if the expiration date has occurred.

last-modified This HTTP header allows a server to indicate when a document has last been modified. This is typically used as a hint for caches when calculating time-to-live (TTL) values, or when returning HTTP headers in response to a client's HEAD request.

As can be seen in table 2, most HTTP headers that can affect cache performance are rarely used. The most frequently used header is the last-modified server response header; this header is now commonly returned by default from most HTTP servers. The presence

Pragma occurrence	11/8/96	4/28/97
(C) no-cache	7.21 ± 0.01 %	5.74 ± 0.06 %
(C) cache-control	0%	0.004 ± 0.002 %
(C) if-modified-since	22.83 ± 0.02 %	20.6 ± 0.1 %
(S) no-cache	0.243 ± 0.002 %	0.53 ± 0.02%
(S) cache-control	0.106 ± 0.001 %	0.56 ± 0.02%
(S) expires	4.761 ± 0.009 %	5.43 ± 0.06 %
(S) last-modified	54.34 ± 0.02 %	54.5 ± 0.1 %

Table 2: **HTTP header frequencies:** this table summarizes the percent of HTTP client requests (C) and server responses (S) that contained various HTTP headers that affect caching behaviour.

of this header in data stored within a middleware cache or end server can be compared to the value of the if-modified-since client header to test whether or not cached data is stale. Unfortunately, only 1/4 of the client requests contained this header. Cache-control, no-cache, and expiration headers are extremely infrequent. These headers should become more commonly used once HTTP 1.1 compliant browsers and servers are deployed.

Internet services can benefit quite strongly from caching, as there is significant locality in a user population's references. Services must be careful to deploy an adequately large cache in order to capture the working set of that population.

3.5 Service Response Times

The recently emerging class of middleware services must take into consideration the performance of conventional content-providing Internet services as well as the characteristics of the client population. Middleware services retrieve and transform content on behalf of clients, and as such interact directly with content-providing services, relying in part on the services' performance to determine their own.

In Figure 13, we present a breakdown of the time elapsed during the servicing of clients' requests. Figure 13a shows the distribution of the elapsed time between the first byte of the client request and the first byte of the server's response observed by the trace gatherer, shown using both a linear and a logarithmic y-axis. This initial server reaction time distribution is approximately exponentially decreasing, with the bulk of reaction times being far less than a second. Internet services are thus for the most part quite reactive, but there is a significant number of very high latency services.

Figure 13b shows the distribution of the elapsed time between the first observed server response byte and the last observed server response byte (as measured by when the TCP connection to the server is shut down).[4] From these graphs, we see that complete server responses are usually delivered to the clients in less than ten seconds, although a great number of responses take many tens of seconds to deliver. (Bear in mind that the response data is being delivered over a slow modem link, so this is not too surprising.)

A number of anomalies can be seen in this graph, for instance the pronounced spikes at 0, 4, 30, and roughly 45 seconds. The spike at 0 seconds corresponds to HTTP requests that failed or returned no data. The spike at 4 seconds remains a bit of a mystery - however, note that the 4 second delivery time corresponds to 14 KB worth of data sent over a 28.8 KB modem, which is almost exactly the size of the "home_igloo.jpg" picture served from Netscape's home page, one of the most frequently served pages on the Internet. We believe that the spikes at 30 and 45 seconds most likely correspond to clients or servers timing out requests. Finally, Figure 13b shows the distribution of total elapsed time until a client request is fully satisfied. This distribution is dominated by the time to deliver data over the clients' slow modem connections.

From these measurements, we can deduce that Internet servers and middleware services must be able to handle very large amounts of simultaneous, outstanding client requests. If a busy service expects to handle many hundreds of requests per second and requests take tens of seconds to satisfy, there will be many thousands of outstanding requests at any given time. Services must be careful to minimize the amount of state dedicated to

[4] Persistent HTTP connections were very uncommon in these traces, but these special cases were handled correctly - the elapsed time until the last byte from the server for a given request is seen is reported in these Figures.

