Purpose:

The purpose of this program is to create a client which can spawn multiple threads to speed up the rate at which it gathers files from a set of servers. All of the servers have the same files, so when the client wants to download a file, it divides the work equally among all of its threads (one for each server). The threads then download a chunk of the specified file and if any one of them fails, another threads takes over provided that there is one available.

Note:

For additional clarification on exact implementation, the author encourages you to examine the source code which provides clear and highly informative comments that explain fully all error-handling and structure.

Protocol:

The protocol for our client-server model is split into three steps. Check-In, File size, and Download.

In the **Check-In** step, the client spawns one thread for each connection specified in the server-info.txt file to see if it can reach the server. The server tries to open the file specified by the client and returns “Y” or “N” depending on whether it was successful.

```
// The Check-In step message format
string msg = "Do you have: " + fileName + "?";
```

In the **File size** step, the client selects one of the connections that passed the Check-In step to query the server for the size of the file. The server checks the size of the file and returns it to the client. We only need to contact one server since we know the servers are mirrored.

```
// The File size step message format
string msg = "Filesize of: " + fileName + "?";
```

In the **Download** step, the client determines the range of the file that each thread will request to download. Then, it initiates the download. On the server size, each server reads the portion of the file specified and sends it to the client piece by piece. The client checks if the download is complete. If not, then it will resolve the error by declaring the failed connection

invalid and then asking any remaining valid thread to finish the download. On the server side, there is no notion of failure or success in a download— it just sends what it is asked to.

```
// Format of the Download step message
string msg = "Send: " + to_string(amountToRead) + " bytes of: " + fileName + " starting from: " + to_string(startLoc) + ".";
```

Client:

Upon being executed, the client validates its arguments immediately by checking the number of arguments, the validity of the arguments. It reads in the list of server ips and ports from the specified server-info file (note that it doesn't necessarily have to be server-info.txt). It reads these into two vectors, "serverIPs" and "serverPorts". If the number of connections specified is more than the number of servers read in, the client reduces the number of connections to the number read in. Unless the arguments are obviously incorrect, the client begins to execute the protocol.

The client performs the **Check-In** step by dispatching a number of threads equal to the number of (IP, port) pairs passed in to perform the checkIn() function. It passes each thread its port and IP through a custom struct of type "threadArgs".

In checkIn(), the client thread attempts to establish a connection with the server it is told to. Then, it sends the message according to the protocol specified above. If it receives anything other than a "Y" or if at any point the client thread encounters an error with a system call, it returns -1. If everything is okay, it returns a 0.

Back in main, the client reads the return value of each thread that called checkIn() into an array of ints. The client loops through the array and if it finds a thread that returned -1, marks the corresponding element of "serverIPs" and "serverPorts" for deletion. Then the client deletes these marked elements from the vectors. If the number of valid (IP, port) pairs is 0 at this point, the client will fail, which handles any other possible errors with input to the program.

Next, the client completes the **File size** step by calling getFileSize and passing in one pair from "serverIPs" and "serverPorts". We only pass in one because this part of the process is not concurrent. Instead, since we know that all servers are mirrored copies of each other, and since we know that all of them have the file specified and our client can connect with them, we only query one server for the file size.

In getFileSize() we build the message specified by the format above and then send it to the client. We return the value the value to main.

Main then passes the valid list of serverIPs, serverPorts, number of valid connections, the file size, and the threads we allocated earlier into downloadStep().

In downloadStep(), we begin by calling calculateDownloadSize() which is a function which determines the size of the chunk that the threads will call. Note that in the case where the file size can not be exactly divided among all clients, then calculcateDownloadSize() gives the extra byte to the last thread.

Back in downloadStep(), we open the output file to store the data in (which we wait to do until now because we didn't know if we'd even get this far when the client first started). If for some reason the file could not be opened, the client dies and raises a fatal error. Next, we initialize the arguments for download() and then dispatch all of the threads to request the chunk assigned to them by calculateDownloadSize().

In download(), each thread is passed in a server (IP, port) pair, an offset, and an amount to read. We initialize the return values of the thread which will be useful in case of an error, and then we create a socket and connect to the server. The arguments are passed in by a custom threadArgs struct. Using a combination of two global variables (the file to write to and the file name) and the arguments passed in, we form and send the message specified for the download step to the server.

