Using a Proxy to Measure Client-Side Web Performance

Richard Liston   *Ellen Zegura*

{liston, ewz}@cc.gatech.edu

Abstract— **Accurately collecting measurements of client-side browsing behavior and performance is important for making decisions about the design and operation of protocols. In this paper, we present the design and implementation of a method for measuring various variables related to the client-side browsing of web pages. For example, with this method we can measure the contribution of DNS lookups to the total latency for loading a web page as experienced by the user. We provide a discussion of the criteria that such a method should meet. We then evaluate the effectiveness of our method and discuss its limitations with respect to our design criteria. We consider the scope of questions that can be answered using this method.**

Keywords— **World Wide Web, performance measurement, HTTP, proxy.**

I. Introduction

Accurately collecting measurements of client-side browsing behavior and performance is important for making decisions about the design and operation of web-related protocols. Good measurements provide a reasonable foundation on which to base choices. They can also provide indications of where it is most fruitful to focus efforts on increasing web performance.

Our eventual interest is in the role of DNS in the response time experienced by users. The advent of content distribution networks and more complex web pages with content from multiple servers have potentially increased the importance of DNS in end-user performance. Because of this interest, we need measurements that capture the components of access response time, isolating DNS access times from object download times. We also need an accurate picture of the concurrency that results with parallel connections and the inherent overlaps in the various access times.

In addition, we are interested in measurements have the following characteristics:

- Taken from real user behavior. We want to collect measurements during standard user operation.
- Transparent to users. Since we will make measurements during operation, we want to minimize any performance degradation or other artifacts from measurement that might alter user behavior.
- Collected in a range of environments, broadly defined to include a variety of operating systems, browsers, access technologies, etc. This is to ensure that any local effects such as an atypically fast connection do not skew our conclusions. As a corollary, we require that the measurement technique require no special privileges (e.g., root access) on the part of the users participating in the study.

We describe related work in more detail later, but observe here that prior approaches to measuring user behavior have limitations in one or more of the dimensions listed above. Techniques that monitor using `tcpdump` require root access and therefore are limited to measurements in environments with privileged access. Modifying browser code to collect measurements has the potential to degrade performance; it also limits the measurement to users of browsers that we are able to instrument because source code is available.

To satisfy our objectives, we chose to implement data collection in a web proxy with one proxy instance per browser instance, as illustrated in Figure 1. It is not required that the proxy run on the same machine as the browser, but doing so minimizes the effects of other network traffic on the measurements. As illustrated, the proxy sits between the browser and any web servers accesses; it also handles all calls for DNS resolution.

The proxy behaves as follows: (1) intercepts client requests, (2) performs the DNS resolution, (3) connects to the target server and passes the request to the server. The server then (4) responds to the proxy with a header and (typically) data for the requested object. As the proxy receives the responses, it (5) passes them to the client. As the proxy passes headers and data in both directions, information of interest is parsed from both the headers and the data and (6) gets logged to a log file. We post-process the log file to extract measurements of interest.

This paper presents the design and implementation of the proxy and post-processing. The next section describes more precisely the measurement information we want to capture via logging and post-processing. We describe in some detail what is logged by the proxy and how the post-processing is done. In Section III we discuss a number of issues affecting the design and implementation of the proxy.

II. Method Overview

As described in the Introduction, our method consists of a proxy that operates between the client and the rest of the network. The proxy logs information to a log file, which is post-processed to extract information of interest. The design of the proxy must balance the goals of transparency to the user and ability to log the information necessary for recovery of measurement information.

From the standpoint of the proxy, the important characteristic of browser behavior is that the browser makes a request for a *base page*. The retrieved base page typically includes some number of *embedded objects* such as frames and images. HTML *tags* are character string tokens used by HTML to structure a web page. The embedded ob-
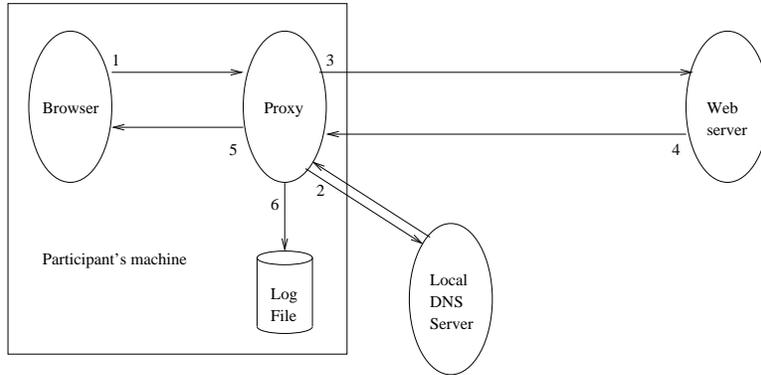
Fig. 1. Measurement architecture

