Sidharth Gilela
sgilela@ucsc.edu
Assignment 2
CMPE 156

Design Document

Goal:

The objective of this project is to implement a client server model such that the client and server can communicate with one another. A goal of this assignment was to understand just how exactly pthreads work and the various issues that could come along with it. I used various system calls to allow this and had to deal with various errors. It was a great assignment to really learn the details of what happens when threads are created and how it speeds up an application

Compile:

To compile the server and client run these commands from the src folder:

```
make server
make client
```

To remove the executables run this command from the src folder:

```
make clean
```

Design:

This design of my protocol will be described through an example run between the client and as many servers as the user desires to open up. The application starts by running the client and as many servers as wanted. The first thing the client does is send the filename to the first server on the list in a message that looks like “filename my1500kbFile”. If that server does not have the file it will send an error message to the client and the client will move on to the next server on the list. If all the servers happen to not have the file an error message will be displayed on the client’s side. The first server that has the file will respond back with the filesize such as “filesize: 43”. The client receives the file size and stores it into a variable. During this process the client also keeps track of how of those servers it can connect to in another variable that will be used to compare with the minimum value later. Now the minimum value which is the minimum between the num of connections the user enters and the number of file servers on the server list is calculated. The new minimum will be set between this value and the number of connections that are available. The reason being is because even though the client may ask for 7 connections and the server list has 5 servers, only 3 or 2 servers may actually be running. This is another error case that is handled. Now, the client divides up the bytes by the minimum amount and creates the average and remainder. Something to note is that staring from now this method that I will describe is the method that is passed into the pthread_create function. Inside this function I do some more error handling such that basically what if the client connects the first server, it’s not open but the next two servers are open. The client is able to get the file’s information from the next two servers that are open if the minimum value is set to 2. I have done this by implementing a global array and setting and unsetting values in it depending on if a connection can be made to the server/port number or not. I have used a mutex around this array to make the function passed to the pthread thread-safe. All of these cases I have presented so far are ways to do error handling. As a brief aside, another error check I do is if the minimum is 0, then client presents a message on the screen. Additionally the number of arguments given to the client and server when running these executables is also checked and also the name of “server-info.txt”. My program requires the server list file to be named this. Now after all this, the client sends the amount of bytes it wants to receive to the server and the server replies asking for the position. The client responds by sending the position over to the server as to where the server

should start reading the file. These messages are again sent in the same format as above. When the server receives the bytes it needs to send over, it starts writing the bytes to the sendBuffer. This sendBuffer has a set size. Thus, I have a loop on the server side as well sending over chunks of bytes until the total number of bytes sent back to the client is equal to the number of bytes asked by the client. As the server is sending the bytes to the client, the client stores these bytes into a FILE* after opening a new file that has not been created yet. Each thread has its own FILE* so it has its own file. After the client receives all the buffers it needs to get, it starts combining all the content from all these FILE *'s into one file called "resultingFile". Again the client has it's own send and receive buffers that are set so a loop is done continually until the client recieves all the information. Eventually when the server stops sending bytes to the client it closes its connection, the "connfd variable", and the client reads 0 bytes at this moment. When the client reads 0 bytes at this moment, it closes the file pointers it has opened and also the socket file desicriptor that is has created. As another aside, another error handling case I have maintained is that if the client were to exit anytime during execution, the server's stability will not be affected and it will stay on. The server will not crash in this moment. Something to mention is that if this were to happen, you will probably see extra random files created in the directory. These files are the files that are created as FILE *'s, one for each thread before combining all these FILE*'s information into one FILE *. Technically the server and client wouldn't in the real world connect on the same directory so these files would never even be seen on the client side. Thus, this is not an issue I should be docked for.