Despina Patronas
CSE 130

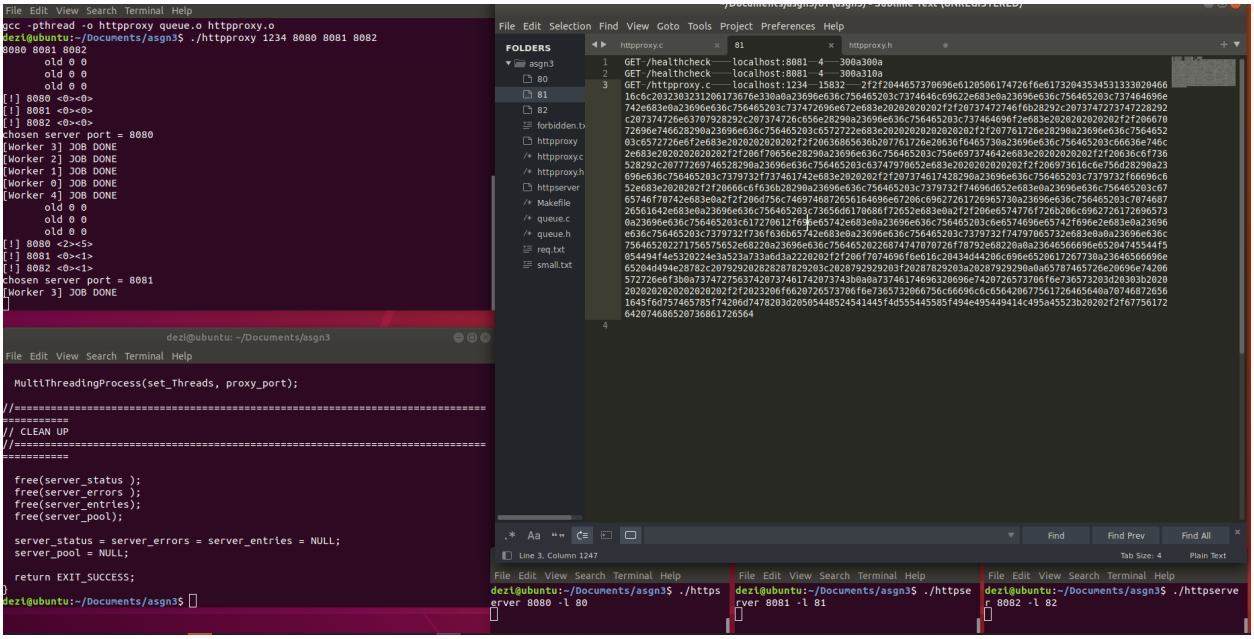
Write Up Document Assignment 3: httpproxy

Testing

httpproxy, should be similar to an http protocol server.

i. When Client sends curl requests (HEAD) to *httpproxy*,
httpproxy should perform the operation and respond back to client properly.

Testing Code for the spec:
Tests were done locally ..



Using various Server Terminals

Using a Proxy Terminal

Using a Client Terminal:
various curl commands for *httpproxy* which
try to fulfill requests for files in proxy / server directory:

<u>Proxy Responds</u>

1) request for !GET curl commands
2) requests which is invalid
3) requests when all servers are down

Proxy Handles:
4) requests where server healthcheck returns a bad status
5) requests when a server goes down which would have been the optimal server to port forward