jects are indicated by tags placed in the HTML code. As the server responds to this request, the proxy parses the response and logs information of interest. To keep the proxy as simple as possible so that its performance interferes minimally with the user's web experience, the proxy only performs those functions related to the retrieval of objects and logging information of interest to the log files. Each line of the log file contains the following information:

- **ID:** An identifier which is unique to each object being requested by the proxy.
- **TYPE:** A string indicating the type of the line. The complete list of TYPES is given in Appendix A. The use of some of these TYPES is highlighted in section III where we describe details of the proxy design.
- **ADDITIONAL INFO:** Each line type may cause additional information to be logged. For example, the actual IMG tag is logged on an IMG line type. Some line types, such as an **EOH** line, do not have additional info.

From the point of view of the proxy, each request is independent. The proxy maintains no explicit information specifying which objects are grouped together into pages. This function is performed by the browser as it determines how to display web pages. Often, however, the variables we wish information about depend on how the objects are grouped together into pages. Therefore, one of the most important functions to be performed during post-processing is to reconstruct how the objects are grouped to form a complete web page.

In Figure 2 we illustrate a simple example of how we perform this grouping. Figure 2-A depicts the proxy's view of objects being transmitted over time. The horizontal axis is the time line, and the thick horizontal lines depict the time from the initial receipt of the request for the object at the proxy to the time the object has been completely transmitted to the browser. On the vertical axis the objects are numbered in the order in which they are requested. Not pictured is the fact that along with information about the timing of the objects, the logs also contain the URLs of the objects and any information about objects which may be requested as a result

of receiving this object.

Figure 2-B depicts the post-processing, which occurs in two steps. First, for each object, we search the set of objects retrieved to date for a tag which caused this object to be retrieved. If one is found, we add an arrow extending from the retrieved object to the one that contains the tag.

When the complete log file has been processed, we have all the information necessary for grouping objects into pages. Figure 2-B also depicts this final stage in post-processing. Any object which does not have an arrow emanating from it is assumed to have been requested by the user. These are the base pages. We then group objects into a web page by associating with each base page all objects which can reach the base page by following a path along the arrows and objects towards the base page. Two such groupings are depicted in the figure.

This discussion describes the simplest case of post-processing. Some issues which cause difficulties and the effects of these issues on our design are discussed in the next section.

III. Proxy Design Issues

In this section we discuss issues affecting the design and implementation of the proxy. These issues determine what information was actually chosen to be logged in the log files. In the following section we discuss some of the limitations of our method.

We group the issues affecting our design into three categories. First, we discuss the issues related directly to the HTTP protocol itself. These issues are involved with information in the HTTP headers of the request and the response. Next, we discuss the issues that relate to the actual content of the objects. Finally, we discuss issues that arise from interactions between the proxy and browser.

