System Architecture & design decisions

1. Server

We design server to use select approach or fork approach. We can specify it with different arguments when compile.

1.1 Server using select

Server using select is really straightforward. In this implementation, server will call select and serve each incoming request. If it is a connection request, then server will serve this connection.

The most important part of select is connection time handling for HTTP/1.1 because there is no specific API for us to use. Thus, we need some information for each incoming connection. Because each connection represents a file descriptor, so we allocate a struct table for all file descriptors. When a connection is being built, the connected field is marked and begin time is recorded. Then select would timeout for each second to check all file descriptors. If a file descriptor is connected and current time minus begin time is bigger than timeout, the file descriptor will be closed. When closing a connection, we just need to unmark the connected field for the closed file descriptor.

1.2 Server using fork

Server using fork is much easier because we only have to fork a child process for each incoming connection.

Timeout check is relatively easy because we only have to call an API: setsockopt. This API will set a timeout for a read. The read function call will automatically return zero when time's up.

1.3 Common part

After building a connection, both select and fork would call serve_client. From now on, everything is architecture independent. serve_client would read data from a specific fd, then check the data and write corresponding data back according to the content.

1.4 Saturation

In order to saturate server, we make server busy by letting it write dummy messages. If client want to trigger this functionality, it only have to get a file named pics/pic_mountains.jpg from server. When server find out the requested file size is 75672, server will write dummy message to stderr for each write. With file size other than 75672, this behavior will not be triggered. This can help us to make two different purposes clients, one for experiment, and one for saturation. We can compile a client for experiment or saturation as we want with different compilation parameters.

2. Client

2.1 Client design

Client is designed to use fork approach. For experiment purpose client, a specific file is specified to be downloaded for totally 100 times. That's to say, for a experiment purpose client with 5 child processes being created, than each process only have to download a file 20 times. And for a saturation purpose client, a specific size file is specified to be downloaded for 100000000 times. Then we can make sure that server can be continuously saturated when we are doing experiment.

2.2 Instrumentation

In order to get experimental data, some part of the client source should be instrumented. Briefly speaking, two parts are important, one is connect, the other is read/write. We instrument connect because we need to know the time used for each connection request. We instrument read/write because we need to the time used for data transmission. Each child process would calculate its own time information such as transmission time and connection time and write to a shared memory space. Then parent process would do the final calculation.

Methodology

1: Evaluation Environment

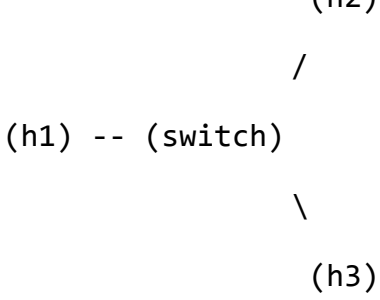
It is not easy to saturate a server. Thus, we let the server do much more things than just sending/receiving files. Our experiments and saturation are all done on mininet running on the virtual box.

1.1: Mininet environment

1.1.1: Overview of mininet environment

We run mininet with the command: sudo mn --topo single,3

For delay-bandwidth product: sudo mn --topo single,3 --link tc,bw=x,delay=1ms



1.1.2: h1 as server

Server, runs HTTP 1.0/1.1, uses fork, and writes dummy message to stderr and redirected to /dev/null. In real-world server applications, server would do much more things than we do in the assignment. Thus, it is much easier to saturate the server if we let it write dummy messages to stderr and redirect those messages to /dev/null. We do this because we want to keep a connection longer in 1.0. Moreover, server write dummy message only when we want it to do so. In other words, client can choose to saturate server if it tries to get a specific file, then server would write dummy messages for that file. Other clients would not encounter this by getting other files.

1.1.3: h2 as experimental client

We focus our experiment on h2, so h2 will show some useful information we instrumented in client.c when h1 is in saturated state. For example, we will calculate and report average connection time and average transmission time when running h2.

1.1.4: h3 as saturating client

h3, forks multiple child processes which open HTTP/1.1 connections to h1, is mainly used to saturate the server. Thus, while we do the experiment on h2, what h3 does is continuously connecting to h1 and receiving specific file from h1.