6) Checking port forwarding is done correctly for load balancing scheme

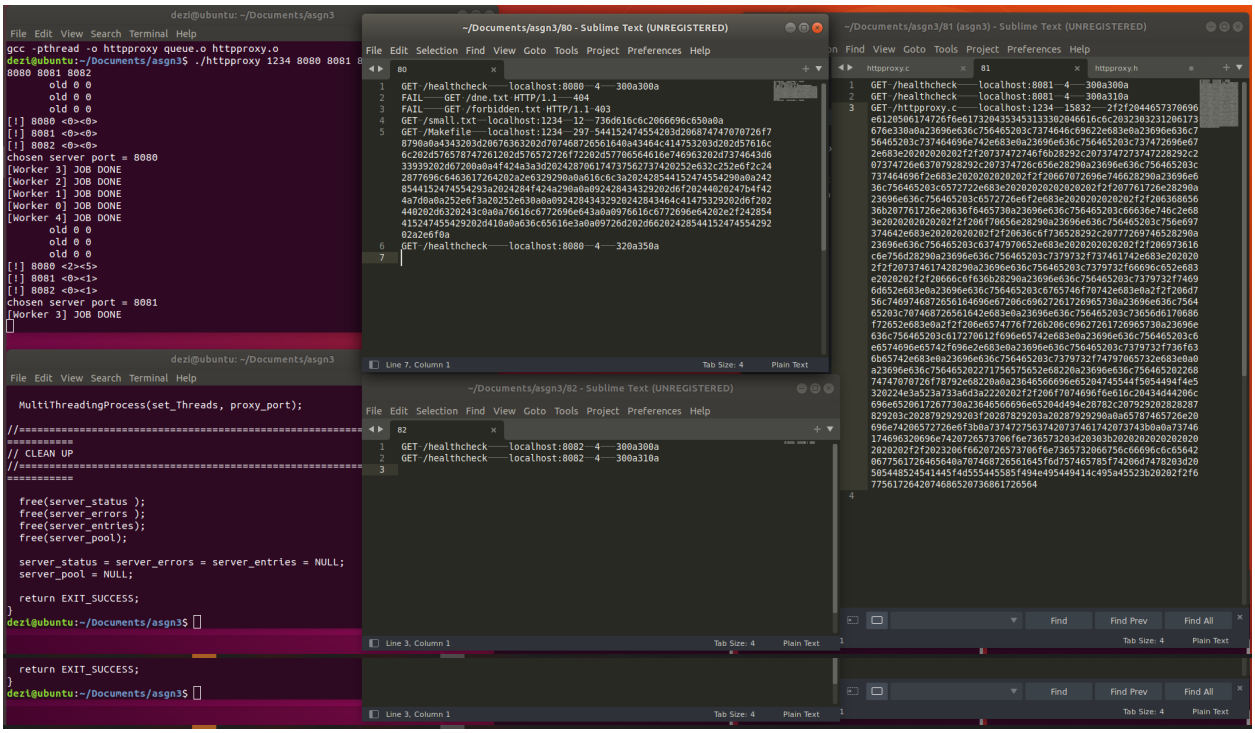
<u>Server Responds</u>

7) request for files that don't exist
8) request for files which are forbidden

Compares to expected outputs from *server*.
Each Request should run in independent threads on server
Healthchecks are done in dispatcher thread

Example Client Sends:

1. curl -T small.txt localhost:1234/small.txt
2. curl localhost:1234/-badrequest.txt
3. curl localhost:1234/dne.txt
4. curl localhost:1234/forbidden.txt
5. curl localhost:1234/small.txt
6. curl localhost:1234/Makefile



For Assignment 3, please answer the following questions:

1) **For this assignment, your proxy distributed load based on number of requests the servers had already serviced, and how many failed.**
   A more realistic implementation would consider performance attributes from the machine running the server. Why was this not used for this assignment?

   The performance balancing was not used because we are not able to easily determine the performance of the server due to factors we can't control. Since the server may be receiving external requests, the server speed is variable as opposed to a server which is dedicated to our proxy server.

   To implement a performance-based approach on a server which takes various requests from any connecting client or server, would require us to compute the performance / request and consider the type of request and the size of transactions in the log.

   It is not straightforward and therefore, it has been omitted.

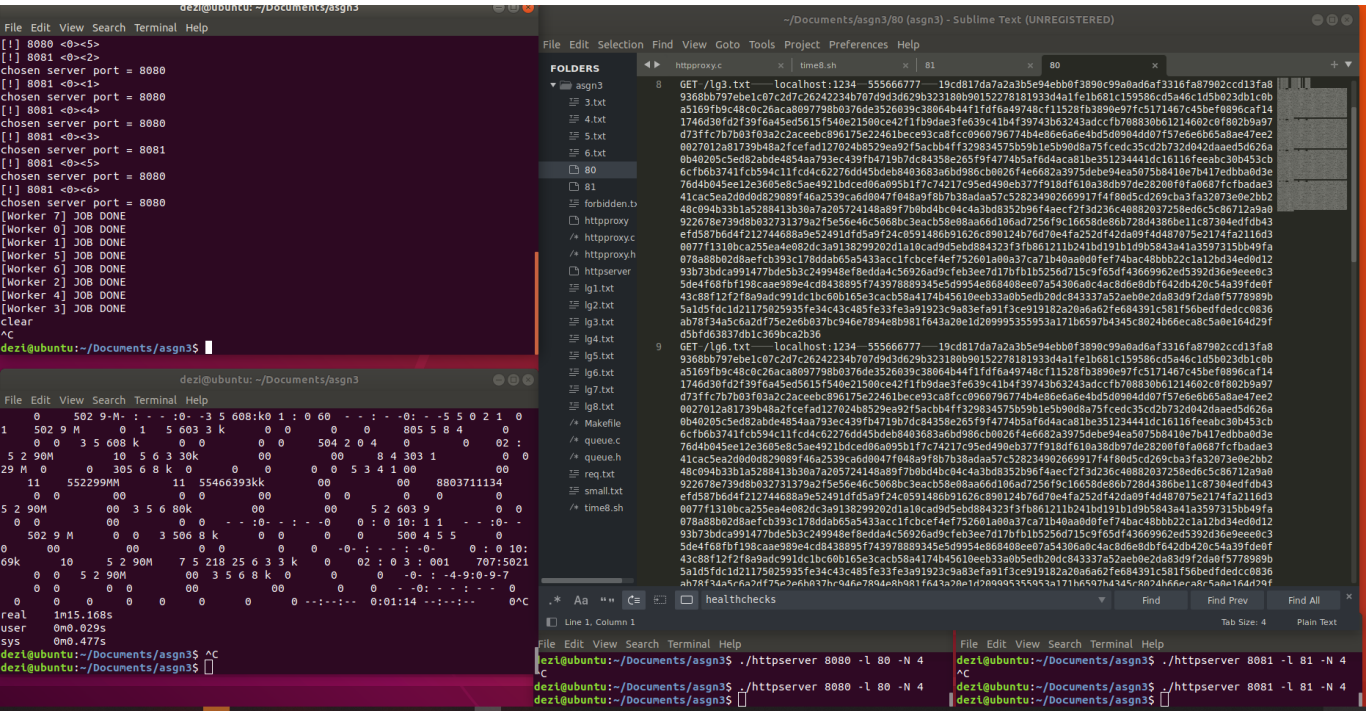
2) **Using the provided httpserver, do the following:**

