Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Performance of HTTP and CGI

Nigel Edwards

Abstract

This paper reports a series of performance experiments investigating the performance of an HTTP server (CERN HTTPD). The objective was to learn what parts of the server consume the most CPU time, and thus on what aspects implementors should concentrate to improve performance.

The results show that the minimum time to complete an HTTP POST request running a CGI program should be about 80 milliseconds.This time includes: 43 milliseconds for the HTTP server to parse the request headers; 24 milliseconds to fork and exec all the processes to handle the requests. In practice we were unable to better 200 milliseconds — the reasons for this are discussed.

In conclusion, these results show that for high performance, better encoding of HTTP header information is needed. ASCII headers are great for debugging, but make for inefficient parsing. In addition, although the fork/exec model of execution enforces stateless interaction, higher performance would be achieved by having a threaded server. Also instead of using CGI to extend the functionality of the server and gateway to other applications, it would be better to define extension HTTP methods. In this model the server could send you an HTML form which invoked an extension HTTP method rather than overloading GET and POST to launch a CGI program. The extension HTTP method would be executed as a thread inside the server.

Threads alone are not sufficient for high performance, because parsing the HTTP headers is CPU bound.

APM.1500.00.02 **Draft** 1st June 1995

Request for Comments (confidential to ANSA consortium for 2 years)

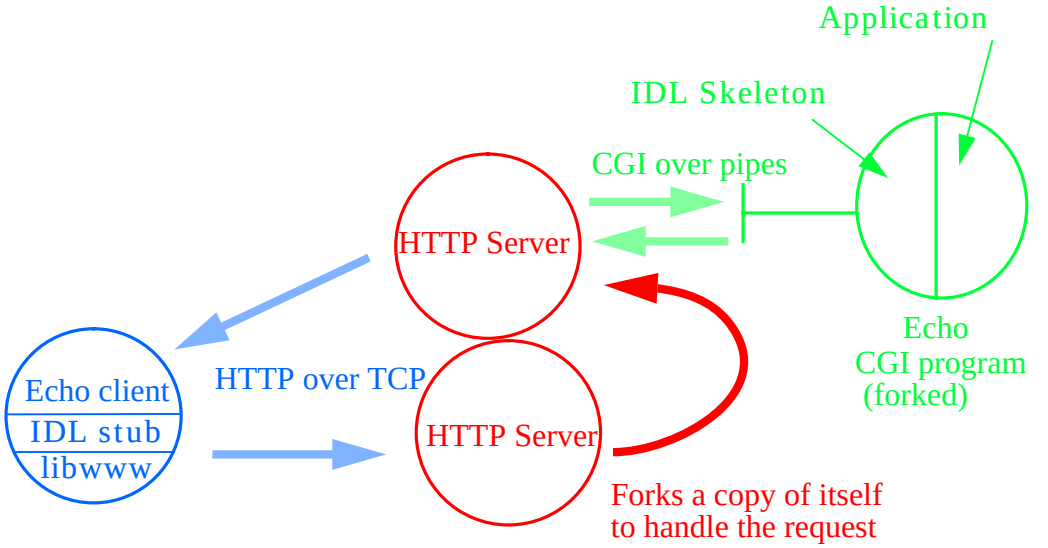
Distribution:
Supersedes:
Superseded by:

1 Performance of HTTP and CGI

This paper reports a series of performance experiments investigating the performance of an HTTP server running a CGI program (CERN HTTPD 3.0 [HTTPD] using HTTP 1.0 on a Hewlett-Pacakard PA-RISC machine). The objective was to learn what parts of the server consume the most CPU time and thus on what aspects implementors should concentrate to improve performance.

The experiment consisted of a client program (written using the programming kit described in [EDWARDS 95]) invoking a CGI program launched by a CERN HTTP server. The CGI program was a variant of the Echo program described in [EDWARDS 95]. The HTTP method used to launch the program was “POST”, sending a body of 542 bytes to the CGI program and receiving the same amount of data back. The set up is shown in figure 1.1. Details of the IDL stubs and skeletons are discussed in [EDWARDS 95].