Saturation

1: Definition

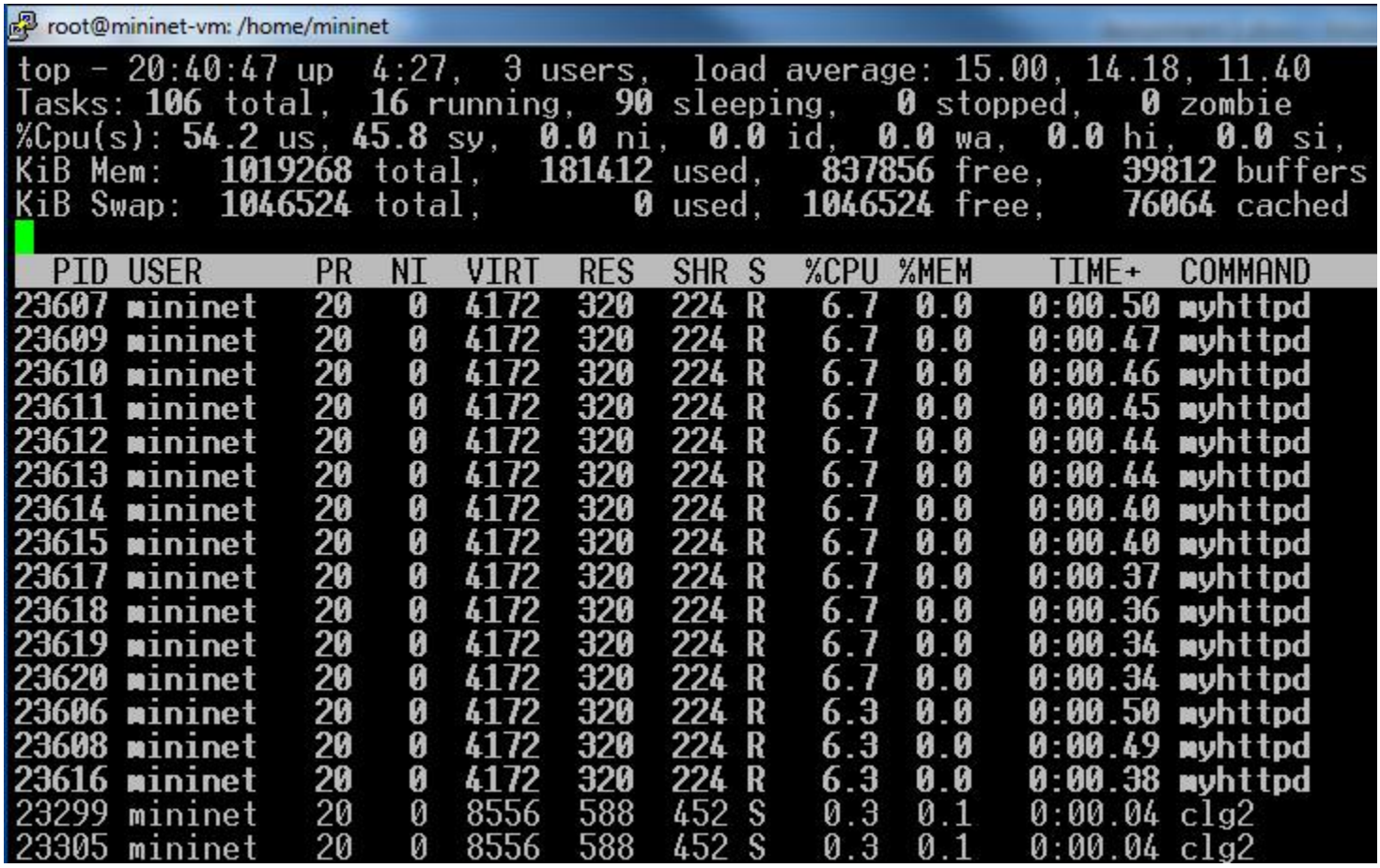
Saturation here means server is over-loaded and its CPU consumption is near 100%. Because it's hard to saturate server on mininet, we do this in mininet by using h3 to run saturate-purpose client and keep saturating h1. Then we can do more experiments by running experiment-purpose client on h2 when h1 is saturated.

2: Non-saturated server

In mininet, h3 cannot saturate the server if we create less than 3 child processes.

3: Saturated server

In mininet, h3 can keep saturating the server when client creates more than 15 child processes. At first, server's CPU consumption may be close to 99%, but it may drop to lower value when using HTTP/1.0 because all connections may close at same time. Thus, we choose to let h3 run HTTP/1.1 because it will not close the connection as long as there is data to get. And We can see from the following figure that that totally 15 myhttpd processes consume 99.3% CPU time, this means the server has been saturated by client. Although server run as HTTP/1.0 would close the connection after transmission, h3 can still keep saturating it after running for few minutes.