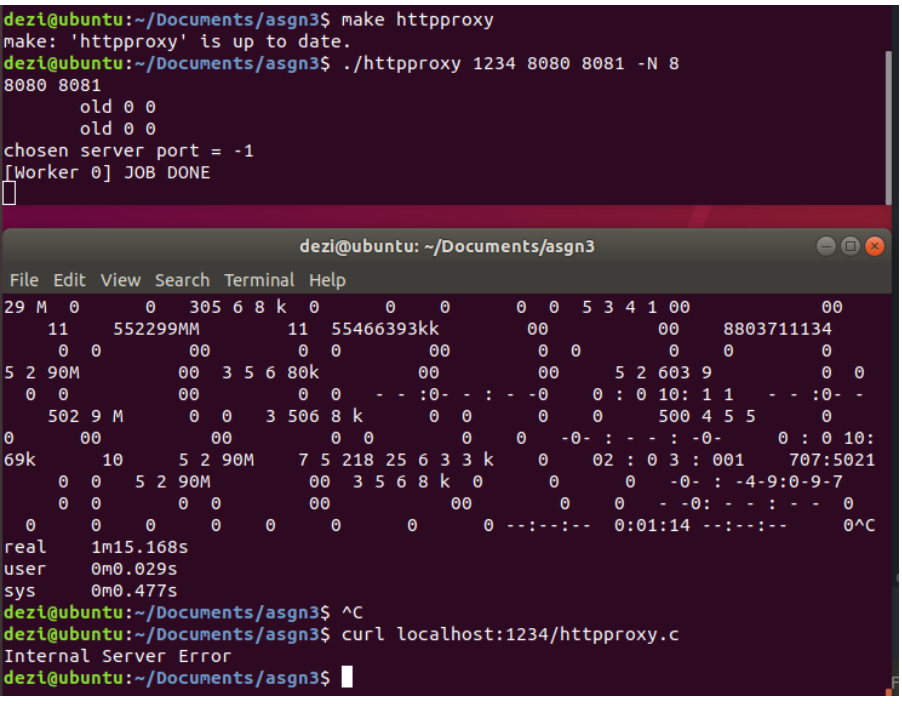
– Place eight different large files in a single directory. The files should be around 400 MiB long, adjust according to your computer's capacity.
– Start two instances of your httpserver in that directory with four worker threads for each.
– Start your httpproxy with the port numbers of the two running servers and maximum of eight connections and disable the cache.
– Start eight separate instances of the client at the same time, one GETting each of the files and measure
(using time(1)) how long it takes to get the files. The best way to do this is to write a simple shell script (command file) that starts eight copies of the client program in the background, by using & at the end.

This took two attempts. One attempt tried to handle binary files, while the other handled text files…

Attempt 1: (8) Binary Files 400mb

My proxy doesn't crash, but the servers both end up crashing and curl looks corrupted.