Figure 1.1: HTTP Performance Experiment configuration



All experiments were conducted on a lightly loaded Hewlett-Packard PA-RISC machine (725 running HPUX 9.01). For most experiments the HTTP server and client were located on the same machine. Section 1.2.1 shows that placing them on different machines connected to the same ethernet did not affect performance.

Before we started the experiments we removed all the logging and unnecessary input/output from the server. After this, even if the daemon is run with -v (verbose) set, nothing is printed. Under these conditions the execution times running the daemon in -v mode and running it normally where within 2% of each other. The remainder of this paper is structured as follows:

- Section 1.1 describes how we measured the fork and exec time for processes;
- Section 1.2 describes the details of the experiments and can be skipped on a first reading;
- Section 1.3 summarises the results and makes suggestions for improving the performance of HTTP and HTTP servers.

1.1 How long does it take to do a fork/exec?

We wrote a program which forks a process, then writes a 1000 bytes down one pipe and reads a 1000 bytes from the other. At the other ends the child engages in the appropriate reads and writes. We found that this takes around 10 milliseconds on a lightly loaded Hewlett-Packard PA RISC machine.

If the child execs another process to do the input/output on the pipes, the whole things take 24 milliseconds to complete.

1.2 Details of the Experiments

This section describes the details of the experiments we conducted. The results are summarised in section 1.3 and figure 1.2, so on a first reading you may want to skip to section 1.3.

All times are averaged over 1000 invocations unless otherwise stated.

1.2.1 Experiment 1

The client invokes the Echo CGI program and measures time to return result. This took 202 milliseconds. (The figure of 202 milliseconds is also obtained when the client and server are on different PA-RISC machines connected to the same Ethernet.)

1.2.2 Experiment 2

We wrote a new CGI program which return two longs and a string (the latter being the echo payload). The IDL skeleton for this service was hand-crafted into the HTTP server. The code of the server was changed so that if it launched a CGI program it would wait for the CGI program to terminate, ignore the results and use the hand-crafted IDL skeleton to return two longs and a string to the client. The two longs marshalled as the results were HT_Start (the start time in milliseconds) and HT_stop (the stop time in milliseconds).

We timed from the point that the forked server starts to immediately before it sends the reply data down TCP/IP connection to the client (but after it has sent the headers). This took 183 milliseconds. This implies that it is taking 10 milliseconds for round trip TCP traffic (including slow start and establishing the connection) and for client to set up call and decode results (including MIME parsing). (We are deducting the 10 milliseconds to fork an HTTP process to deal with the request).

1.2.3 Experiment 3

We timed from the call to fork the CGI process to same point as above. This took 140 milliseconds. This implies that it takes 43 milliseconds to parse the headers and decode an incoming request.

The headers of the incoming request are shown below:

```
POST /cgi-bin/Echo2 HTTP/1.0
Accept: application/x-corba
Accept: */*; q=0.300
Accept: application/octet-stream; q=0.100
Accept: text/plain
Accept: text/html
From: nje@socrates.ansa.co.uk
User-Agent: Echo client/0 libwww/2.17
Content-length: 542
Content-type: application/x-www-form-urlencoded
```

The server parses the headers by doing a string comparison on each line. Note that a normal browser would include much more extensive header information than this, because it can cope with many more types of media.

In CERN HTTPD the headers are parsed in HTParseRequest(). This routine reads the headers in one line at a time from the socket. However, measurements suggest that this is non-blocking (or at least no significant blocking): we measured the time to read in each line to be about 60 microseconds.

1.2.4 Experiment 5

We timed from call to fork CGI process to the point where the HTTP daemon has finished writing data down the pipe to the CGI process. This was an average of 133 milliseconds per call. So it would seem it takes about 7 milliseconds for the CGI process to unmarshal the data, marshal the results and write them back down the pipe. This is not surprising given that all data in the body of a request is url-encoded, so each character need checking to see if it needs decoding (or unescaping).

There is no need to url-encode all data in the body of the POST. Since the data is not included in the headers, it cannot confuse the parsing machinery. However, this emulates current browsers, which url-encodes all data regardless of whether they are invoking GET or POST from a form. (The encoding is necessary for GET, since the request body is included in the HTTP headers).

