Jesi Merrick A20264903
Ron Pyka A20250364
Dylan Boliske A20252450

Test Cases and Performance

Our program was tested for several cases, and below are a table of known working cases, and a table for known not-working cases. We have also included the performance results at the bottom.

Most of our test files were generated from random data using dd. example:
`dd if=/dev/urandom of=test1.txt bs=1K count=1`
these were verified to match on both ends using both word count (`wc -c test1.txt`) and the MD5 sum (`md5sum test1.txt` on linux or `md5 test1.txt` on OSX.)

Working tests:

Test	**Result**
Client can launch, connect to server, and exit	client is able to successfully connect to the server on launch, and the exit command kills running connections and exits gracefully.
Client can add a file to the server	client is able to successfully add a file to the server.
Client can connect to peer and transfer file	client is able to successfully request a file from another client and receive the correct file with the correct contents.
Client can register all files to the server	client is able to register all of its files, not including the application files, to the server
Server can handle multiple client connections	The server is able to handle receiving and sending information to multiple clients connected at the same time.
Client can handle multiple client connections	A client is able to handle receiving and sending information to multiple other clients connected at the same time.

Non-working tests:

Test	Result
Client receives list of IPs for a file held by multiple clients	The correct number of IPs return, but all are the same.
Client makes a change to a file listed on server	The client does not check to see if the files have been changed.
Client can remove a file from the server	The server seg faults when the client requests to remove the last file stored in the indexing server.

Performance results:

For several trials of the client and server on the same machine, (running through the local network instead of the loopback, and sharing the same port for the traffic,) 1000 sequential requests for a file took an average of 45.995 ms to complete all 1000. From a separate machine, 1000 sequential requests took an average of 1515.62 ms.

When run on two machines, the average time for machine 1 (the same machine the server ran on) was 52.174 ms and the average time for machine 2 (the same machine used for the separate machine test) was 1031.04ms

The table for all these trials is shown below (all times in ms):

Times from same machine (1) over network	Times from separate machine (2)	Times for machine 1 (2 machines simultaneously)	Times for machine 2 (2 machines simultaneously)
46.558	1720.04	59.692	869.975
47.327	1741.99	45.222	889.811
44.702	1916.80	50.873	1187.44
46.941	1314.63	53.530	1113.54
44.446	884.647	51.552	1094.42
average: 45.995	average: 1515.62	average: 52.174	average: 1031.04

The reason the times were so short on machine 1 is either that the machine recognized that it was connecting to itself and was able to bypass the network, or being on a wired network as opposed to wireless gave it a speed advantage. We are not sure why the times for machine 2 accessing the server at the same time as machine 1 were shorter. It might simply be chance, since the times still vary quite a bit for machine 2.

To test the timing, we added the following code to the client:

```
#include <sys/times.h>
#include <sys/time.h>
#include <time.h>

...

struct timeval etstart, etstop;
struct timezone tzdummy;
unsigned long long usecstart, usecstop;

...

gettimeofday(&etstart, &tzdummy);
for(i=0;i<1000;i++){
//send command and receive reply
}
gettimeofday(&etstop, &tzdummy);
usecstart = (unsigned long long)etstart.tv_sec * 1000000 +
etstart.tv_usec;
usecstop = (unsigned long long)etstop.tv_sec * 1000000 +
etstop.tv_usec;

printf("Elapsed time: %g ms\n", (float)(usecstop -
usecstart)/(float)1000);
```