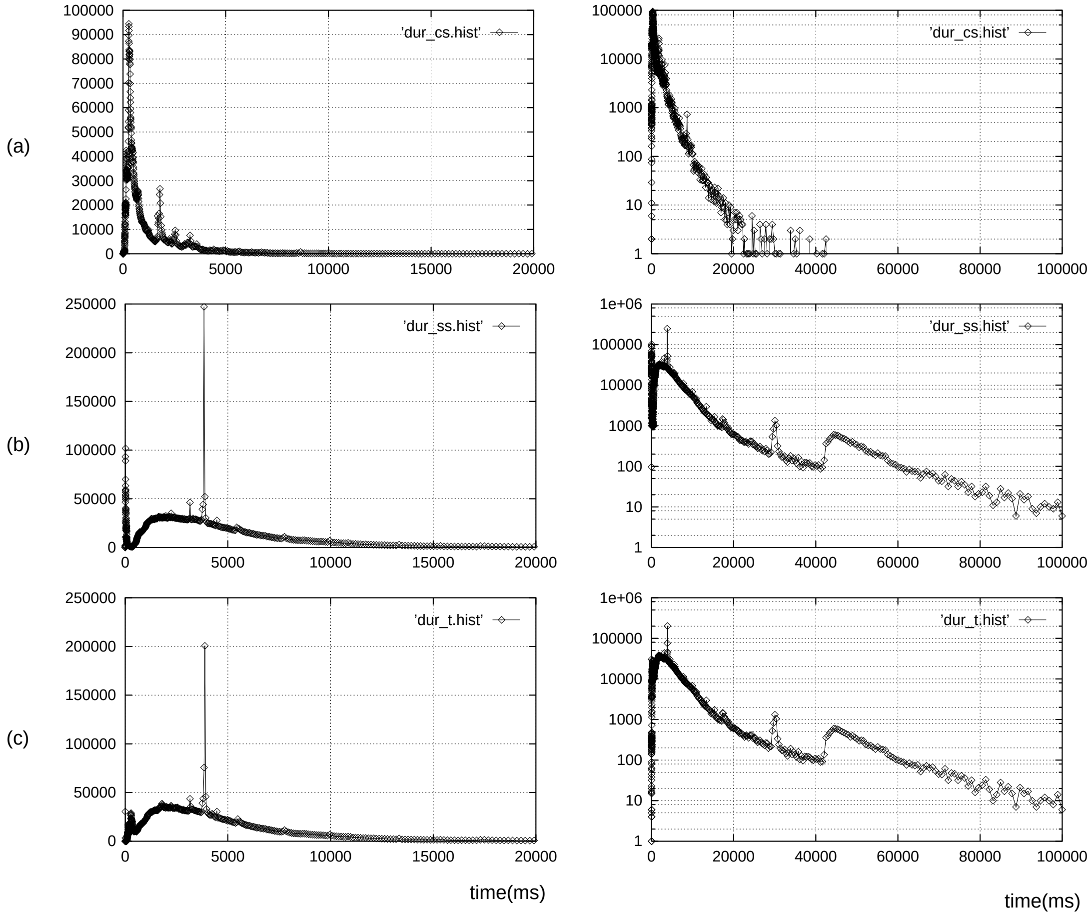

Figure 13: **Response time distributions** (a) elapsed time between the first observed byte from the client and the first observed byte from the server, (b) elapsed time between the first observed byte from the server and the last observed byte from the server, and (c) total elapsed time (between the first observed byte from the client and the last observed byte from the server). All distributions are shown with both a linear and a logarithmic Y-axis.

each individual request the overhead incurred when switching between the live requests.

3.6 Summary

This section of the paper presented a detailed analysis of the Berkeley Home IP traces. We demonstrated the following properties of the traces:

- **Heterogeneity of the user population**: although the lion's share of observed browsers are Netscape or MSIE browsers, there is significant variation across versions and therefore capabilities in these browsers. Also, there are many non-standard browsers in use, such as PDA browsers, although they generate a miniscule fraction of the total traffic.
- **A strong and predictable diurnal traffic cycle**: our web surfing population showed extremely predictable behaviour, beginning their surfing activity at 7:00am, finishing at 4:00am, peaking at midnight, and taking time off for lunch and dinner.
- **Burstiness of traffic at fine-grained time scales**: we observed peak to average traffic ratios of roughly 6:1 at second-long time scales.
- **Locality in client web requests**: with a large population size (¿8,000 users) and a large cache (¿6 GB), we observed up to 55% hit rates in our cache simulations.
- **Heavy-tailed web service response times**: the response and data delivery times that we observed from web servers ranged from milliseconds through tens of seconds.

In the next section, we discuss how these observations relate to a real Internet middleware service designed at Berkeley, the TranSend distillation proxy.

