CS 5600 Final Report

Team Members

Madeline Cameron
Xiao Deng
Matthew Lindner

Introduction

During the first half of our project, we met three times to discuss implementation and draft basic pseudocode. We spent our first meeting contrasting C and Python as possible implementation languages. After preparing prototypes in each, we quickly settled on C. During our second meeting, we planned which functionalities of both the server and client applications we should tackle first. Our last meeting was held the night before the midterm demo. We spent our time finalizing the required functionality for the demo.

Immediately after the midterm demo, the team added multithreaded capabilities to the server, and the ability to ignore comments in tracker files. The midterm report was completed shortly after.

During the time between the midterm and final demonstration, our group met a total of four times. We spent each meeting focusing on the following details: separating a file into chunks, joining chunks using C++.

The Thanksgiving holiday made collaboration difficult due to family obligations, and thus all work was completed on an individual basis.. What time we had was dedicated to building a library of chunking functions, the ability to load settings from a configuration file, and documentation.

For the Final demonstration, we were able to present a roughly 90% complete implementation: multithreading, TCP peer-server communication and (tracker) file exchange, TCP peer-peer communication, tracker file parsing, config file loading, MD5 calculation, file chunking, and file joining. The ability to request and send file chunks was unfortunately unfinished.

Team Roles

Madeline was lead author on the mid-term and final report. Also, Madeline implemented three functions for the client. One function was for passing a client's IP to connected clients, one was for capturing commands so that long if-else statements were not needed and finally a function to read a configuration file so that users wouldn't need to recompile the code to change their configuration. The first two functions were ultimately cut from the project for a combination of time reasons and superfluity of their functionality. Finally, she wrote all of the shell / .ex files. Madeline attended three out of four times the team met in the final half of the semester, she was out of town for one of the meetings.

Xiao retained his position as lead developer on the client. Xiao implemented a series of functionalities to support the client for most of the file parsing, construction, chunk data management as well as a test program for debugging all the individual functions. He also implemented the makefile and documentation on all the code he was involved in. Xiao attended every meeting in the final half of the semester. He also co-wrote portions of the mid-term and final reports.

Matthew also retained his position as lead developer on the server, implementing the networking code, multithreading code, preliminary tracker file handling (with the assistance of Xiao), server-client protocol messages, a MD5 function, and basic file transfer. He also co-wrote the client with Xiao, specifically the networking and multithreading code. Additionally, Matthew co-wrote portions of the midterm and final reports. Matthew attended every meeting in the final half of the semester.

Installation Guide

Code will run on any Unix / Linux computer as it is compiled C/C++. We recommend using the provided CS Unix installations (eg: csXXxcs213.managed.mst.edu) as they are easily accessible to all students and all needed compilers have been installed. The code will need to be compiled with the g++ compiler which is widely available via the package manager or pre-installed. After the g++ compiler is successfully installed / verified to be present, the code can be compiled with the makefile in the root directory. Makefile also includes clean argument for cleaning up all compiled files and demo file system structure.

Client:
make client
Server:
make server
Clean:
make clean

Running Guide

After the code is successfully compiled using the makefile, the necessary executables will be in the project root folder.

Server:

The server can be started with the command "./server.out" command. When starting the server, the user can specify the desired port via a command-line argument (ie ./server.out 3456), or by editing the first line in the configuration file (server.conf). If no argument is specified via the command-line, the server will read the port number (line 2), as well as max number of clients (line 3) and chunk size (line 4) from the configuration file.

The server can also optionally be executed via the "test_server.ex" script outlined in the final project specification handout.

Client:

Each client can be started with "./client.out". When starting a client, the user can specify: the mode ('0' for **SEED** or '1' for **DOWNLOAD**), the port on which it will seed, the client index (ie "client_5"), and how frequently the client should contact the server (<*updatetracker*> or <*REQ LIST*>). Much like the server, if no arguments are specified via the command-line, *server_port*, *max_client*, *chunk_size*, and *server_update_frequency* can be changed in the client configuration file (client.conf) .

An example of running a client that is in download mode, seeding on port 6699, has a client index of 5, and updates the server every 10 seconds: "./client.out 0 6699 5 10".

An example of running a client that is in seed mode, seeding on port 6700, has a client index of 4, and updates the server every 5 seconds: "./client.out 1 6700 4 5".