1.2.5 Experiment 6

We started the timer at the point where the HTTP daemon starts writing down the pipe. Stopped it when it finishes writing down the pipe. This took 124 milliseconds, which is astonishing considering it is writing only 542 bytes.

Note that if we subtract 124 milliseconds from 133 milliseconds (obtained in section 1.2.4) we get 9 milliseconds which confirms the time to fork a process measured independently in section 1.1.

1.2.6 Experiment 7

Examining the source of the HTTP Daemon we noted that it first reads in the HTTP headers and processes them. It is not until it has forked the child that it reads the body of the request using HTInputSocket_getBlock(). So we timed the length of execution of this routine. The result was 122 milliseconds. This is clearly one of the major performance bottle necks.

1.2.7 Experiment 8

We placed calls of gettimeofday() either side of NETREAD() in HTInputSocket_getBlock() in HTFormat.c in libwww. On HPUX NETREAD() is a call to read(). It reads in the body of the request from the socket. We found that on a lightly loaded PA-RISC machine (even with the client and server "niced" to run at the top priority of 0) the call to NETREAD() still took about 110 milliseconds (averaged over 10 invocations). It seems likely that the kernel must be doing some scheduling here and that a context switch occurs, blocking the server for a period of time.

When a client does a "POST" using libwww, it sends the headers and then immediately makes a separate second call to write() on the socket sending the body of the request. If the client and server are on the same machine, one possible scenario is that the CPU does a context switch after the client has sent the headers and then runs the server until it blocks trying to read in the body of the request. Next the client would be rescheduled to send the body of the request.

However, I do not believe that this context switching is the major performance overhead. As otherwise we would expect to see a significant speed up when client and server were executed on different machines. We don't.

It seems most likely that what is causing this blocking is of the server is some interaction with the underlying sockets and TCP library. (Probably during the reading of the headers.) We have not pursued this any further.

1.3 Summary of results and conclusions

Figure 1.2 shows a time line for the execution of a CGI POST request using the data obtained from the experiments.

