

The World Wide Wait: Where Does the Time Go?

Colin Allison, Martin Bramley, Jose Serrano

Division of Computer Science,
University of St Andrews,
Scotland, KY16 9SS

Abstract

The continuing explosive growth of the web has not been matched by an adequate enhancement of the infrastructure on which it depends. Both consumers and producers are often left frustrated at the disappointingly slow speed of service. Yet there has been little systematic attempt to analyse the sources of delay experienced by users. We present a structured timing model which covers the entire period from a user clicking on a URL to the resultant display becoming usable. The model is not predicated on any particular server, browser or networking infrastructure and provides a basis for discovering the constituent parts of the total delay as experienced by the user in any given situation. Only when the delay is broken down is it possible to make productive decisions about tuning systems and replacing hardware or software components. We illustrate the use of the model with reference to understanding and reducing delays encountered in Finesse, a widely distributed and highly interactive teaching and learning environment.

1. Introduction

The volume of traffic on the World Wide Web (web) has increased exponentially over recent years [1]. It is increasingly criticised by users for slow response times which make web browsing an unpleasant process. It is particularly frustrating in cases where a site is providing an interactive application such as the Finesse Portfolio Management Facility [2]. The total delay experienced may be due to a wide range of causes such as network delay, browser design, server overload or poor HTML [3] page design. However, because analysing this delay and breaking it down into its constituent parts is not trivial, people often make decisions about how to improve their web service based on little more than guess work. Work to-date, as summarised in Section 6, has focused on

improving web server performance or web communication protocol efficiency [4,5,6,7] with little regard for the users perspective. In this paper we make three contributions: we present a simple structured timing model to guide analysis of total delay; we describe a number of lightweight methods for measuring the constituent parts of the total delay; we list a set of recommendations for reducing delay in specific situations. For the sake of clarity we restrict this report to the common and frequent case of basic web operations, static HTML retrieval and CGI invocation, as described in the next section.

2. Basic Web Operations

There are two fundamental standards associated with the web: Hyper Text Mark-up Language (HTML) and Hyper Text Transfer Protocol (HTTP [8]). The former specifies tags for structuring the appearance of text, graphics and hypertext. Hypertext tags, called Uniform Resource Locators (URLs), can trigger HTTP operation when the resource is remote. HTTP is a basic client-server, request-response model. The client is typically an interactive application called a *browser* and the server application is an opaque remote process. Figure 1 summarises basic web operations. 1a shows a basic web interaction involving a client and a server. The server response may be a simple file retrieval and transmission, or the invocation of a program via the Common Gateway Interface (CGI), which generates output that is returned to the client, 1b and 1c show the potential for concurrency in client and server. HTTP 1.0¹ [8] has four methods of communication between the client browser and the web server. These are GET, POST, HEAD and PUT. The PUT method is used to upload documents from the client to the server.

¹ HTTP 1.1 is briefly discussed in Section 6.

