
httperf

httperf is a tool for measuring web server performance. It provides a flexible facility for generating various HTTP workloads and for measuring server performance.

The focus of httperf is not on implementing one particular benchmark but on providing a robust, high-performance tool that facilitates the construction of both micro- and macro-level benchmarks. The three distinguishing characteristics of httperf are its robustness, which includes the ability to generate and sustain server overload, support for the HTTP/1.1 and SSL protocols, and its extensibility to new workload generators and performance measurements.

Building httperf

This release of httperf is using the standard GNU configuration mechanism. The following steps can be used to build it:

In this example, SRCDIR refers to the httperf source directory. The last step may have to be executed as “root”.

First, some tools which are required for the build process need to be installed.

```
1 $ sudo apt install automake libtool
```

Then, run the following steps in order to build. Note that some of these might have to be executed as “root”, i.e., with sudo.

```
1 $ autoconf
2 $ libtoolize --force
3 $ autoreconf -i
4 $ automake
5 $ ./configure
6 $ make
7 $ sudo make install
```

This step may need to be run as root:

```
1 make install
```

Since httperf 0.9.1, the the idleconn program is no longer built by default. Using the configure option `--enable-idleconn` will instruct the build system to compile the tool.

To build httperf with debug support turned on, invoke configure with option “`--enable-debug`”.

By default, the httperf binary is installed in `/usr/local/bin/httperf` and the man-page is installed in `/usr/local/man/man1/httperf`. You can change these defaults by passing appropriate options to the “configure” script. See “configure `--help`” for details.

This release of `httpperf` has preliminary SSL support. To enable it, you need to have OpenSSL (<http://www.openssl.org/>) already installed on your system. The configure script assumes that the OpenSSH header files and libraries can be found in standard locations (e.g., `/usr/include` and `/usr/lib`). If the files are in a different place, you need to tell the configure script where to find them. This can be done by setting environment variables `CPPFLAGS` and `LDFLAGS` before invoking “configure”. For example, if the SSL header files are installed in `/usr/local/ssl/include` and the SSL libraries are installed in `/usr/local/ssl/lib`, then the environment variables should be set like this:

```
1 CPPFLAGS="-I/usr/local/ssl/include"
2 LDFLAGS="-L/usr/local/ssl/lib"
```

With these settings in place, “configure” can be invoked as usual and SSL should now be found. If SSL has been detected, the following three checks should be answered with “yes”:

```
1 checking for main in -lcrypto... yes
2 checking for SSL_version in -lssl... yes
3      :
4 checking for openssl/ssl.h... yes
```

Note: you may have to delete “`config.cache`” to ensure that “configure” re-evaluates those checks after changing the settings of the environment variables.

WARNING: `httpperf` uses a deterministic seed for the random number generator used by SSL. Thus, the SSL encrypted data is likely to be easy to crack. In other words, do not assume that SSL data transferred when using `httpperf` is (well) encrypted!

This release of `httpperf` has been tested under the following operating systems: HP-UX 11i (64-bit PA-RISC and IA-64) Red Hat Enterprise Linux AS (AMD64 and IA-64) SUSE Linux 10.1 (i386) openSUSE 10.2 (i386) OpenBSD 4.0 (i386) FreeBSD 6.0 (AMD64) Solaris 8 (UltraSparc 64-bit)

It should be straight-forward to build `httpperf` on other platforms, please report any build problems to the mailing list along with the platform specifications.

Mailing list

A mailing list has been set up to encourage discussions among the `httpperf` user community. This list is managed by majordomo. To subscribe to the list, send a mail containing the body:

```
1 subscribe httpperf
```

to majordomo@linux.hpl.hp.com. To post an article to the list, send it directly to httpperf@linux.hpl.hp.com.

Running httpperf

IMPORTANT: It is crucial to run just one copy of httpperf per client machine. httpperf sucks up all available CPU time on a machine. It is therefore important not to run any other (CPU-intensive) tasks on a client machine while httpperf is running. httpperf is a CPU hog to ensure that it can generate the desired workload with good accuracy, so do not try to change this without fully understanding what the issues are.