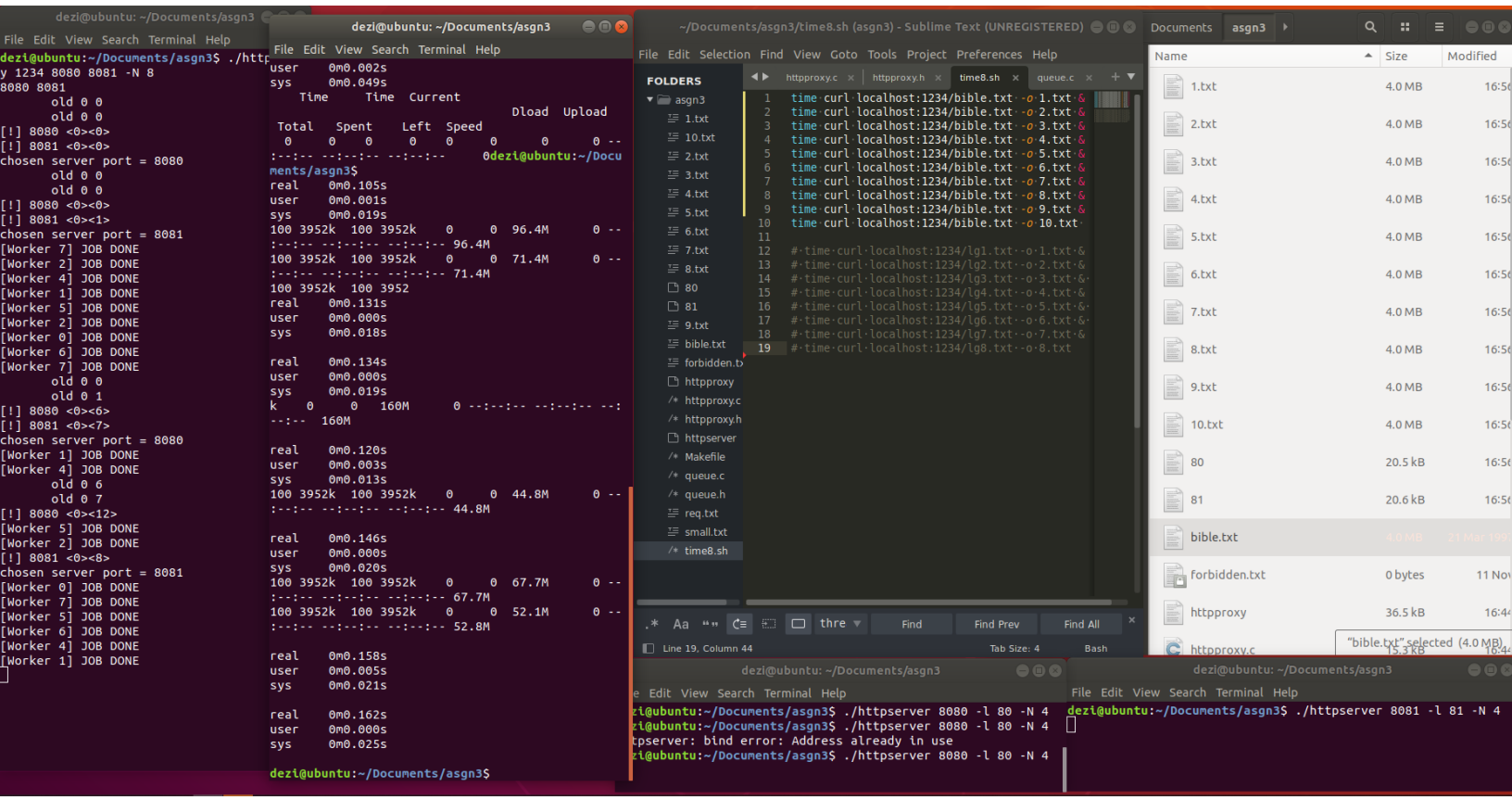




Appears some of the files were able to be processed while some were not. Both servers failed. Trying to process the binary files. Surprisingly my proxy was not able to send the internal server error response back like it should have when all servers are offline.

This experiment did not go well, so I will try it with regular text files instead.

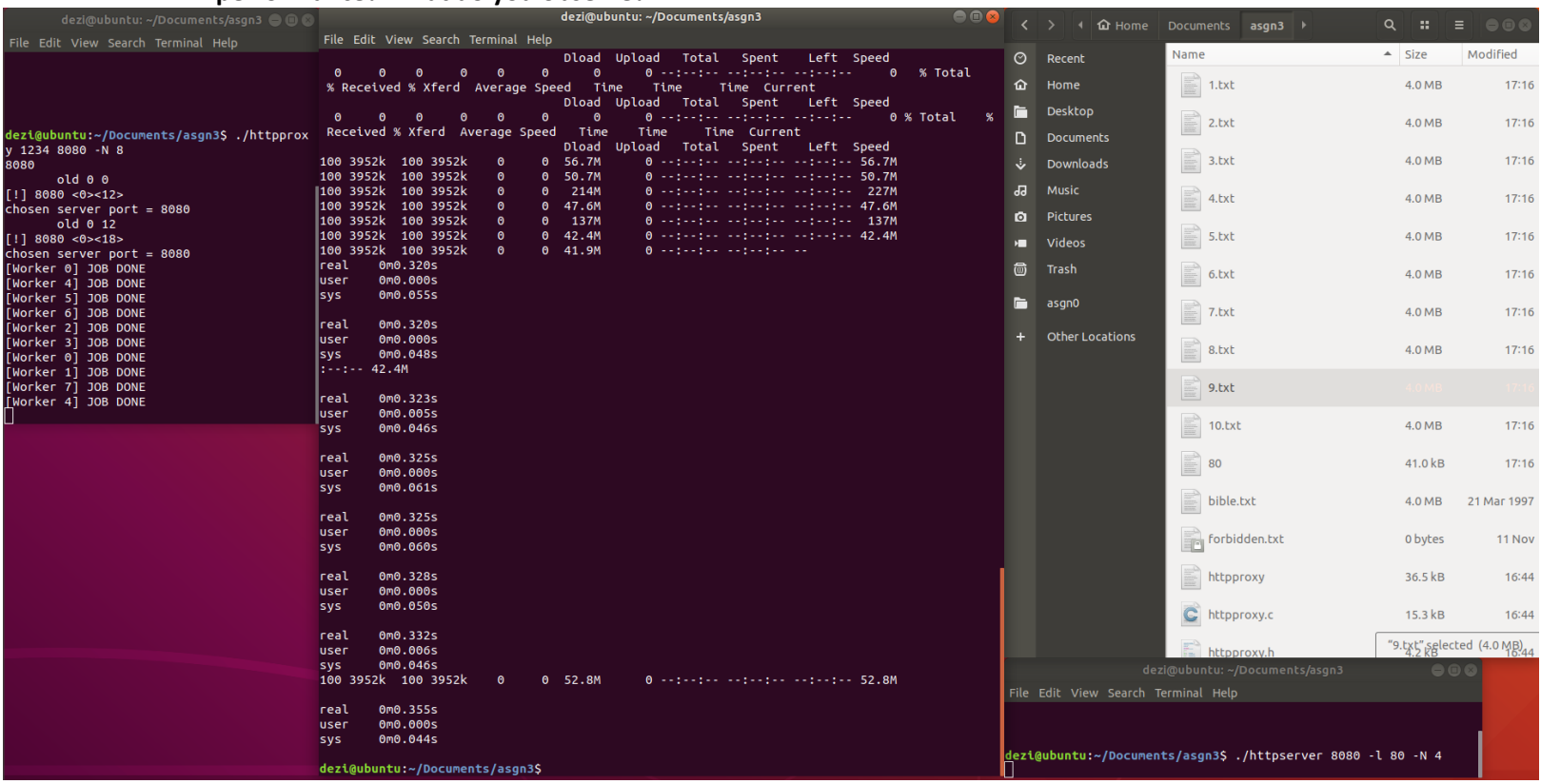
This test was successful.