A. HTTP Issues

Content-Encoding The HTTP protocol allows the data in a response to be encoded according to some encoding scheme. Requests may contain an "Accept-Encoding" field in the request header that indicates
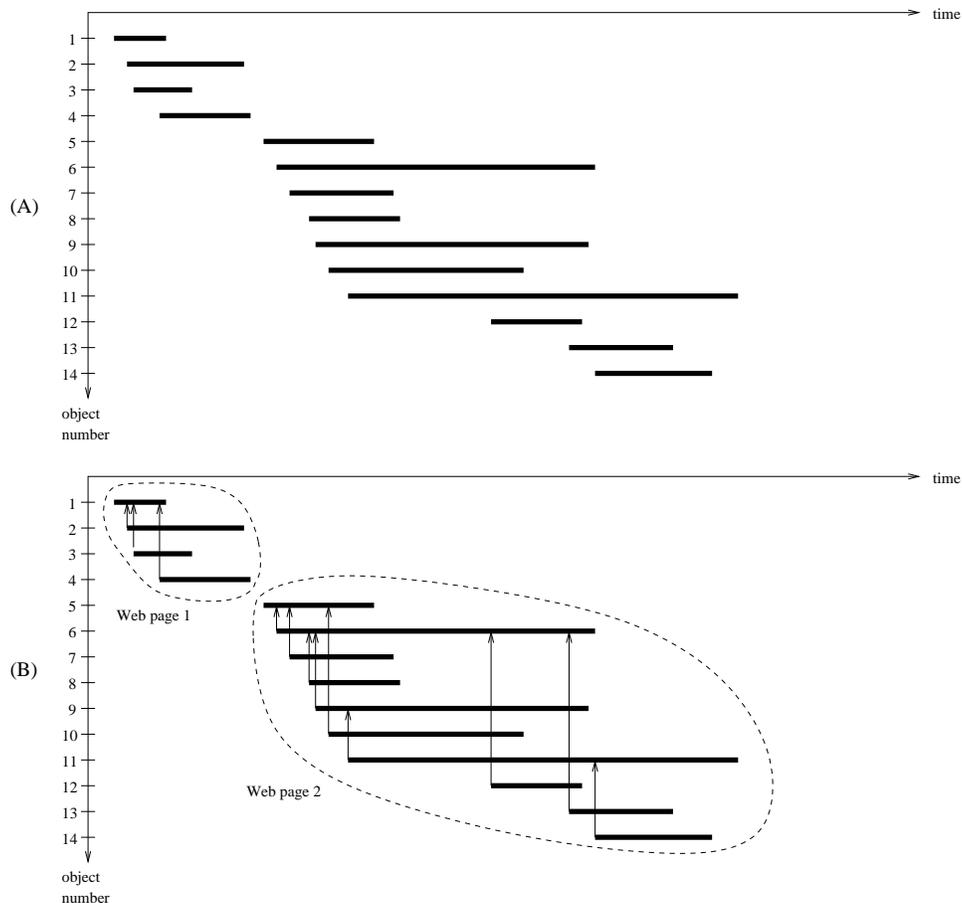
Fig. 2. Grouping objects into web pages during post-processing

to the server what encoding types the client is willing to handle. The server informs the browser via the "Content-Encoding" header field how the data must be decoded. For example, the server can deliver a gzipped HTML page accompanied by the header field "Content-Encoding: gzip". This, however, causes all HTML in the transferred object to be unreadable by the proxy since the data will not be decompressed until it is received by the browser.

To alleviate this problem, the proxy intercepts the "Accept-Encoding" field sent by the browser and changes it to "Accept-Encoding: identity; q=1.0, *;q=0". This causes the server to decode any encoded content, if possible, before sending. The proxy can then examine and parse the plain HTML. Doing so will cause a change in the timing of the rendering of the page. The degree to which this impacts proxy transparency warrants further study. However, during development of the proxy we noted few situations in which servers provided compressed HTML objects.

Location   The response header for an object may include a "Location" field. This field indicates to the

browser that the object being requested is found at an alternate URL, and provides the URL for the browser to automatically query. There may be more than one location indirection that must be followed before the request for the object is satisfied. The new URL may contain the same domain name as the initial query, a completely different domain name, or it may point to a different name within the same zone as the original query.

The proxy logs the "Location" header field, and in post-processing, objects are linked to previous requests in which the response header had a matching "Location" field.

Proxy-Connection   Our proxy implements HTTP/1.0. It therefore does not handle Keep-Alive connections. Specifically, it does not handle more than one set of headers sent from the client to the server on a single connection. The proxy handles this by removing the "Proxy-Connection: Keep-Alive" from headers sent to the server. This forces the server to close the connection after the requested object has been sent. It has the effect of requiring a separate process to be created for each requested object, each with a separate tcp handshake. Cohen et

al. [6] suggest that the effect this has on our measurements is highly dependent on time of day. Their data shows that roughly 20% of the connection-establishment times to servers can take 400 msecs during busy hours, and 1% may take as long as 8 secs. We anticipate that we can modify the proxy to support Keep-Alive connections with minimal changes to the proxy.