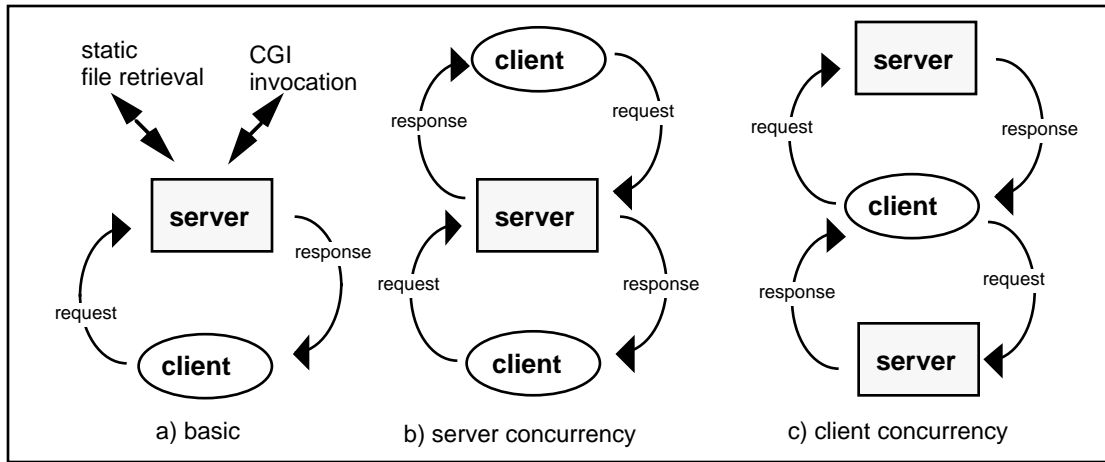


Figure 1: Web Client Server Model

The HEAD method is used by search engines to request meta-information about the requested page to create an entry in the search engines database. The POST method is used in HTML forms to send structured information entered via the web browser back to the web server. The GET method is by far the most frequently used method, which requests an entire document from the server. It is the GET method we are most interested in as it is typically a user triggered event to which the user awaits for the returned data to be output. This returned data could be of four generic types; static-HTML, applets, streams and output from CGI programs. These programs may return any type of data although in practice it is often HTML.

2.1 Proxies and caching

There are two widespread optimisation strategies used to minimise delay. *Client-side caching* works by keeping copies of recently accessed files on local storage. An HTTP request-response period can be shortened: a check is made on the header of the remote file, a comparison is made with the header of the local copy in cache, and, if there has been no change, the local copy is used. *Proxy servers* can be setup at any number of levels e.g. local, campus or national. The idea is that a browser will go to the proxy to see if the file is cached (i.e. it has been requested by at least one other user). If the proxy does not have the file it will attempt to fetch it, cache it and then return it to the client. Unfortunately, neither of these optimisations improve the response times of CGI programs, which are highly interactive and non-cacheable. Accordingly, we address neither directly in this paper. HTTP 1.1[17] offers new support for the caching of database query results but exactly how this will reduce CGI response time in general remains to be seen.

3 A Structured Timing Model

The model covers the entire period from the user initiated selection of a URL, to the resolution of all its associated links within the scope of an HTML page. We refer to this period as the closure over a URL, or *CURL*. A CURL may terminate correctly or incorrectly. When a CURL terminates correctly all images, files, applets and other components which are associated with a URL are located or generated, retrieved, displayed and/or activated. A CURL may terminate incorrectly if there is a failure in any of its constituent parts. There are numerous possible sources of failure and many will not result in any diagnostic message being delivered to the user. In order to simplify the description of the model we will only consider CURLs which terminate correctly. A CURL involving an HTTP URL is initially broken down into one or more instances of a top level three phase pattern:

- A) client-side HTML parse and request generation
- B) server-side request processing: response generation may be CGI or static HTML.
- C) client-side rendering time consisting of parse time and display update

Network transit times, including source and destination protocol processing times, are represented by N_{CS} and N_{SC} , where N_{CS} denotes the time taken during the communication phase from client to server, and N_{SC} the delay in the other direction. Phase C may result in further requests and there is usually an overlap between request generation and other HTML parsing.