Processing 8 text files 4.0 MiB Concurrently
On Proxy with 8 Threads
Using 2 Servers with 4 Threads each

Fastest time:		**Longest time:**	
real	0m0.071s	real	0m0.162s
user	0m0.000s	user	0m0.000s
sys	0m0.014s	sys	0m0.025s

3) Repeat the same experiment, but turn off one of the servers. Is there any difference in performance? What do you observe?



The single server, gave times that are about 2x slower than the previous experiment. This makes sense since only one server can be replying to the messages sent by the proxy.

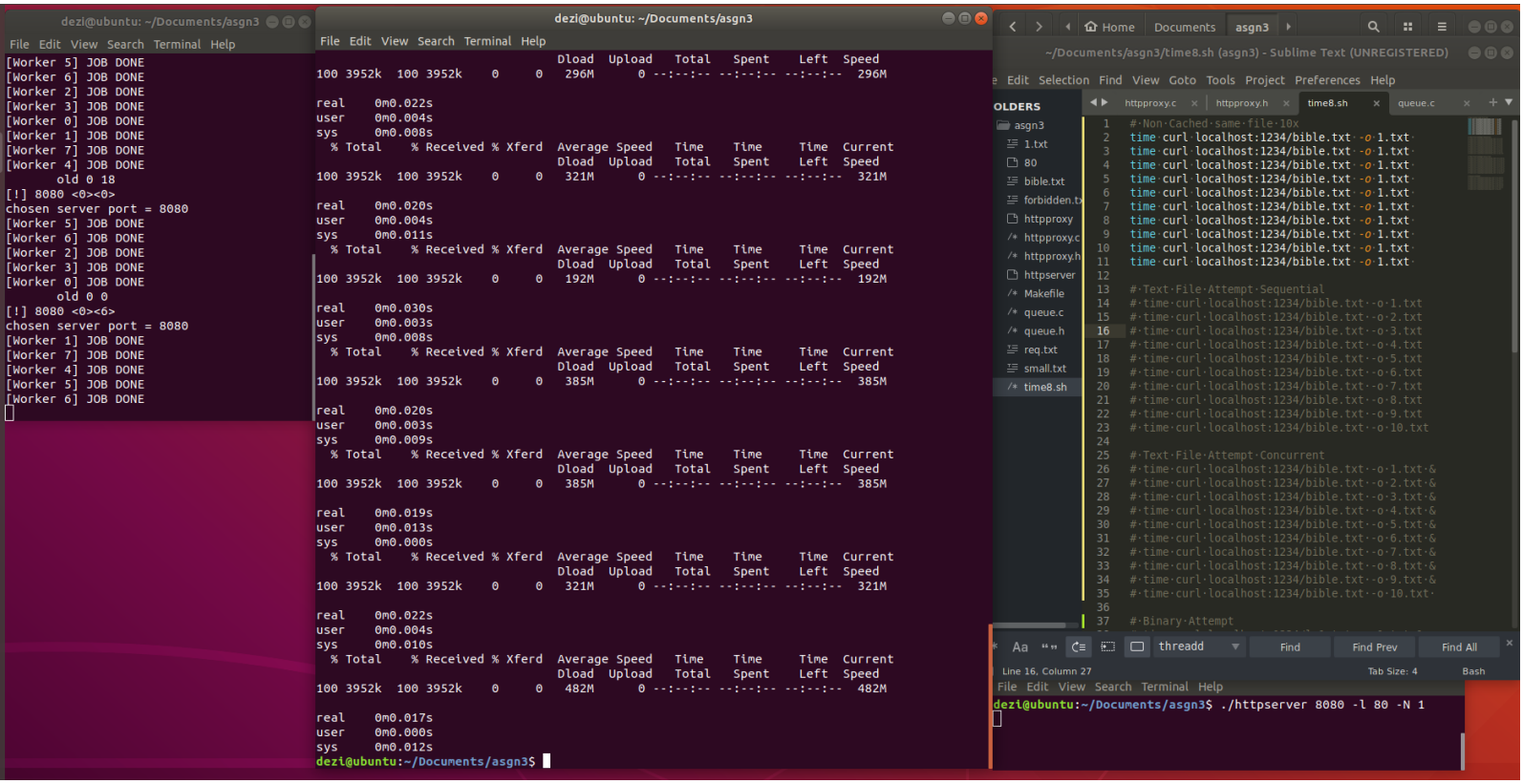
Processing 8 text files 4.0 MiB Concurrently
on Proxy with 8 Threads
Using 1 Servers with 4 Threads