4 System Design Experience from TranSend

The TranSend middleware service provides distillation ([13], [14]) services for the Berkeley Home IP modem user population, representing roughly 8,000 active users of a bank of 600-700 modems. Distillation is data-type specific, lossy compression - for example, a distilled image may have reduced resolution or color depth, sacrificing image quality for compactness of representation. Although a small additional latency is introduced by performing distillation, the byte-wise savings realized by the more compact distilled representations more than compensates for the latency of performing the distillation, resulting in a factor of 3-7 reduction in the end-to-end latency of delivering web content to users over their slow modem links. It was therefore an explicit design goal of TranSend to help mitigate the heterogeneity of Internet clients by adapting servers' content to clients' needs.

4.1 Burstiness

The TranSend service runs on a cluster of high performance workstations. Client requests are load balanced across machines in the cluster in order to maximize request throughput and minimize the end-to-end latency of each request through the system [15]. As observed in the Home IP traces, the load presented to TranSend is quite bursty on time scales on the order of seconds. Fortunately, the service time for an individual request is on the order of milliseconds; if a burst of traffic arrives at the system, it takes only a few seconds for the backlog associated with that burst to be cleared from the system.

Over longer time scales, we have indeed observed relatively stable, non-bursty load. Certain real-world events (such as the publication of an article in the campus newspaper about the service) did trigger temporary load bursts that persisted for hours, however these bursts were extremely rare (they have only occurred two or three times during the 4 months that TranSend has been active). Because of this long-term smoothness, we were able to allocate a fixed number of cluster nodes to TranSend. To handle the infrequent long-term bursts of activity, we designed TranSend to easily recruit "overflow nodes" in times of need.

4.2 Reference Locality

The TranSend service incorporates a large web cache. We have observed that there is locality in both the pre- and post- transformed representations of web content. In our experience, a 6 gigabyte web cache has been more than sufficient to serve the needs of the Home IP service, providing close to a 50% hit rate, as predicted by the cache simulations.

4.3 Service Response Times

The two largest components of end-to-end latency perceived by the end users of TranSend are the time that it takes TranSend to retrieve content from web services on a cache miss, and the time it takes TranSend to deliver transformed content to users over the slow modem lines. The time spent by TranSend actively transforming content is less than 100 milliseconds, but content retrieval and delivery latencies often exceed tens of seconds. This means that at any given time, there are many idle, outstanding tasks supported by TranSend, and a large amount of associated idle state.

We engineered TranSend to assign one thread to each outstanding task. Because of these high latencies, we have observed that there must be on the order of 400-600 task threads available. A large amount of the computational resources of TranSend is spent context switching among these threads. In retrospect, we concluded that a more efficient design approach would have been to use an event-driven architecture, although we would certainly lose the ease of implementation associated with the threaded implementation. Similarly, each task handled by TranSend consumes two TCP connections and two associated file descriptors (one for the incoming connection, and one for the connection within TranSend to the cache). We did not attempt to measure the overhead we incurred from this large amount of network state.

5 The Berkeley Trace Playback Engine

In order to test TranSend under load, we designed and implemented a tool, the Berkeley trace playback engine, that is capable of replaying the request stream captured in the Home IP traces. We created two generations of the playback engine: a simple, parametrizable constant-rate playback engine, and a more advanced engine that can faithfully reproduce the request interarrival rates of individual clients within the traces and simulate any size client population.

5.1 The Simple Playback Engine

The simple playback engine reads entries from a tracefile, and plays them back at a constant, parameterizable, rate. The engine is massively multithreaded in order to keep many requests outstanding at any time. One thread is a dedicated scheduler: it maintains a list of scheduled tasks, and creates and dispatches a new thread to perform the task at the correct time. After a task has been dispatched, the scheduler then reads the next entry from the trace file, schedules a task for that entry, and then puts itself to sleep until the time it has to dispatch the task.

A dispatched task opens a TCP connection to the specified middleware service (in this case, TranSend), and issues the HTTP requests as observed in the trace file. The task thread then waits for the server's response; it reads the response data (but ignores the content) and then destroys itself.

The simple playback engine is limited by the time takes for TranSend to respond to HTTP requests, which in turn is largely dictated by web server latencies. This dictates how many thread tasks will be active at any given time. We were able to use a single playback engine to maintain up to 500 active task threads, after which thread creation, destruction, and context switching overhead prevented us from pushing the engine any further. If we desired more load on TranSend, we simply started up another playback engine on a second machine that read its input from a different segment of the trace files.