Examples

The simplest way to invoke httpperf is with a command line of the form:

```
httpperf -server wailua -port 6800
```

This command results in httpperf attempting to make one request for URL `http://wailua:6800/`. After the reply is received, performance statistics will be printed and the client exits (the statistics are explained below).

A list of all available options can be obtained by specifying the `-help` option (all option names can be abbreviated as long as they remain unambiguous).

A more realistic test case might be to issue 100 HTTP requests at a rate of 10 requests per second. This can be achieved by additionally specifying the `-num-conns` and `-rate` options. When specifying the `-rate` option, it's generally a good idea to also specify a timeout value using the `-timeout` option. In the example below, a timeout of one second is specified (the ramification of this option will be explained later):

```
httpperf -server wailua -port 6800 -num-conns 100 -rate 10 -timeout 1
```

The performance statistics printed by httpperf at the end of the test might look like this:

```
1 Total: connections 100 requests 100 replies 100 test-duration 9.905 s
2
3 Connection rate: 10.1 conn/s (99.1 ms/conn, <=1 concurrent connections)
4 Connection time [ms]: min 4.6 avg 5.6 max 19.9 median 4.5 stddev 2.0
5 Connection time [ms]: connect 1.4
6 Connection length [replies/conn]: 1.000
7
8 Request rate: 10.1 req/s (99.1 ms/req)
9 Request size [B]: 57.0
10
11 Reply rate [replies/s]: min 10.0 avg 10.0 max 10.0 stddev 0.0 (1
    samples)
12 Reply time [ms]: response 4.1 transfer 0.0
```

```
13 Reply size [B]: header 219.0 content 204.0 footer 0.0 (total 423.0)
14 Reply status: 1xx=0 2xx=100 3xx=0 4xx=0 5xx=0
15
16 CPU time [s]: user 2.71 system 7.08 (user 27.4% system 71.5% total
    98.8%)
17 Net I/O: 4.7 KB/s (0.0*10^6 bps)
18
19 Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
20 Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

There are six groups of statistics: overall results (“Total”), connection related results (“Connection”), results relating to the issuing of HTTP requests (“Request”), results relating to the replies received from the server (“Reply”), miscellaneous results relating to the CPU time and network bandwidth used, and, finally, a summary of errors encountered (“Errors”). Let’s discuss each in turn:

“Total” Results

The “Total” line summarizes how many TCP connections were initiated by the client, how many requests it sent, how many replies it received, and what the total test duration was. The line below shows that 100 connections were initiated, 100 requests were performed and 100 replies were received. It also shows that total test-duration was 9.905 seconds meaning that the average request rate was almost exactly 10 request per second.

```
1 Total: connections 100 requests 100 replies 100 test-duration 9.905 s
```

“Connection” Results

These results convey information related to the TCP connections that are used to communicate with the web server.

Specifically, the line below show that new connections were initiated at a rate of 10.1 connections per second. This rate corresponds to a period of 99.1 milliseconds per connection. Finally, the last number shows that at most one connection was open to the server at any given time.

```
1 Connection rate: 10.1 conn/s (99.1 ms/conn, <=1 concurrent connections)
```

The next line in the output gives lifetime statistics for successful connections. The lifetime of a connection is the time between a TCP connection was initiated and the time the connection was closed. A connection is considered successful if it had at least one request that resulted in a reply from the server. The line shown below indicates that the minimum (“min”) connection lifetime was 4.6 milliseconds, the average (“avg”) lifetime was 5.6 milliseconds, the maximum (“max”) was 19.9 milliseconds, the

median (“median”) lifetime was 4.5 milliseconds, and that the standard deviation of the lifetimes was 2.0 milliseconds.