Fastest time:		**Longest time:**	
real	0m0.273s	real	0m0.355s
user	0m0.007s	user	0m0.000s
sys	0m0.047s	sys	0m0.044s

4) Using one of the files that you created for the previous question, start the provided server with only one thread, then do the following:

– Start your proxy without caching.

– Request the file ten times. How long does it take?



Processing 8 text files 4.0 MiB Sequentially
on Proxy with 5 Threads
Using 1 Servers with 1 Threads

Fastest time:		**Longest time:**	
real	0m0.016s	real	0m0.020s
user	0m0.004s	user	0m0.003s
sys	0m0.008s	sys	0m0.008s

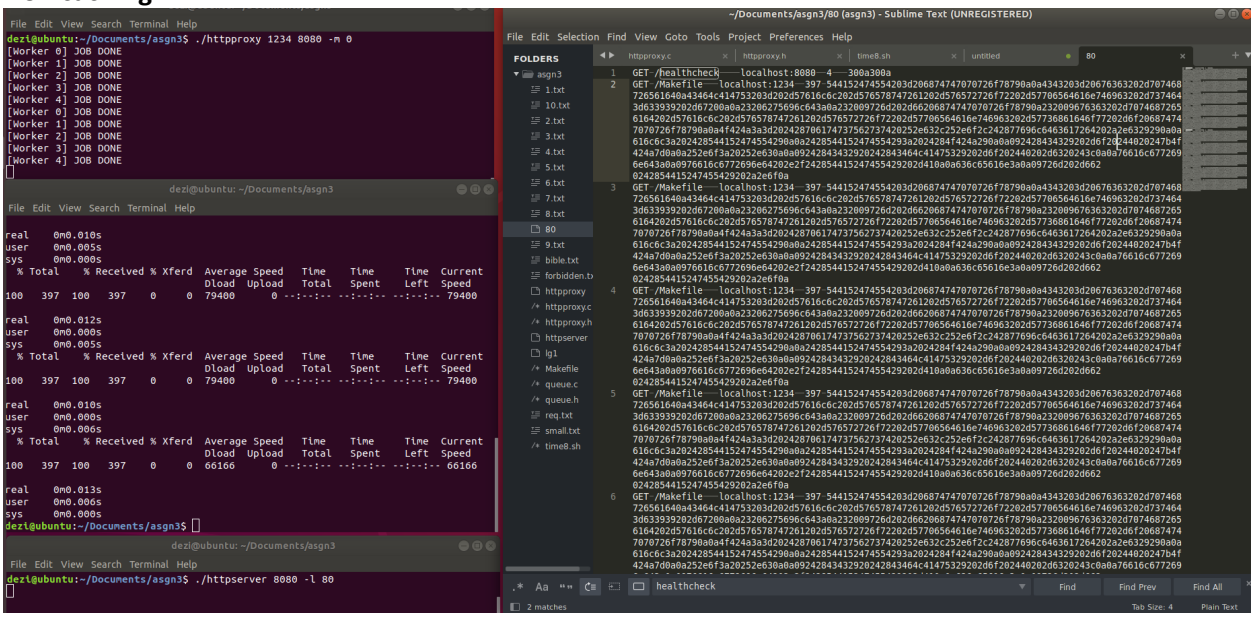
Now stop your proxy and start again, this time with caching configured so it can store the selected file.

5) Request the same file ten times again. How long does it take now?