5.2 The Advanced Playback Engine

The more advanced generation of the trace playback engine has the ability to reproduce the request interarrival rates of individual clients. Individual clients' request streams were separated and sorted into one file per client; the advanced engine maintained an offset pointer into each of these files in order to track the clients' request streams. We reused the scheduler thread facility from the simple playback engine. For each client, we created a "shepherd" thread in the advanced engine that is responsible for scheduling successive tasks at the correct time. The shepherd thread does this by ensuring that at least a threshold number of tasks from its client are in the scheduler's queue.

Because the advanced engine reads entries from a large number of files (we tested it up to 1000 simulated clients and therefore 1000 files), we implemented a large trace entry cache that allowed us to prefetch large numbers of entries for each read() operation, amortizing the cost of the read(). The Solaris operating system only allowed us to open 1024 file descriptors, including TCP socket descriptors. For this reason, we allocated a pool of 250 file descriptors, and assigned file descriptors to client files in an LRU fashion. LRU was used because we observed that clients tend to request data in bursts: if a block of trace entries was just read from a particular client, it is highly likely that another block of entries from that client will be needed soon, and thus LRU works well.

Although the advanced engine did work, its performance left much to be desired. It would take a long time (on the order of 10 minutes) for the engine to ramp up to speed as its trace entry cache was filled up from its initial cold state. Because of the design decision to have a shepherd thread per client, at least one thread per client needed to be created, and depending on the activity of the client, many threads would be outstanding for that client. Fairly quickly, the thread overhead limited the performance of the advanced engine, causing tasks to fall behind schedule. However, the shepherds dutifully spawned tasks according to their occurences within the traces, so more and more tasks would build up, causing the engine to slow down even further, and eventually to come to a crashing halt as too much state was allocated for all of these tasks and threads.

Because of this poor performance, and because we never had a strict need to exactly duplicate the timings within the Home IP traces, we reverted back to using the simple playback engine during the testing of TranSend. The simple playback engine was perfectly suitable for our purposes, since we cared more about the volume of traffic that TranSend can handle that the specific inter-request timings within the traffic.

6 Related Work

A number of web client tracing efforts have been made in the past. One of the earliest was performed by Boston University [10], in which about a half million client requests were captured. These traces are unique in that the Mosaic browser was exclusively used by the client population; the Boston University researchers instrumented the browser source code in order to capture their traces. This research effort concentrated on analyzing various distributions in the traces, including document sizes, the popularity of documents, and the relationship between the two distributions. They used these measured distributions to make a number of recommendations to web cache designers.

Our traces are similar to the Boston University traces in spirit, although by using a packet snooper to gather the traces, we did not have to modify client software. Also, our traces were taken from a much larger and more active client population (8,000 users generating more than 24,000,000 requests over a 45 day period, as compared to the Boston University traces' 591 users generating 500,000 requests over a 6 month period).

In [20], a set of web proxy traces gathered for all external web requests from Digital Electronics Corporation (DEC) is presented. These traces were gathered by modifying DEC's two SQUID proxy caches. These traces represent over 24,000,000 requests gathered over a 24 day period. No analysis of these traces is given - only the traces themselves were made public. Only requests flowing through the SQUID proxy were captured in the traces - all web requests that flowed from DEC to external sites were captured, but there is a lack of DEC local requests in the traces.

Many papers have been written on the topic of web server and client trace analysis. In [34], removal policies for network caches of WWW documents are explored, based in part on simulations driven by traces gathered from the Computer Science department of Virginia Tech. In [9], WWW traffic self-similarity is demonstrated and in part explained through analysis of the Boston University web client traces. In [25], a series of proxy-cache experiments are run on a sophisticated proxy-cache simulation environment called SPA (Squid Proxy Analysis), using the DEC SQUID proxy traces to drive the simulation. A collection of proxy-level and packet-level traces are analyzed and presented in [12] to motivate a caching model in which updates to documents are transmitted instead of complete copies of modified documents. Finally, an empirical model of HTTP network traffic and a simulator called INSANE is developed in [23] based on HTTP packet traces captured using the *tcpdump* tool.