Content-Type Objects with certain content-types will never generate subsequent requests from the browser. So in order to reduce the performance impact of the proxy, it examines the content-type of the object as it is received from the server. If it will not cause a request to be sent, the proxy passes the data directly to the browser without parsing it.

Also, if the content-type is application (e.g., application/x-javascript or application/x-shockwave-flash), then we assume that another page caused it to be loaded. In this case, if no other heuristics have determined which base object this object should be linked to, we link it to the most recently identified base object.

B. Content-Driven Issues

JavaScript JavaScript is a scripting language that allows for development of web applications. Client-side JavaScript is dynamically interpreted by the browser and can cause the browser to retrieve remote objects. It is impossible for a proxy to predict with 100% accuracy what a request from a browser will look like when the page includes JavaScript and the browser has JavaScript turned on. A simple example of the problem is illustrated by the following JavaScript code:

```
⟨script lang="javascript"⟩
NumR=Math.floor(Math.random()*10000000);
document.write('⟨IMG SRC="http://foo.bar/' + NumR + '"⟩');
⟨/script⟩
```

In the above case, the proxy and the browser will choose different random numbers and the resulting IMG tags will not match. The post-processor will be unable to match the incoming request with the page causing the request, so it will incorrectly interpret the new request as a base page. We handle this ambiguity by causing delimiters to be placed in the proxy's log when a section of JavaScript has been encountered (see START_JS and END_JS in Table II). During post-processing, any object that occurs between the delimiters is assumed to have been generated by the JavaScript section of the previous object.

As illustrated in Figure 3 the delimiters cannot be logged while the stream is being received from the server since the timing of the proxy's parsing of the JavaScript tag and the subsequent browser requests are not synchronized. The figure shows two different timing diagrams of events seen by three entities: the browser, proxy and remote server. In Figure 3-A, the browser generates a request to the proxy which, at (1) is logged and forwarded to the server. The server responds with an object which contains HTML and JavaScript portions. At (2) the proxy logs START_JS to indicate the start of JavaScript code and at (3) it logs END_JS to indicate that the end of JavaScript code has been reached. However, the request generated by the JavaScript code is not seen by the proxy until (4), which occurs outside the delimiters.

The delimiters can be properly placed as shown in Figure 3-B. The idea is to cause the browser to send well-known requests to the proxy indicating when the JavaScript has begun and ended interpretation. The delimiters are logged when these requests are received by the proxy. Object requests resulting from the JavaScript interpretation will be more likely to fall between the two delimiters in this case. As before, the proxy logs and forwards the request at (1). At (2) the proxy receives an HTML tag of a form such as ⟨script language="javascript"⟩ and the proxy embeds the tag ⟨script language="javascript" src="http://start.js"⟩ into the page. This will cause a well-known request to be made from the browser to the proxy. The proxy intercepts this request at (3), places a START_JS delimiter in the log, and closes the connection to the browser. The browser continues parsing the subsequent HTML without making any changes to the rendering of the page. The request generated by the JavaScript code is seen by the proxy at (5), still inside the delimiters. When the ⟨/script⟩ tag is encountered by the proxy at (4), it is passed to the browser followed immediately by an embedded tag similar to the above. The browser will make another well-known request to the proxy, which at (6) will place the END_JS delimiter in the log file, close the connection and continue sending the page to the browser.