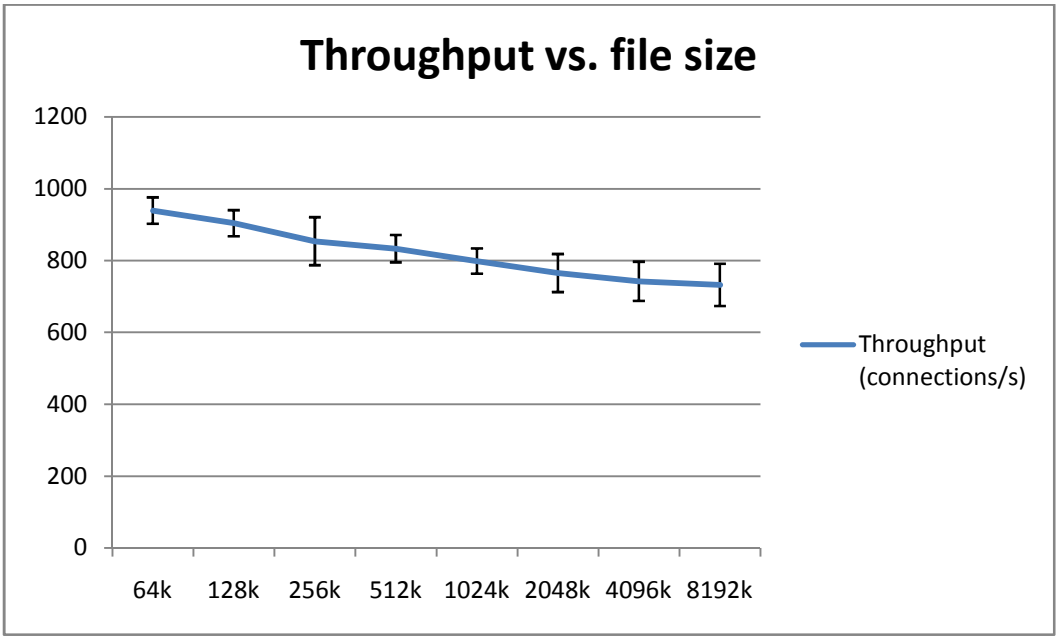

root@mininet-vm: /home/mininet
top - 20:40:47 up 4:27, 3 users, load average: 15.00, 14.18, 11.40
Tasks: 106 total, 16 running, 90 sleeping, 0 stopped, 0 zombie
%Cpu(s): 54.2 us, 45.8 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si,
KiB Mem: 1019268 total, 181412 used, 837856 free, 39812 buffers
KiB Swap: 1046524 total, 0 used, 1046524 free, 76064 cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
23607 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.50 myhttpd
23609 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.47 myhttpd
23610 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.46 myhttpd
23611 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.45 myhttpd
23612 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.44 myhttpd
23613 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.44 myhttpd
23614 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.40 myhttpd
23615 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.40 myhttpd
23617 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.37 myhttpd
23618 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.36 myhttpd
23619 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.34 myhttpd
23620 mininet 20 0 4172 320 224 R 6.7 0.0 0:00.34 myhttpd
23606 mininet 20 0 4172 320 224 R 6.3 0.0 0:00.50 myhttpd
23608 mininet 20 0 4172 320 224 R 6.3 0.0 0:00.49 myhttpd
23616 mininet 20 0 4172 320 224 R 6.3 0.0 0:00.38 myhttpd
23299 mininet 20 0 8556 588 452 S 0.3 0.1 0:00.04 clg2
23305 mininet 20 0 8556 588 452 S 0.3 0.1 0:00.04 clg2

Graphical Result

All four experiments are done in mininet. h1, h2 and h3 run as we told in saturation part. h1 runs as server, h2 is used to do real experiment and h3 keep saturating h1. Each data is being collected 20 times then mean value and standard deviation is gotten.

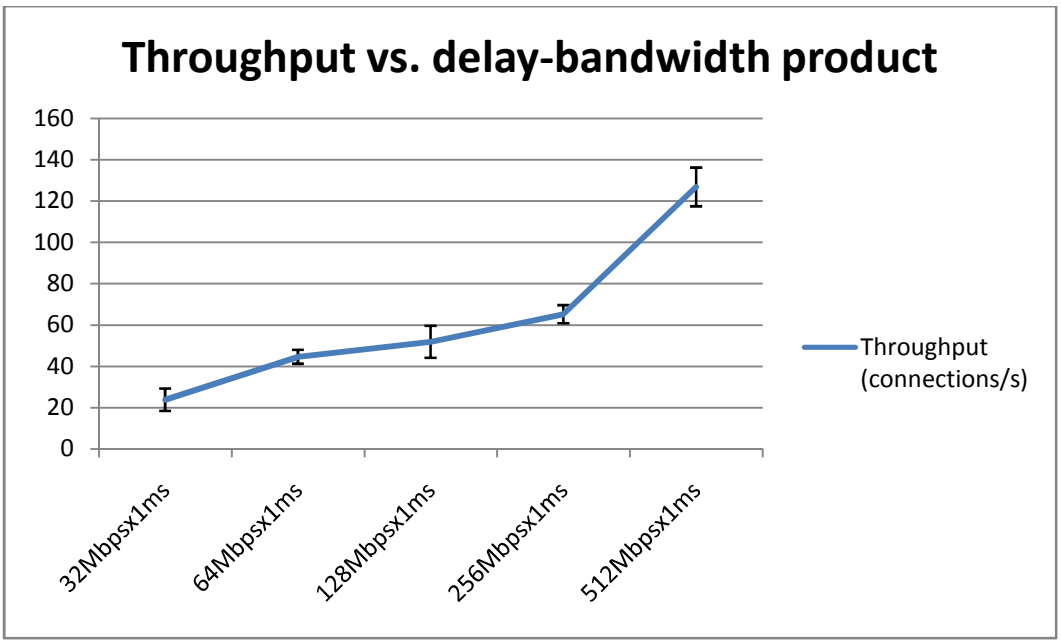
1: Throughput vs. file size (for HTTP/1.0)