In [35], a tool is described for the capture of HTTP packet traces. This tool, called *HTTPDUMP*, is implemented as a set of multithreaded C routines that communicated with a packet filter via a FIFO IPC mechanism. In HTTPDUMP, a main thread classifies each incoming packet as to its type and source/destination IP address/port pair. Separate worker threads are assigned to each active HTTP connection; worker threads write out log file entries once FIN packets are received for the active session. Even though significant effort was spent on performance tuning, HTTPDUMP suffers from severe performance drawbacks. Many HTTP sessions were being dropped when HTTPDUMP collected traces from a 10 Mb/s Ethernet using a DECstation 5000/125, even with a load as light as one request per second. The IPSE tool that we used did not suffer from performance drawbacks, partly because of a cleaner design (one that did not involve IPC or threads), and also partly because of the more powerful Pentium Pro 200 MHz hardware that we used to collect the traces.

7 Conclusions

In this paper, we presented the results of an extensive, unintrusive client-side HTTP tracing efforts. These traces were gathered from a 10 Mb/s Ethernet over which traffic from 600 modems (used by more than 8,000 UC Berkeley Home IP users) flows. Forty-five days worth of traces were gathered. We used a custom module written on top of the Internet Protocol Scanning Engine (IPSE) to perform on-the-fly traffic reconstruction, HTTP protocol parsing, and trace-file generation. Being able to do this on the fly allowed us to write out only the information that interested us, giving us smaller and more manageable trace files. Anonymized versions of the traces are available through the Internet Traffic Archive (ITA) at "http://www.acm.org/ita/".

We describe two generations of a trace playback tool that allowed us to replay the Home IP traces at our middleware transformational proxy service, TranSend. The first generation tool, the simple playback engine, is capable of replaying the traffic at a constant (but parametrizable) rate. The second generation tool, the advanced playback engine, can simulate any number of users, and faithfully reproduces the interarrival rates of requests from individual users in the traces. This second generation tool suffered from poor performance, and its extra capabilities did not end up being useful; the simple playback engine sufficed for our purposes.

We measured and observed a number of interesting properties in our Home IP HTTP traces, from which we have drawn a number of conclusions related to Internet middleware service design:

1. Although most web clients can be classified as accessing Internet services using a PC-based browsers and desktop machines, there is significant heterogeneity in the client population that Internet middleware services must be prepared to handle. Specifically, the popular Netscape Navigator desktop PC browser accounts for roughly 90% of all requests that we observed; however, there were many tens of versions of the browser

that span different operating systems, release versions, and most importantly, feature sets. Furthermore, 166 unique "useragent" HTTP headers were observed, including from extravagant devices such as the Newton PDA.

2. Although there was significant diversity in the client population observed, there was less diversity in the documents that those clients accessed. GIF, JPEG, and HTML documents make up more than 90% of the documents accessed on the web, and over 75% of web traffic (in terms of bytes). GIFs account for 27% of the bytes and 51% of the documents, JPEGs account for 31% of the bytes and 16% of the documents, and HTML accounts for 18% of the bytes and 22% of the documents. The distribution of document sizes is fairly heavy-tailed, implying that there is a non-trivial number of large documents accessed on the web. The average document sizes accessed are 4.1 KB for GIF, 12.8 KB for JPEG, and 5.6 KB for HTML.

3. There is an extremely prominent diurnal cycle affecting the rate at which clients access services. Furthermore, clients' activity is relatively smooth at large time scales (on the order of tens of minutes, hours, or days), but increasingly bursty at smaller time scales (order of minutes or seconds). Internet middleware services can thus provision their resources based on the request rate observed over several hours if they can afford to smooth bursts observed over second-long time scales. We provided a simple mathematical model of this observed diurnal cycle.

4. There is a very large amount of locality of reference within clients' requests. The amount of locality (i.e., the hit rate) increases with the client population size, as does the working set of the client population. The maximum hit rate that we observed was 55% for a cache size of 6GB and a population size of 8,000 users. Caches that take advantage of this locality must grow in size in parallel with the client population that they service in order to avoid thrashing. We calculated that roughly 0.8 MB of

cache size must be added for each additional user in order to maintain the maximum possible cache hit rate. Except for the "if-modified-since" client HTTP header, we observed that HTTP caching pragmas were seldom used by either clients or servers.

5. Although Internet services tend to be very reactive (usually sending their first response byte in well under a second), the latency of delivering data to clients is quite lengthy (averaging around 3 seconds but quite often reaching latencies of several tens of seconds), implying that there could potentially be many hundreds or thousands of outstanding, parallel requests being handled by a middleware service. Services must thus minimize the amount of state and switching overhead associated with these outstanding, mostly idle tasks.