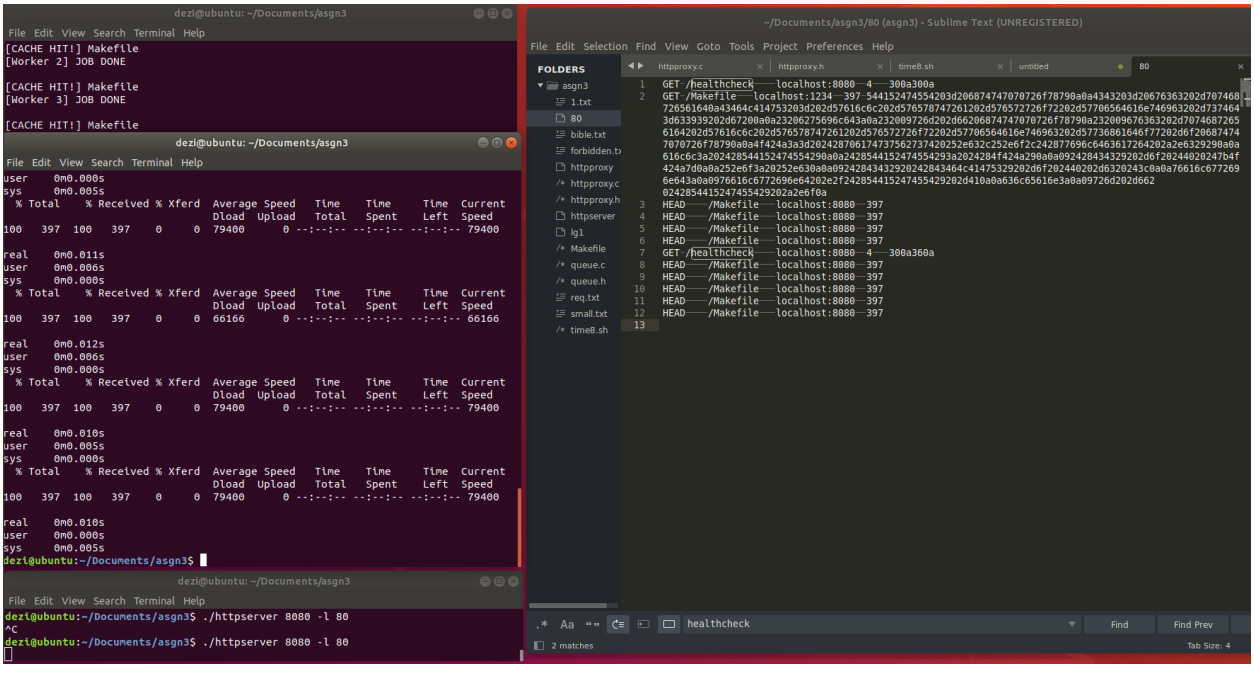
```
dezi@ubuntu:~/Documents/asgn3$ ./httpproxy 1235 8080 -m 4047392
Killed
```

Comparing a smaller file..

Non caching:

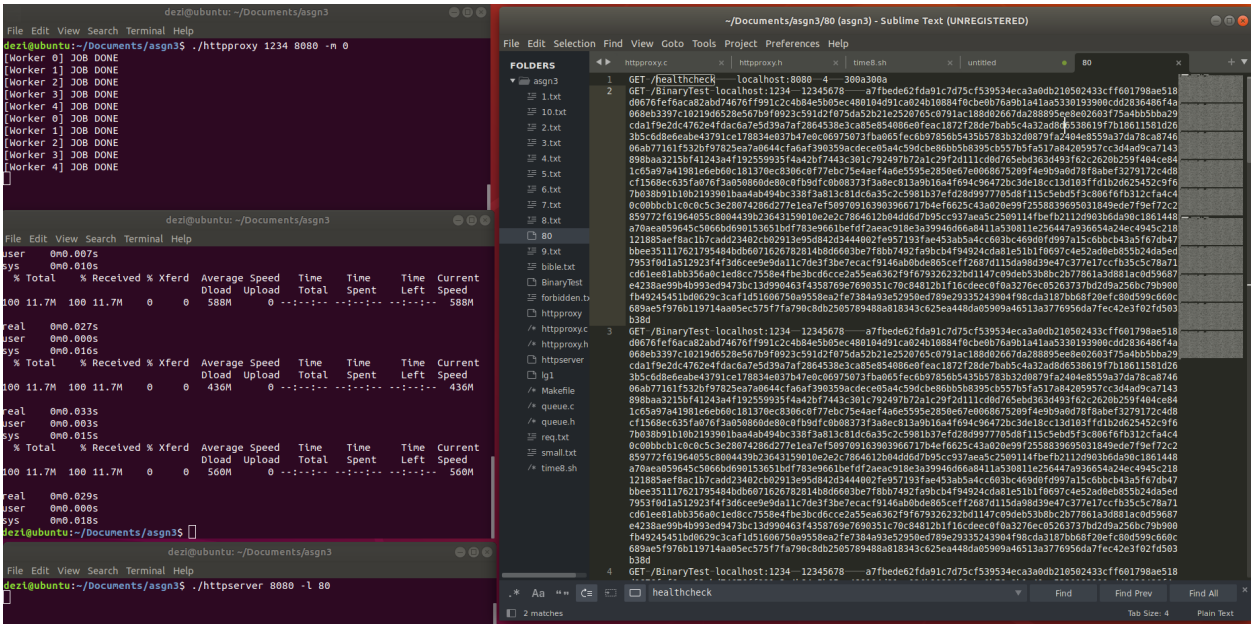


Looks very good almost instantaneous results. It is a small file. Only one file is generated
Seems to have shaved off 0.001 to 0.002 seconds.