```
1 Connection time [ms]: min 4.6 avg 5.6 max 19.9 median 4.5 stddev 2.0
```

To compute the median time, httpperf collects a histogram of connection lifetimes. The granularity of this histogram is currently 1 milliseconds and the maximum connection lifetime that can be accommodated with the histogram is 100 seconds (these numbers can be changed by editing macros BIN_WIDTH and MAX_LIFETIME in stat/basic.c). This implies that the granularity of the median time is 1 millisecond and that at least 50% of the lifetime samples must have a lifetime of less than 100 seconds.

The next statistic in this section is the average time it took to establish a TCP connection to the server (all successful TCP connections establishments are counted, even connections that may have failed eventually). The line below shows that, on average, it took 1.4 milliseconds to establish a connection.

```
1 Connection time [ms]: connect 1.4
```

The final line in this section gives the average number of replies that were received per connection. With regular HTTP/1.0, this value is at most 1.0 (when there are no failures), but with HTTP Keep-Alives or HTTP/1.1 persistent connections, this value can be arbitrarily high, indicating that the same connection was used to receive multiple responses.

```
1 Connection length [replies/conn]: 1.000
```

“Request” Results

The first line in the “Request”-related results give the rate at which HTTP requests were issued and the period-length that the rate corresponds to. In the example below, the request rate was 10.1 requests per second, which corresponds to 99.1 milliseconds per request.

```
1 Request rate: 10.1 req/s (99.1 ms/req)
```

As long as no persistent connections are employed, the “Request” results are typically very similar or identical to the “Connection” results. However, when persistent connections are used, several requests can be issued on a single connection in which case the results would be different.

The next line gives the average size of the HTTP request in bytes. In the line show below, the average request size was 57 bytes.

```
1 Request size [B]: 57.0
```

“Reply” Results

For simple measurements, the section with the “Reply” results is probably the most interesting one. The first line gives statistics on the reply rate:

```
1 Reply rate [replies/s]: min 10.0 avg 10.0 max 10.0 stddev 0.0 (1
  samples)
```

The line above indicates that the minimum (“min”), average (“avg”), and maximum (“max”) reply rate was ten replies per second. Given these numbers, the standard deviation is, of course, zero. The last number shows that only one reply rate sample was acquired. The present version of `httperf` collects one rate sample about once every five seconds. To obtain a meaningful standard deviation, it is recommended to run each test long enough so at least thirty samples are obtained—this would correspond to a test duration of at least 150 seconds, or two and a half minutes.

The next line gives information on how long it took for the server to respond and how long it took to receive the reply. The line below shows that it took 4.1 milliseconds between sending the first byte of the request and receiving the first byte of the reply. The time to “transfer”, or read, the reply was too short to be measured, so it shows up as zero (as we’ll see below, the entire reply fit into a single TCP segment and that’s why the transfer time was measured as zero).

```
1 Reply time [ms]: response 4.1 transfer 0.0
```

Next follow some statistics on the size of the reply—all numbers are reported in bytes. Specifically, the average length of reply headers, the average length of the content, and the average length of reply footers are given (HTTP/1.1 uses footers to realize the “chunked” transfer encoding). For convenience, the average total number of bytes in the replies is also given. In the example below, the average header length (“header”) was 219 bytes, the average content length (“content”) was 204 bytes, and there were no footers (“footer”), yielding a total reply length of 423 bytes on average.

```
1 Reply size [B]: header 219.0 content 204.0 footer 0.0 (total 423.0)
```

The final piece in this section is a histogram on the status codes received in the replies. The example below shows that all 100 replies were “successful” replies as they contained a status code of 200 (presumably):

```
1 Reply status: 1xx=0 2xx=100 3xx=0 4xx=0 5xx=0
```

Miscellaneous Results

This section starts with a summary of the CPU time the client consumed. The line below shows that 2.71 seconds were spent executing in user mode (“user”), 7.08 seconds were spent executing in sys-

tem mode (“system”) and that this corresponds to 27.4% user mode execution and 71.5% system execution. The total utilization was almost exactly 100%, which is expected given that httpperf is a CPU hog:

```
1 CPU time [s]: user 2.71 system 7.08 (user 27.4% system 71.5% total
98.8%)
```

Note that any time the total CPU utilization is significantly less than 100%, some other processes must have been running on the client machine while httpperf was executing. This makes it likely that the results are “polluted” and the test should be rerun.

The next line gives the average network throughput in kilobytes per second (where a kilobyte is 1024 bytes) and in megabits per second (where a megabit is 10^6 bit). The line below shows an average network bandwidth of about 4.7 kilobyte per second. The megabit per second number is zero due to rounding errors.