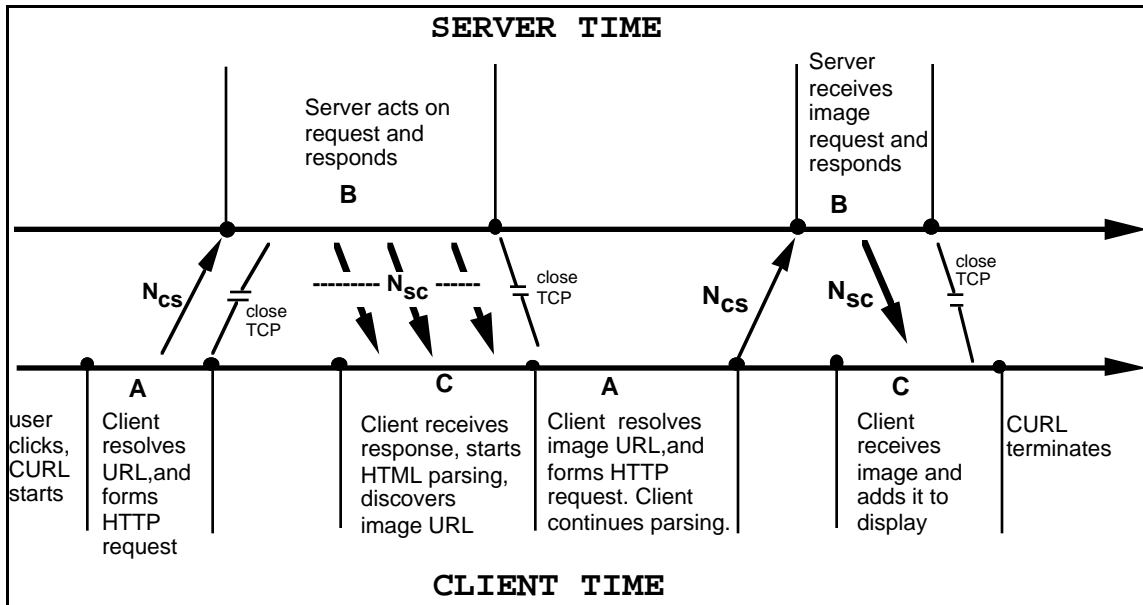


Figure 2: Three Phase Pattern and Overlap

Figure 2 shows an example CURL consisting of a URL which results in the retrieval of an HTML page which in turn contains a single image reference. The top level phases can be broken down further. The following list illustrates basic web interaction:

Phase A

1. Make a DNS lookup if raw IP number not used in URL (the typical case).
2. Client opens a TCP connection to the server.
3. Client sends an HTTP GET request which includes client-side environment variables.
4. Client closes TCP connection.

Phase B

1. Server listener process receives the GET request and spawns or calls a process to deal with it. Exact concurrent server implementation varies widely [9].
2. Server parses the request.
3. Server opens a TCP reply connection . (Always necessary, even if only to report error).
4. Server does a DNS lookup on client.
5. Searches for .htaccess files in directory containing requested file and parental directories.
6. If not allowed due to (5) returns fail message. Write to error log.
7. Determines requested type and invokes appropriate handler e.g. CGI or Static HTML.
- 8a. In the case of CGI, handler starts CGI program, passes it CGI variables.

- 8b. In the case of Static HTML returns file.
9. Return data is buffered for network transmission.
10. Server closes TCP connection.

Overlap between Phases B and C

1. Client starts reading from the receive buffer and parsing the HTML
2. The server flushes the buffer to the client and closes the connection
3. Server writes to access log.

Phase C only

1. Client finishes reading from receive buffer and parses HTML into part of viewable web page.
2. If Client comes across an image URL it enters Phase A with the URL as a parameter.

This model is still evolving but has already proved useful in guiding analysis of delay in the Finesse environment which is described as a case study in Section 5. The next section explains the options and difficulties in getting real and useful timing figures out of the web.

4. Timing Issues

To measure a CURL we would ideally have access to the source code for the servers, browsers and their enclosing operating systems. In practice reasonable estimates may be achieved by using Javascript on the browser and CGI programs on the server. This allows

us to insert timing probes through-out the resolution of a CURL. One other important assumption is that the Network Time Protocol (NTP) [10] is running on the server and all the clients. NTP clients are widely available for most platforms and were operational on all the machines in our tests. Figure 3a is an outline of a typical CGI script which will be used as an example. Figure 3b is a modified version which displays timing information. The script is written in TCL [18].

4.1 Minimum Client Rendering Time

In order to estimate the clients rendering time (Phase C) network overheads are eliminated by running the browser on the server and using loopback communication. The end of client rendering time may be gained by using JavaScript's *onLoad* method. The start of client parse time is harder to record because there is not a Javascript event which is triggered when the client starts to receive the server's response. At the same time a client may start parsing before a CGI program has finished running on the server. However, we can get the minimum rendering time by measuring the time elapsed between the server completing CGI execution and the browser completing the rendering of the resultant HTML page.

4.2 Network Overhead Time

It is possible to get an approximation of the time it takes to transport the data across the network by eliminating the clients parse time and using the same CGI script as in the previous timing method. This involves running the client browser on a machine which is equivalent to the server. If we assume that the parse time is approximately the same as that obtained in 4.1, then the remaining time is the network transfer. Within the CURL this may be expressed as ($N_{sc} + \text{Phase C time}$) - Phase C time. $N_{sc} + \text{Phase C}$ is timed as stated in this paragraph and Phase C time is estimated as described in 4.1.