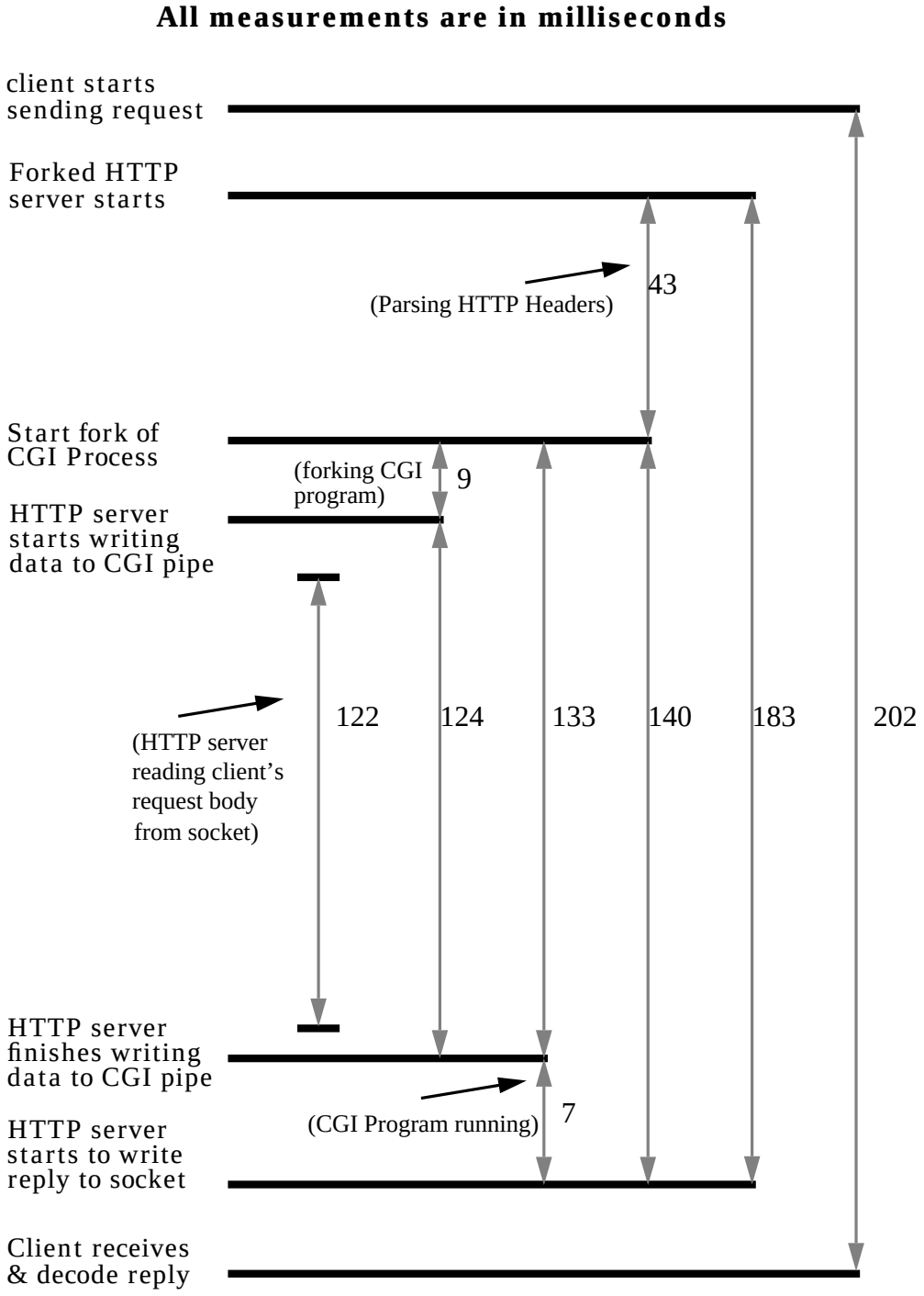
These results show that the minimum time to handle a CGI POST request using HTTP 1.0 on a PA RISC machine should be about 80 milliseconds for the CERN HTTP Server. This includes:

- 43 milliseconds for the HTTP server to parse the request headers;
- 24 milliseconds to fork and exec all the processes to handle the requests.

The headers sent with the request are just about the minimum you would get (see section 1.2.3). A typical browser will include many more headers, as it will be able cope with many more media types. So the time of 43 milliseconds will almost certainly be exceeded in practice. This is a CPU bound activity (no blocking on i/o), so using threads will not help to lower this overhead.

Of the remaining 23 milliseconds 7 milliseconds are taken by the CGI program to url-decode the data and format the results (this will be more if more data is sent in the request). I speculate that most of the remaining 16 milliseconds are spent by the client preparing the request (e.g. url-encoding the data), parsing the reply headers and setting up the TCP connection. (Setting up the TCP

Figure 1.2: Summary of Results



connection will almost certainly take longer and extend this time in a wide-area network.)

In conclusion, these results show that for high performance, better encoding of HTTP header information is needed. ASCII headers are great for debugging, but make for inefficient parsing. (I understand the developers of HTTPng are considering a binary encoding for headers.)

Although the fork/exec model of execution enforces stateless interaction, higher performance would be achieved by having a threaded server. Also instead of using CGI to extend the functionality of the server and gateway to

other applications, it would be better to define extension HTTP methods. In this model the server could send you an HTML form which invoked an extension HTTP method rather than overloading GET and POST to launch a CGI program. The extension HTTP method would be executed as a thread inside the server.

I am unable to explain is why the server “hangs” for 122 milliseconds to read in 542 bytes of data which have almost certainly arrived at the socket. So the quoted minimum time of 80 milliseconds is arrived at by subtracting 122 milliseconds from the measured minimum time of 202 milliseconds.

References

[EDWARDS 95]

"A Stub Compiler for CGI and HTTP: The Programmer's Guide", APM.1465.00.02 , Nigel Edwards, APM Ltd., Cambridge U.K., May 1995.

[HTTPD]

"CERN HTTP server status", <url:http://www.w3.org/hypertext/WWW/Daemon/Status.html>.