Meta Tags Another form of redirection is performed when the HTML content of a page contains a tag such as ⟨meta http-equiv="refresh" CONTENT="5; url=foo.htm"⟩. This causes browsers to wait the number of seconds indicated by the CONTENT attribute before automatically generating a request to the URL indicated by the URL attribute. The proxy cannot distinguish between a user making this request and the automatic request made by the browser, so in this instance we consider both the original page and the page to which the browser is directed to be two separate user requests. In some instances this is appropriate since the amount of delay is forced by the CONTENT attribute. Counting this in the overall page loading would artificially inflate the amount of time we measured that it took to load a page. However, if the delay is specified as 0, the META tag will effect an immediate request, which should be treated as the "Location" header field is treated. While we currently log META tags which may cause this form of redirection to take place, we do not currently count these redirections as belonging to the same request for a base page; the subsequent request is counted as a completely separate base page.
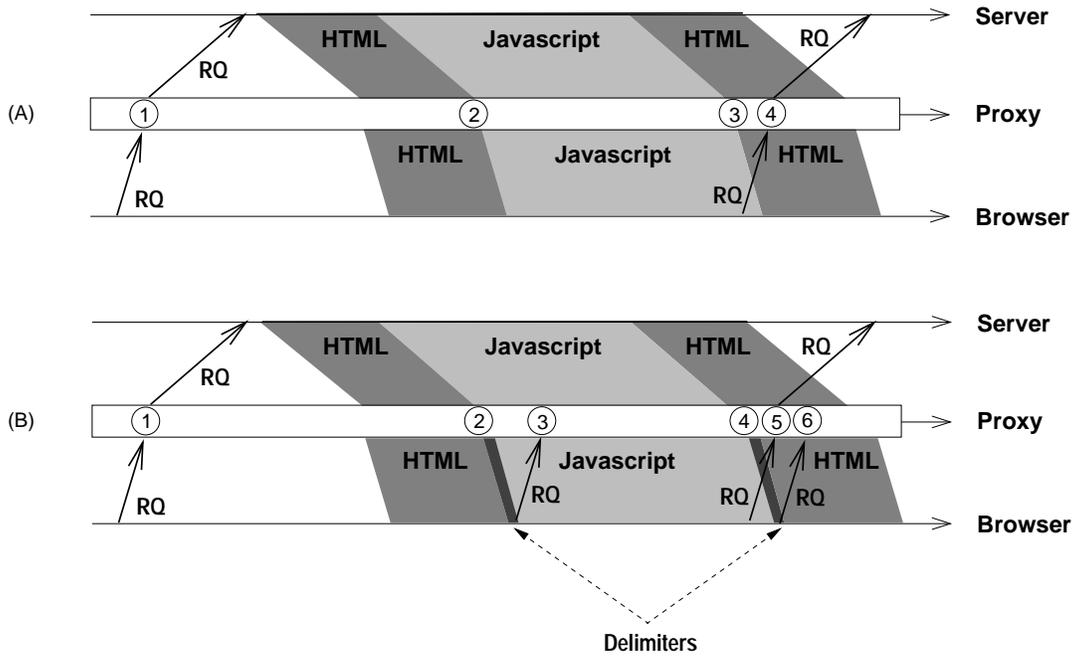
Fig. 3. Identifying requests generated by JavaScript

C. Proxy/Browser Interaction Issues

Caching   The browser typically makes a request for a base page that includes some number of embedded objects such as frames and images. As the server responds to this request, the proxy parses the embedded objects and logs the information of interest. In post-processing, the objects are grouped together into web pages.

Most browsers provide for caching of objects. Cached objects may be saved to memory and/or disk. When a browser detects that a request is being made for an object that it has in the cache, it may use the object immediately, or it may generate a request which contains an "If-Modified-Since" header. The server determines whether the object has been modified, and responds either with the object, or with a 304 (not modified) Status Code, and no message body, causing the cached copy of the page to be used.

The case may arise where the cached copy of the base page is used without a request being generated, but a request is generated for an embedded object. This causes difficulty in determining how to group objects to form web pages.

Our solution to this problem is to have the user flush the browser's disk and memory caches when the proxy is first turned on. Then for each retrieved object, we determine whether the same object was embedded in some previously requested web page. If it was embedded in the immediately preceding page, it is assumed to belong to that page. If it was embedded in a page prior to the preceding page, it is assumed that that page was re-requested and was retrieved from the cache. In this case we use the prior information to group the current objects that may appear to be unrelated.

Flushing caches will negatively affect browser performance during the period when the caches are being repopulated. After this transient period, the effect will become negligible.

Browser Awareness of Proxy Existence   In order to use the proxy, the browser must be made aware of the existence and location of the proxy. This causes browsers to behave slightly differently. Two differences which have been identified are (1) instead of generating the header line "Connection: Keep-Alive", the browser generates "Proxy-Connection: Keep-Alive" and (2) without the proxy the browser normally generates a request line with a relative GET request such as "GET /images/example.jpg", whereas with the proxy it generates a complete URL such as "http://www.example.com/images/example.jpg".