4.3 Fastest CGI Run Time

As an aid to comparison it is useful to get the minimal run time of the CGI program. To do this the network, client, and server overheads must be eliminated. This may be achieved by running the CGI program in a command shell. This gives an absolute minimum possible Phase B time.

4.4 Actual Run Time

This is the actual Phase B time within the CURL i.e. the actual time taken for the CGI script used in 4.3 to run, plus the overheads involved when the output is being passed back through the web server to the browser running on the client. It has been noticed that the slower the client machine, the longer the script will take to run. This appears to be a knock-on effect from limited buffering at the client and server, and slow buffer emptying at the client. When both the client and the server buffers are full the CGI process sleeps until more buffer space is available for its output.

4.5 Client Request Time

This is the time period between the client submitting the request to the server till the server starting sending data back to the client browser. It is measured by appending a time-stamp to the client request and comparing it to a time stamp at the start of the CGI script. This provides timings of Phase A + N_{cs} .

4.6 Extending and refining analysis

More detail may be obtained on network behaviour by running a network snooping program in the same collision domain as the server and client (or either one). This allows for non-intrusive monitoring of IP traffic of interest. Server and browser profiles on Unix can be created by using system call tracing tools such as *truss* [19]. Detailed analysis on personal computer operating systems such as Windows 95 and MacOS is more difficult, but possible.

```
#!/usr/local/bin/tclsh
# a generic CGI script written in TCL

package require cgi

cgi_eval {
    cgi_input          # variables setting up the environment

    cgi_head {        # specifying the head of the generated HTML document
    }

    cgi_body {        #main block of CGI code creating main body of HTML document
    }
}
```

Figure 3a: CGI script outline - no timing information displayed

```

#!/usr/local/bin/tclsh
#a modified CGI script for timing various sections.

# get the start time
set start [clock seconds]

package require cgi

cgi_eval {
  cgi_input

  # if variable sendtime exists import it (Time user made request for
  # document)
  set sendtime ""
  catch { cgi_import sendtime }
  cgi_head {
    # javascript function to display load completion time
    cgi_javascript { cgi_puts {
      function whattimeisit() {
        return (new Date()).toLocaleString()
      }
    } }

  }

  # body modified to display time on completion of document parsing
  cgi_body onLoad=alert(whattimeisit()) {
    # if sendtime does not exist allow user to reload document
    # with sendtime.
    if {$sendtime == ""} {
      cgi_form timing onSubmit=this.sendtime.value=whattimeisit() {
        cgi_export sendtime
        cgi_submit_button
      }
    } else {
      # run normal CGI code
      # main block of CGI code creating main body of HTML
      # document

      # now display cgi parsing times
      puts "<P>Client request time:$sendtime\n"
      puts "<P>Start CGI time: [clock format $start]\n"
      puts "<P>End CGI time: [clock format [clock seconds]]\n"
    }
  }
}
}

```

Figure 3b: script modified to provide timing

5. Case Study

Finesse is a web-based interactive teaching and learning environment for finance education. It is run on a single web server serving four universities connected by a 34Mb/s IP/ATM infrastructure across distances between 30 and 200 Km. Due to a large and diverse base of client machines it is not possible to make any assumptions other than they have web-browsing capabilities. To this end the entire application has been written in interactive CGI scripts. The largest script generates an HTML table of current stock market prices. It has seven columns and over a thousand rows, and is about 180 Kbytes. We use this as an example here not only because it seems to stress browser and server-side processing to the limits, but also because it is the single most frequently requested page from the server. Note that the small raw data size of the table belies its ability to cause delay.

The timings listed in Tables 1a and 1b were obtained using the techniques described in Section 4. Table 1a consists of timings made when only one client was accessing the server. Table 1b shows the results for a concurrent load of four clients. The server timings in Table 1a were generated by a client being run on the server, giving a basis for estimating network overhead time when they are subtracted from the same constituent times measured from an identical model of computer as the server across the network.