```
1 Net I/O: 4.7 KB/s (0.0*10^6 bps)
```

The network bandwidth is computed from the number of bytes sent and received on TCP connections. This means that it accounts for the network payload only (i.e., it doesn’t account for protocol headers) and does not take into account retransmissions that may occur at the TCP level.

“Errors”

The final section contains statistics on the errors that occurred during the test. The “total” figure shows the total number of errors that occurred. The two lines below show that in our example run there were no errors:

```
1 Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
2 Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

The meaning of each error is described below:

total: The sum of all following error counts.

client-timo: Each time a request is made to the server, a watchdog timer is started. If no (partial) response is received by the time the watchdog timer expires, httpperf times out that request a increments this error counter. This is the most common error when driving a server into overload.

socket-timo The number of times a TCP connection failed with a socket-level time out (ETIMED-OUT).

connrefused The number of times a TCP connection attempt failed with a “connection refused by server” error (ECONNREFUSED).

connreset The number of times a TCP connection failed due to a reset (close) by the server.

fd-unavail The number of times the httpperf client was out of file descriptors. Whenever this count is bigger than zero, the test results are meaning less because the client was overloaded (see discussion on setting `-timeout` below).

addrunavail The number of times the client was out of TCP port numbers (EADDRNOTAVAIL). This error should never occur. If it does, the results should be discarded.

ftab-full The number of times the system's file descriptor table was full. Again, this error should never occur. If it does, the results should be discarded.

other The number of times other errors occurred. Whenever this occurs, it is necessary to track down the actual error reason. This can be done by compiling httpperf with debug support and specifying option `-debug 1`.

Selecting appropriate timeout values

Since the client machine has only a limited set of resource available, it cannot sustain arbitrarily high HTTP request rates. One limit is that there are only roughly 60,000 TCP port numbers that can be in use at any given time. Since, on HP-UX, it takes one minute for a TCP connection to be fully closed (leave the TIME_WAIT state), the maximum rate a client can sustain is about 1,000 requests per second.

The actual sustainable rate is typically lower than this because before running out of TCP ports, a client is likely to run out of file descriptors (one file descriptor is required per open TCP connection). By default, HP-UX 10.20 allows 1024 file descriptors per process. Without a watchdog timer, httpperf could potentially quickly use up all available file descriptors, at which point it could not induce any new load on the server (this would primarily happen when the server is overloaded). To avoid this problem, httpperf requires that the web server must respond within the time specified by option `-timeout`. If it does not respond within that time, the client considers the connection to be “dead” and closes it (and increases the “client-timo” error count). The only exception to this rule is that after sending a request, httpperf allows the server to take some additional time before it starts responding (to accommodate HTTP requests that take a long time to complete on the server). This additional time is called the “server think time” and can be specified by option `-think-timeout`. By default, this additional think time is zero, so by default the server has to be able to respond within the time allowed by the `-timeout` option.

In practice, we found that with a `-timeout` value of 1 second, an HP 9000/735 machine running HP-UX 10.20 can sustain a rate of about 700 connections per second before it starts to run out of file descriptor (the exact rate depends, of course, on a number of factors). To achieve web server loads bigger than that, it is necessary to employ several independent machines, each running one copy of httpperf. A

timeout of one second effectively means that “slow” connections will typically timeout before TCP even gets a chance to retransmit (the initial retransmission timeout is on the order of 3 seconds). This is usually OK, except that one should keep in mind that it has the effect of truncating the connection life time distribution.