We ran 8 experiments with file size from 64KB to 8192KB. In each experiment, h2 forks 5 child processes to download a specific-size file for totally 100 times. Thus the average result is based on the information of each child and calculated by the parent process. We can observe that when server is in saturated state, the number of connections it can serve is reversely-proportional to the file size.



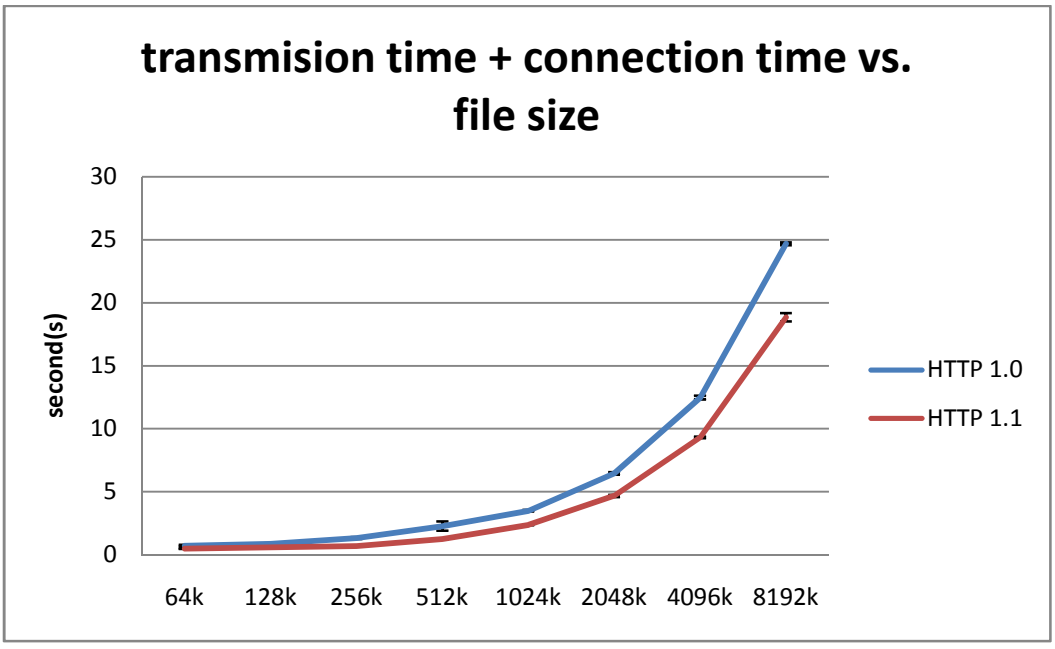
2: Throughput vs. delay-bandwidth product (for HTTP/1.0)

We cannot set delay too high because the server process will be killed during the experiment. Thus, we set delay to 1ms and change the bandwidth. The experimental result is showed in the following. h2 creates five child processes and download a 512KB file for totally 100 times and see the result. Obviously, as we increase the bandwidth, the throughput also increases.



3: Transmission and connection time vs. file size (HTTP 1.0/1.1)

In this experiment, h1 and h2 run in HTTP 1.0 and 1.1 for blue line and red line respectively. h2 creates 10 child processes to download each specific size file for totally 100 times from h1. Observe the following figure, it is obvious that HTTP 1.1 can easily beat HTTP 1.0 if they create the same number of child process because HTTP 1.1 require much less connections than HTTP 1.0. Furthermore, the larger the file size, the bigger the difference of total time. We know that the throughput would decreases when file size increases for HTTP 1.0 according to experiment 1. In other words, when throughput decreases, the connection time increases.



4: Transmission and connection time vs. delay-bandwidth product (HTTP 1.0/1.1)

In this experiment, h2 creates 5 child processes and get a 512KB file for totally 100 times under different delay-bandwidth product. Obviously, HTTP 1.1's total time also beats HTTP 1.0's total time. The reason is that HTTP 1.1 requires much less connections time than HTTP 1.0.