The following computers were used as clients:

Mac:	200 Mhz Mac 8600, 64Mb, MacOS 7.6.1
Sun:	Ultra 140, 64Mb, Solaris 2.6
PII:	266Mhz Pentium II, 64Mb, Windows 95
486:	Intel 66Mhz 486, 20Mb, Windows 95.

The server is a Sun Ultra 140, configured identically to the client. We also tested two different types of browser on the Intel clients.

	Mac	Sun	PII Netscape	PII Explorer	486 Netscape	486 Explorer	Server (Sun)
total time (CURL)	60	45	38	24	285	103	43
submit time (Phases A+B - overlap)	0	0	0	0	0	0	0
CGI runtime overlap - Phase C	26	22	18	19	56	28	21
minimum network & client rendering time (Phase C)	34	23	20	5	229	75	22

Table 1a: CURL constituent timings in seconds for single client access

	Mac	Sun	PII Netscape	486 Netscape
total time (CURL)	130	81	85	283
submit time (Phases A+B - overlap)	0	0	0	0
CGI runtime overlap - Phase C	70	58	62	79
minimum network & client rendering time (Phase C)	60	23	23	204

Table 1b: CURL constituent timings in seconds for four client concurrent access

5.1 Analysis and solutions

This simple but replicable set of tests allows us to draw useful conclusions about the delays as perceived by the user. Perhaps the most significant result is that network overheads are very insignificant whilst client rendering time and CGI runtime account for the bulk of the delays. There are a number of improvements that can reduce the delay without having to upgrade either computer or network hardware: choice of browser, revision of output format and optimisation of CGI programs. We briefly discuss each in the context of the case study.

The choice of browser and version of browser is important, as shown by the differences between Netscape and Explorer in the same situation.

Revising the format of the results sent to the client can reduce rendering time. For example, large HTML tables take a surprisingly large amount of time. One solution to this particular overhead is to format the data into a more concise form. We have reduced the data from approximately one thousand to six hundred rows without information loss or reduction in display quality, and this has resulted in a 25% reduction in rendering time. Further improvements could conceivably be achieved through the use of pre-formatted text or the conversion of table text to images.

CGI program times have been reduced by tackling the problem of concurrent data loading. The results of commonly called functions are now cached in shared memory regions (outside of the HTTP server) and this has drastically reduced CGI run times. Although we have not modified the HTTP server itself we would expect that increasing network buffer sizes would also result in fewer CGI programs running at the same time.

While none of these optimisations are particularly novel or complex in themselves it is important to know where to optimise, and that is what the analysis based on the timing model provides. The combined effect of the optimisations described above has been to reduce the CURL time by approximately 80%.

6. Related work

Analytical work on the web has focused mainly on the respective performance of web servers and the HTTP protocol and we have not found any other model which provides a comprehensive analysis of a CURL. Nevertheless the work is related in that server and protocol overheads form part of the delay as experienced by the user. Hu et. al. analyse design aspects of Web servers in high-speed networks and show their influence on performance [9]; Slothouber proposes a mathematical model of server activity which explains the relationship between server and network speeds [11]. Benchmarking systems such as WebStone [12] and SPECweb96 [13] have been developed to measure the performance of servers. These work by generating simulated workloads of concurrent client requests on a server. Improvements to these types of tools have been proposed, for example Surge [14] and [15]. WebCompare [16] offers a comprehensive list of servers and technical features as an aid for site administrators to decide which server best suits their organisation. The list is a more of a catalogue than a true comparison however. HTTP has often been criticized for its inefficiency, but mostly with regard to network latency and Internet congestion. Padmanabhan and Mogul support the idea that the multiple TCP connections required by HTTP 1.0 are a significant cause

of latency in the Web [5]. Spero [6] analyses a typical HTTP transaction with timings and concludes that the interaction between HTTP and TCP is a major cause of delay in the Web. On the other hand work reflected in [7] suggests that the overhead of multiple TCP connections is insignificant for low bandwidth users and that its effect is only apparent when using high-speed connections. Edwards and Rees have an interesting paper about the execution of CGI applications in servers [4] through HTTP. HTTP 1.1 [17] attempts to address some of the shortcomings in HTTP 1.0. New features include:

- *persistent connections* – this allows for an initial TCP connection between a server and a client to be kept open. It cuts out multiple TCP Open and Close packets thus matching the typical session semantics of the browser-server relationship better, reducing the number of IP packets during a session, and reducing repeated startup latency overhead.
- *pipelining of connections* further exploits persistent connections by allowing multiple requests and responses to be placed in a single packet. Reduces number of packets and the number of network round trip delays.
- *reliable caching* – support for servers to mark the results of a CGI output as cacheable with an expiration time set to that of the next expected database update. Results of popular queries can be cached at proxies and invalidated according to the servers update cycle. This could potentially benefit the Finesse share price database by removing the CGI run time from the delay experienced by users making the popular request for a full listing.

7. Conclusions and further work

We have presented a simple but useful analytical model for measuring web delays as experienced by users. Such analysis is essential if productive decisions are to be made about reducing the delay. The notion of a CURL, a closure over the resolution of a URL, underlies the model. As a CURL includes client-side, network, and server-side components it complements the extensive focus on web server and web protocol performance by bringing client-side delays and CGI performance into the picture. A set of techniques for making the measurements needed by the model have been described and their use has been illustrated in a case study analysis of a problem drawn from the Finesse distributed learning environment. Optimisation techniques have been sketched, and when applied to the specific bottlenecks identified in the case study have resulted in an 80% reduction in the overall CURL time and a drastically reduced concurrent load on the server.

Further desirable improvements to our model include a fully automatic system which will log reports on a routine basis, and a better set of timing tools. We have focused on CGI generated HTML and ignored several novel features of the web but there is no reason why the model cannot be extended to accommodate timings associated with scenarios such as reliable caching or media streaming.

8. References

- [1] Gray, M.K. "Web Growth Summary"
<http://www.mit.edu/people/mkgray/net/web-growth-summary.html>
- [2] Finesse: Finance Education in a Scalable Software Environment <http://tullamore.accountancy.dundee.ac.uk/>
- [3] Berners-Lee T., Connolly D, *Hypertext Markup Language*. RFC 1866, Nov. 1995.
- [4] N. Edwards, O. Rees. "Performance of HTTP and CGI", <http://www.ansa.co.uk/ANSA/ISF/1506/APM1506.html>
- [5] V. N. Padmanabhan, J. C. Mogul. "Improving HTTP Latency", Proceedings of the 2nd International WWW Conference.
<http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html>
- [6] Spero, S. E. "Analysis of HTTP Performance Problems"
<http://sunsite.unc.edu/mdma-release/http-prob.html>
- [7] Touch, J, J. Heidemann, K. Obraczka. "Analysis of HTTP Performance" <http://www.isi.edu/lisam/publications/http-perf/>
- [8] T. Berners-Lee, R. Fielding, H. Frystyk. "Hypertext Transfer Protocol HTTP/1.0" RFC 1945, May 1996.
- [9] Hu J.C., Smedh M., Schmidt D. C. "Techniques for Developing and Measuring High-Performance Web Servers over ATM Networks", submitted to InfoCom '98.
- [10] Mills, D. *Network Time Protocol*. RFC 1129.
- [11] Slothouber, L. P. "A model of Web Server Performance"
<http://louvx.biap.com/webperformance/modelpaper.html>
- [12] Mindcraft, Inc. "WebStone". Software.
<http://www.mindcraft.com/webstone/>
- [13] "SPECweb96". Software.
<http://www.spec.org/osg/web96/>
- [14] P. Barford, M. Crovella. "Generating Representative Web Workloads for Network and Server Performance Evaluation", Technical report BU-CS-97-006.
<http://www.cs.bu.edu/techreports/97-006-surge.ps.Z>
- [15] M. F. Arlitt, C. L. Williamson. "Internet Web Servers: Workload characterization and implications", *IEEE/ACM Transactions on Networking*, Vol. 5. No. 5, 631-644, October 1997
- [16] WebCompare <http://webcompare.internet.com>
- [17] Fielding et al. "Hypertext Transfer Protocol HTTP/1.1" RFC 2068.
- [18] Ousterhout, John K. "Tcl and the Tk toolkit" Addison-Wesley, 1994.
- [19] SUN Microsystems. Solaris 2.6 Manual, 1997.