The header line generated by case (1) is removed as described in the section on "Proxy-Connection" above, so the difference in behavior has no effect. With case (2), however, if this request line is passed without mod-

ification directly to the server, in rare cases the server does not return the requested object, but returns a 403 Forbidden error. We handle this case by removing the "http://www.example.com" portion of the request. This worked without error for all servers we tested during development.

Streaming   When making a request for an object, the browser opens a TCP connection to the proxy and sends the request. The proxy performs the DNS resolution on the name of the server, opens a TCP connection to the server and forwards the request. The server responds with the header and data for the object.

The proxy reads the data sent by the server and parses it to determine whether one of the line types described in the previous section must be logged before sending the data to the browser. For example, the proxy may encounter an HTML tag such as ⟨IMG SRC="foo.jpg"⟩, which will cause an **IMG** line type to be logged.

Since TCP is a byte stream, however, a tag may occur across two consecutive data reads by the proxy. If no provisions are made for reconstructing tags that occur across consecutive data reads, many such tags will go undetected. Therefore, the proxy maintains a per-object working buffer in which partially received tags are stored. When complete tags have been received, they are parsed and the proxy determines whether the tag must be logged.

IV. Limitations

The source code for the Internet Junkbuster Proxy(TM) [1] (IJP) was the starting point for implementing our proxy. IJP provided a framework for filtering request and response headers, and allowed for insertion of code to parse the content of responses. It is a relatively lightweight proxy, provides fine-grained manipulation of header fields and provides platform independence. It also allows us to relatively easily install the proxy on a per-user basis. The Windows version of our proxy, however, is still not sufficiently robust to deploy. Work continues to make our proxy available on this platform.

The ability to parse content on the fly instead of capturing a complete web page before parsing the information we need to log appears to work extremely well. As qualitative evidence, we typically see browsers generate requests for objects while the base objects are being received, indicating that online parsing is working. While we believe the delay introduced in the received data stream by the proxy is relatively small, we have not yet evaluated the degree of this delay.

A limitation of the method is that it only measures network performance at the user's access point. It may be the case, however, that a page becomes useful to a user before all the data for the page has been received. Neither the proxy nor the post-processor can determine the point at which the page becomes useful to a user. There are several heuristics that can be applied to overcome this limitation. One is to consider each object being loaded to have a "usability threshold". This would consider a page

to be usable when the percentage of the page that has been received exceeds this threshold. Another example is to consider a page as being usable when it has caused subsequent objects to be received. The heuristics could also be different for different object types.

Some of the above limitations are more serious than others. The types and degree of the limitations will determine which performance variables are sufficiently measured by our method, or whether other methods must be developed to measure them.

V. Evaluation

We evaluated our method using a single instance of browser and proxy, both running on a Sun Ultra 1, and logging to an NFS-mounted logfile. Anecdotally, we noted that highly popular web pages tend to be the most complicated in terms of content and organization of embedded objects. Therefore, we used some of the most popular sites in several categories as ranked by Top9.com [2] to test our method. Note that this evaluation was performed prior to making the design change described in section III-C under "Caching".

At this stage in development of our method, the most important task is to be able to properly group objects into web pages during post-processing. We visited the selected sites, then post-processed the resulting logfile. The results are shown in Table I. The table lists the 41 sites we used for stress testing. For each site, it shows the total number of objects in addition to the initial request that were retrieved to satisfy a request for a web page. These additional objects include both embedded objects and Location indirections. 2232 objects in total were transferred to the browser for these 41 sites. 2191 of these were embedded objects or Location indirections. The third column shows the results of post-processing. The "# Correct" indicates the number of objects that were correctly identified as belonging to their base page. The "# Incorrect" indicates the number of objects where the method failed to identify the page to which they belonged. Objects that failed to be associated with the correct base page were identified as being base pages themselves. All the base pages were correctly identified as base pages.

114 of the 2232 embedded objects, or approximately 5%, were not correctly associated with base pages. The cause of the errors were primarily related in some fashion to the handling of JavaScript, although the specific reasons varied. For example, it is possible for a web page to define, within the START_JS and END_JS delimiters, a JavaScript function which will retrieve an object, but to invoke the function outside of the delimiters. During logfile post-processing, without using other heuristics, we will not detect the subsequent object as being embedded in the page that invoked the function.