Finally, the client is ready to read from the server. The client begins to receive data and uses pwrite() to write to the output file without overwriting other threads. As it is reading in and writing data, it decrements the amount left to read and modifies the offset accordingly. If it reaches 0 bytes left to read, then it returns 0 to back to download. More interestingly, however, is when download() errors out (as is possible in case one or more servers crash). In this case, the client will return where it was reading, how much it had left to read, how much it has read so far, and finally a status code of -1. Note that like in the server's sendChunk(), download() does not have a concept of whether or not something has gone wrong. It just does what it is told and if an error comes up returns and lets someone else handle the problem.

In the second half of downloadStep(), we wait for all threads to return and then transfer the result of their returns to an array of custom threadResults structs. Then, we create a variable "totalRead" which tracks how much our threads have reported reading. While the totalRead is strictly less than the size of the file returned in the File size step, we loop over all the threads and perform different actions according to their status. If a thread has a status of 0, we set it to status 1 and add how much it read to the value of "totalRead". If a thread has a status of 1, we know it has already been counted, and so we will not accidentally add it to "totalRead" twice. If a thread

has a status of -1 (which we refer to as “bad threads”), we check if it was able to read any data at all. If it did, we add how much it read to “totalRead” and then find another thread that can finish downloading whatever the bad thread did not. We determine if a thread can cover for the bad thread by looping through all threads and checking for any threads with a status of 0 or 1. If we find such a thread, we set the bad thread to status 3, which means it will now be ignored forever. We also set the status code of the working thread to 2, to indicate that the working thread has decided to take over for the bad thread. Setting a thread to status code 2 is important in case multiple threads fail (we do not want to accidentally try to get thread A to cover for both thread B and thread C at the same time.) We set up the working thread to run download() based on where the bad thread left off (using as arguments what the bad thread returned).

If after looping through all threads we set a thread to 2, we indicate that all hope for finishing the download is not lost, and our program will not die. In the event that no thread what so ever was found to handle a bad thread, this means that all threads are marked as -1 or 3 the client has no available connections to download the file and will therefore die. Finally, we dispatch the valid threads to cover for the bad threads and then reenter the loop with the results of the valid threads to run through the loop again. In this way, even if we have 4 bad threads and only one good thread, the one good thread will make up for all the others, read only exactly as much as it needs to, and will not cause any problems.

Finally, once “totalRead” equals the size of the file returned by the File size step, we know that our download is complete and terminate our program with a status code of 0 to indicate success.

Server:

The server has a much simpler job than the client, and tends to let the client resolve all of its errors so as to prevent itself from crashing. The reason for this is that in a client server model, we always want the server to be able to continue to serve new clients even if something prevents it from serving one client.

To start with, the server validates the input arguments and sets itself up, exiting with status code 1 and giving a fatal error if it could not start for any reason.

Whenever the server receives a new connection, it checks what step of the protocol that the client is asking it to complete by calling checkStep(). checkStep() reads the first two bytes of the incoming message and uses it to determine what function to call. Note that in all of the

protocol messages, the first two bytes do not contain any information that the server needs to determine what to do (like a filename or an offset).

If the return value of checkStep() is 1, the server calls tryOpen(), the server equivalent of the **Check-In** step. Here, it tries to open the file. It determines what file to try to open by parsing the full message from the client. If the file is opened successfully, it sends the client a message saying “Y”. If it can’t open the file for any reason, it returns “N”. If the server encounters any other while in the process, it simply closes the connection and leaves it to the client to fix the problem.

If the return value of checkStep() is 2, the server calls getFileSize(), the server equivalent of the **File size** step. It uses the stat struct utility to determine the file size and sends the client the result. Again, if the result is nonsensical, the client will handle it, not the server. However, we expect the value to be reasonable because our server was able to open it in the Check-In step.

Finally, if the return value of checkStep() is 3, the server calls sendChunk, the server equivalent of the **Download** step. Here, the server knows nothing about what thread is calling it, or whether the thread calling it is trying to make up for the mistakes of another thread. It simply does what it’s told and reads from the file it is told to. After parsing the message to get the offset, the file name, and how much to read, the server will send the bytes it is told to piece by piece. As before, if anything goes wrong during this process, the server gives up and lets the client handle the error.

The server is much less complex than the client because it does not need to do any real error handling. This also means that the server is faster than the client to recover from errors and therefore has a higher throughput, which is what we prefer in a server.