Bibliography

[1] Allpen Software Home Page. http://www.allpen.com.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] Rob Barrett, Paul P. Maglio, and Daniel C. Kellem. How to personalize the web. In *Proceedings of the 1997 Conference on Human Factors in Compuer Systems (CHI 1997)*, Atlanta, Georgia, USA, March 1997.

[4] Jan Beran, Robert Sherman, Murad S. Taqqu, and Walter Willinger. Long-range dependence in variable-bit-rate video traffic. *IEEE Transactions on Communications*, 43:1566–79, March 1995.

[5] Tim Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol - http/1.0. RFC 1945, May 1996.

[6] Timothy C. Bickmore and Bill N. Schilit. Digestor: Device-independent access to the world wide web. In *Proceedings for the Sixth International World Wide Web Conference*, 1997. available at http://www.fxpal.xerox.com/papers/bic97/.

[7] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as http stream transducers. In *Proceedings of the 4th International World Wide Web Conference*, May 1996.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[9] Mark E. Crovella and Azer Bestavros. Explaining world wide web traffic self-similarity. Technical Report TR-95-015, Computer Science Department, Boston University, Oct 1995.

[10] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of www client-traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 1995.

[11] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of SIGCOMM '93*, September 1993.

[12] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the world wide web. In *Proceedings of the 1997 USENIX Symposium on Internet Technolgoies and Systems (USITS)*, Monterey, CA, USA, December 1997.

[13] Armando Fox and Eric A. Brewer. Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation. In *Proceedings of the 5th International World Wide Web Conference*, May 1996.

[14] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variation via on-demand dynamic distillation. In *Fifth International Con-*

ference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), October 1997.

[15] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Scalable cluster-based network services. In *To Appear in the Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-16)*, October 1997.

[16] Nicolas D. Georganas. Self-similar ("fractal") traffic in atm networks. In *Proceedings of the 2nd International Workshop on Advanced Teleservices and High-Speed Communications Architectures (IWACA '94)*, pages 1–7, Heidelberg, Germany, September 1994.

[17] Ian Goldberg. The internet protocol scanning engine. Personal communications.

[18] Graphic, Visualization, & Usability Center. 6th www user survey. Summary available at http://www.cc.gatech.edu/gvu/user_surveys/survey-10-1996/.

[19] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. In *Proceedings of the SIGOPS European Workshop*, September 1996.

[20] Tom M. Kroeger, Jeff Mogul, and Carlos Maltzahn. Digital's web proxy traces. Online at ftp://ftp.digital.com/pub/DEC/traces/proxy/ webtraces.html, aug 1996.

[21] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2, February 1994.

[22] Nikolai Likhanov, Boris Tsybakov, and Nicolas D. Georganas. Analysis of an atm buffer with self-similar ("fractal") input traffic. In *Proceedings of IEEE INFOCOM '95*, Boston, MA, April 1995. IEEE.

[23] Bruce Mah. An empirical model of http network traffic. In *Proceedings of INFOCOM '97*, Kobe, Japan, apr 1997.

[24] Benoit Mandelbrot. Self-similar error clusters in communication systems and the concept of conditional stationarity. *IEEE Transactions on Communication Technology*, COM-13, 1965.

[25] David Marwood and Brad Duska. Squid proxy analysis (spa). Report available at http://www.cs.ubc.ca/spider/marwood/, 1996.

[26] Microsoft Corporation Home Page. http://www.microsoft.com.

[27] Jeffrey C. Mogul. Operating systems support for busy internet servers. In *Proceedings HotOS-V*, Orcas Island, Washington, May 1995.

[28] Netscape Corporation Home Page. http://www.netscape.com.

[29] Lawrence Berkeley Laboratory Network Research Group. Libpcap packet capture library. available at http://www-nrg.ee.lbl.gov/ftp.html.

[30] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Technical Conference*, June 1990.

[31] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of Poisson modeling. In *ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, London, UK, August 1994.

[32] Margo I. Selzter. Issues and challenges facing the world wide web. Talk presented to Lotus Corporation, available at http://www.eecs.harvard.edu/margo/slides/ lotus.html, March 1997.

[33] David Wagner. Personal communications regarding the home ip trace anonymization. June 1997.

[34] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM*, Stanford, CA, USA, August 1997.

[35] Roland Wooster, Stephen Williams, and Patrick Brooks. Httpdump network http packet snooper. available at http://ei.cs.vt.edu/ succeed/96httpdump/.