The objects that were identified as base pages included both those that were correctly and incorrectly identified as base pages. There are a total of 114 + 41 = 155 such objects.

Site	# Additional Objects	# Correct/# Incorrect
www.real.com	68	68/0
www.napster.com	68	67/1
www.mp3.com	17	17/0
www.winamp.com	25	19/6
www.liquidaudio.com	51	49/2
www.peoplesound.com	95	74/21
www.audiofind.com	7	7/0
www.musicmatch.com	60	59/1
www.lyrics.com	4	3/1
www.launch.com	54	49/6
www.marsmusic.com	39	35/4
www.ubl.com	61	57/4
www.billboard.com	78	72/6
www.harmony-central.com	21	21/0
www.musiclyrics.net	45	43/2
www.music.com	68	64/4
www.mtv.com	59	54/5
www.discovery.com	54	49/5
www.abc.com	87	86/1
www.pbs.org	99	92/8
www.tvguide.com	35	32/3
www.gist.com	47	46/1
www.hollywood.com	75	72/3
www.ifilm.com	64	60/4
www.moviefone.com	62	60/2
www.movietickets.com	86	83/3
www.channel2000.com	79	76/3
www.thebostonchannel.com	91	85/6
www.imdb.com	49	49/0
www.movies.com	50	49/1
www.dragonballz.com	6	6/0
www.startrek.com	82	82/0
www.starwars.com	47	47/0
www.twistedhumor.com	18	17/1
www.funstun.com	45	45/0
www.windowsmedia.com	47	45/2
www.adobe.com	63	62/1
www.allposters.com	37	37/0
www.eonline.com	49	49/0
www.disney.com	37	35/2
www.ebay.com	62	57/5
Totals:	2191	2077/114

TABLE I
RESULTS OF TESTING.

As noted in previous sections, we plan to evaluate the degree to which the proxy affects performance as experienced by the user. The major topics for this evaluation are (1) the impact of limiting the "Accept-Encoding", (2) the impact of disallowing "Keep-Alive" connections, and (3) the delay introduced by the proxy.

VI. RELATED WORK

There are other tools and methods that have been developed to obtain web performance data. They vary in their objectives and therefore, in their architectures.

Crovella, Cunha et al. [7] [8] instrumented a browser to capture the URL, time of access and transfer time. As mentioned earlier, this method limits us to browsers for which source code is readily available.

A tool currently under development is called *webtest* [9]. The objective of *webtest* is closely related to ours. It is intended to measure various aspects of the performance of loading random web pages. By default the web pages it uses are drawn from a set of names from

a large database of URLs obtained by a web spider.

Choi et al. [5] developed a method for measuring user characteristics as an aid in modeling user behavior. While they achieve a high degree of accuracy with their measurements, their method requires access to privileged network information.

Some studies have used shared web cache logs and domain proxy logs in order to measure various aspects of web performance [6] [3] [4]. An advantage of this method is the ready availability of the logs. There are some drawbacks, however. First, the measurements are made on a machines which are not co-located with the user, and the effects of network characteristics between the user and the point of measurement are not available. Second, the domain proxy and web cache do not capture all the information we are interested in. For example, DNS lookup time and how objects are grouped together to form a page are not logged. Finally, they reflect web performance experienced in a limited number of settings.

VII. Conclusions and Future Work

Our method provides us with the ability to relatively transparently measure web performance as experienced by actual users. In the future we will be measuring the degree to which web performance degradation will be perceived. The method can currently be applied on UNIX systems, and the proxy can be installed by users without special privilege. Work is continuing to make the proxy available on other platforms. In the following we discuss some of the variables we expect to be able to measure with our method.

DNS Prefetching Our primary interest in developing the tool was to investigate policies related to DNS prefetching. Prefetching is a general technique in which data that has a high probability of being required in the near future is retrieved and placed in a cache near the point where it will be used. It is most effective when the cache miss penalty is high. Prefetching has been applied at both the chip level (e.g., to prefetch instructions from memory) and the network level (e.g., to prefetch entire web pages).

DNS prefetching is an application of the technique to DNS name resolution. Domain names which have a high probability of being resolved in the near future are resolved. Using this technique, some DNS cache misses can be avoided. When predictions are incorrect, however, resources are unnecessarily consumed. This has the potential, in fact, cause a decrease in performance. Some criteria for performing DNS prefetching are proposed in [6].