Same experiment but trying with a 12345678 byte binary file instead..

Non Cached:



The 12mb binary file has an average tome of 0.027

Fastest time:
real 0m0.026s

Longest time:
real 0m0.033s

Cached

Unfortunately, the server seems to fail during transfer. Very hard to test because it causes my program which is trying to memcpy data over to fail and seg fault. With the small files it works.

I tried to run the program with valgrind and it killed my process for:

valgrind ./httpproxy 1234 8080 -m 12345678

I tried valgrind with the caching the small Makefile example above..

```
Process terminating with default action of signal 2 (SIGINT)
   at 0x4E4F7C7: accept (accept.c:26)
   by 0x10B6E3: MultiThreadingProcess (httpproxy.c:612)
   by 0x10BB0A: main (httpproxy.c:706)

HEAP SUMMARY:
    in use at exit: 37,931,934 bytes in 39 blocks
  total heap usage: 97 allocs, 58 frees, 38,026,047 bytes allocated

LEAK SUMMARY:
   definitely lost: 64 bytes in 2 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 1,360 bytes in 5 blocks
   still reachable: 37,930,510 bytes in 32 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
Use --track-origins=yes to see where uninitialised values come from
ERROR SUMMARY: 6 errors from 4 contexts (suppressed: 0 from 0)
```

noticed there is no memory leaks in the caching of small files.

```
Syscall param write(buf) points to uninitialised byte(s)
   at 0x4E4F371: write (write.c:27)
   by 0x4E4E358: sem_open (sem_open.c:269)
   by 0x109719: make_semaphore (queue.c:65)
   by 0x1097A6: queue_init (queue.c:83)
   by 0x10B5E1: MultiThreadingProcess (httpproxy.c:596)
   by 0x10BB0A: main (httpproxy.c:706)
 Address 0x1ffefffdac is on thread 1's stack
 in frame #1, created by sem_open (sem_open.c:141)

Syscall param write(buf) points to uninitialised byte(s)
   at 0x4E4F371: write (write.c:27)
   by 0x4E4E358: sem_open (sem_open.c:269)
   by 0x109719: make_semaphore (queue.c:65)
   by 0x1097B7: queue_init (queue.c:84)
   by 0x10B5E1: MultiThreadingProcess (httpproxy.c:596)
   by 0x10BB0A: main (httpproxy.c:706)
 Address 0x1ffefffdac is on thread 1's stack
 in frame #1, created by sem_open (sem_open.c:141)
```

Note: the leftover bytes coming from the semaphores doesn't clean up well… Have had this 64 bytes issue since I began the project carried over from asgn2.

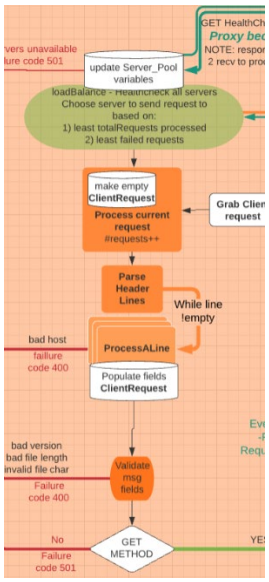
6) What did you learn about system design from this class? In particular, describe how each of the basic techniques (abstraction, layering, hierarchy, and modularity) helped you by simplifying your design, making it more efficient, or making it easier to design.

Abstraction –

Having very generic functions help increase abstraction. Tolerant of inputs and strict on outputs ie: making a file descriptor be defined as anything from a client to a server, or even a file or stdout. They are all treated similarly. The result is that an operation will take place on that file descriptor no matter what it points to. Very powerful technique to reducing repetition and extraneous bugs caused by unnecessary code that has the same functionality but used in a different part of the code. All buffer transfer logic remained consistent throughout projects and it was interesting to see how a function could be used in both instances of communication server<->client or GET<->PUT. Very similar functionality but with roles being reversed.

Layering –

Learned to make a design more linear by layering modules in order to reduce the interactions. This kind of control flow makes it easy to detect when a portion breaks and cannot progress in the line of modules. This kind of module organization makes the program easier to trace as well.



Hierarchy –

Using structures and enumeration to produce hierarchical organization. Threads are organized In structures which contain information about the threads.
Using structures nested with enumerated or defined structures make it easy to store, and contain information which can be defined in a fault tolerant way.

Modularity –

learned to group operations better into functions and make functions more GENERIC to produce functionality in broad terms. When there is a logical jump in flow, it is best to put that code in a function to isolate it from other operations. This way, when a block of code is misbehaving, one can ascertain the issue lies in the parameters passed in or the logic within the code which gives an incorrect output or outcome. It's easier to narrow down bugs using modularity since it provides a indication to the piece of logic that has failed based on the function that performed it.