Using our method, we will investigate the contribution made by DNS lookups to user-perceived delays in loading web pages, and consider the advantages and disadvantages of DNS prefetching in terms of speedup and consumption of resources: network bandwidth, nameserver load and memory usage.

Servers Contacted We will be able to model the number of servers contacted and the time that they are contacted during the loading of web pages.

JavaScript Performance Since we are able to determine the point at which JavaScript code begins and ends execution, we will be able to estimate with a high degree of accuracy how long a specific section of JavaScript takes to execute on a variety of platforms.

Embedded objects We intend to model the distribution by type of object embedded into web pages. A simple extension to the proxy will allow us to log the size of each object by type. This is done by logging the "Content-Length" header field of each received object.

Object Retrieval Depth We define the *Object Retrieval Depth* (ORD) of a web page to be the maximum number of links followed from any object retrieved for the web page to the base page. As a (not farfetched) example, consider the following scenario: the response to an initial URL request contains a "Location" header field; the object retrieved as a result contains three frames, one of which has JavaScript code; the JavaScript code causes a request for a URL which is an advertising banner; the response for the request for the advertising banner contains another "Location" header indicating where the banner should be retrieved. In this example, the ORD is 4. We will be able to model the ORD of web pages.

VIII. Acknowledgements

References

[1] Internet Junkbuster Proxy(TM). http://www.junkbuster.com/ijb.html.

[2] Top9.com. http://www.top9.com/.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *IEEE Infocom*, pages 126–134, March 1999.

[4] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web Proxy Caching: The Devil is in the Details. In *Proceedings of Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[5] H. Choi and J. Limb. A Behavioral Model of Web Traffic. In *Proceedings of International Conference on Network Protocols*, Sep 1999.

[6] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. In *IEEE Infocom*, March 2000.

[7] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *IEEE/ACM Transactions on Networking*, volume 5(6), pages 835–846, December 1997.

[8] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. Technical report, TR-95-010, Boston University Computer Science Department, 1995.

[9] C. Huitema. Webtest - testing the internet quality of service. http://ww.netsizer.com/webtest.html.

Appendix A

Line Type	Description
ENTER	Provides a timestamp when the proxy first receives a request from the browser.
REQ	Logs the actual request line sent from the browser.
HOSTNAME	Name of the machine in the URL of the requested object, and the time it took for the name to be resolved.
EOH	Indicates that the proxy has received and parsed the complete header.
META	Indicates that the received object contained a META tag. META tags may contain a "url" attribute causing another object to be requested.
BODY	Indicates that the received object contained a BODY tag. BODY tags may contain a "background" attribute causing another object to be requested.
IMG	Indicates that the received object contained an IMG tag. IMG tags are required to contain a "src" attribute which may cause another object to be requested.
INPUT	Indicates that the received object contained an INPUT tag. INPUT tags may contain a "src" attribute causing another object to be requested.
TD	Indicates that the received object contained a TD tag. TD tags may contain a "background" attribute causing another object to be requested.
TABLE	Indicates that the received object contained a TABLE tag. TABLE tags may contain a "background" attribute causing another object to be requested.
EXIT	Provides a timestamp when the proxy has completed handling the request.
ELAPSED	Provides a calculation of how long the proxy took to load this object.
APP	Indicates that the received object is an application as specified by the **Content-Type** header field, and will not cause another object to be requested.
APPLET	Indicates that the received object contained an APPLET tag. APPLET tags may contain a "src" attribute causing another object to be requested.
FRAME	Indicates that the received object contained a FRAME tag. FRAME tags may contain a "src" attribute causing another object to be requested.
LOCATION	Indicates that the header for the requested object contained a **Location** header field. Upon receiving this header field, browsers immediately make a request to for the indicated object.
REFERER	Indicates that the header of the request contained a **Referer** header field. This field can indicate which object caused the immediate request for the object, or it can indicate the page that contained the link the user selected in order to retrieve the object.
START_JS	Provides an indication that the browser has begun interpreting a section of javascript. See section III-B.
END_JS	Provides an indication that the browser has completed interpreting of a section of javascript. See section III-B.

TABLE II

LOG FILE LINE TYPES.