The client can also be tested with two of the scripts that were outlined in the final project specifications: t*est_client_rcv.ex* and *test_client_send.ex*.

Code Design

Server:

The server has three configurable variables, *server_port* (the port the server listens on), *max_client* (maximum number of concurrent connections) and *chunk_size* (the size in bytes of the chunks). All of these variables can be configured via the configuration file named *server.conf*, and *server_port* can be set via command line arguments.

In the server, there is a struct named *peer* which is for containing information about a client: the file descriptor messages will be sent over (*m_peer_socket*), its index in the clients array (*m_index*), a buffer for temporarily storing data (*m_buf[]*), a file pointer for opening tracker files (*m_file*), and its own thread for executing commands from connecting peers (*m_thread*). We store the peers in an array (*clients[]*).

When the server is initiated, it checks for a command line argument, expecting server port to be the first parameter. If a parameter is not found, it reads from the configuration file to get the port and other configurations. After establishing a TCP port to listen for connections, and initializing the *clients[]*" array, the server is then put into a loop which can exited with the keyboard signal Control + C. During this loop, the server listens for up to *max_client* number of connections. When a client tries to connect, the server first checks if there are any openings in the *clients[]* array (*findClientArrayOpening()*). The server then spins off a thread to another function to process the peer.

The *client_handler()* function is passed the index of the peer in the *clients[]* array which it will be processing. The function copies the data sent over the client's socket into a buffer and finds which command was requested.

The <createtracker> command first ensures the correct number of arguments were sent, and then checks whether or not the requested tracker file already exists. If it does not, the server creates the tracker file. If the tracker file for this file already exits, the server responds with an error message.

The <updatetracker> command checks to make sure that the tracker file already exists and appends the chunk info (ip, port, start byte, etc) to the bottom of the file.

The <req list> command first opens the tracker file directory, counts the number of files, copies the name, filesize and MD5 of each file and sends this information to the client.

The <get> command opens the requested tracker file, writes the contents to the client's socket and appends the MD5 (calculated by the *computeMD5()* function) of the tracker file to the end of the protocol footer.

After a command is processed, the server closes the client's socket and marks its index in the *clients[]* array as empty (the *m_peer_socket* value for the peer in that index is set to -1).

Once the loop is terminated (via a Control + C), the server closes it's listening socket, waits for any running thread to terminate (*pthread_join()*) and exits.

Client:

The client has four configurable variables, *server_port* (what port the tracker server is listening on), *max_clients* (maximum number of concurrent connections), *chunk_size* (how big the chunks of data are in bytes) and *server_update_frequency* (how often to update the server in seconds).

Exactly like the server, there is an array of *peers*, where each peer has its own: socket, array index, temporary buffer, file pointer and thread.

When a client is initiated, it checks for command line arguments. Mode ('0' for Seed, '1' for Download), what port it will be operating on, the client's desired index and how often it updates the server can all be set by command-line argument. Server port, max number of connected clients, chunk size and how often it updates the server can also be set in the configuration file. The function *setUpPeerArray()* executes exactly as described in the server section.

Depending on the value of the "mode" variable, two different branches (located inside of if statements) of execution can occur: seeding or downloading.

When in **SEED** mode, the client first establishes a TCP connection with the tracker server, and sends a <createtracker> command for the "picture-wallpaper.jpg" specified in the Final Demo Guidelines. The client then spins off a single thread (*client_handler()*) to listen/process connections. While listening for connections, the client sends the tracker server an <updatetracker> command every 10 seconds (or whatever time is specified in *server_update_frequency*) indicating that it is now sharing an additional 5% of its (20%) segment. Which 20% it is responsible for is calculated using the *client_i* parameter passed by command line argument.
The *client_handler()* function first establishes the client as a sort of server. It begins by establishing a TCP socket to listen for connections. Once a peer has connected (*accept()*), the client prints a message indicating to the user that a connection has been established. It is here where our team had planned to implement the transfer of file chunks, but

were unable to do to a lack of time.

When in **DOWNLOAD** mode, the client first connects to the tracker server. Every 5 seconds, the client sends a <req list> command, and searches the server's response for the "picture-wallpaper.jpg" tracker file. Once the server indicates someone is sharing this photo, we store "<GET picture-wallpaper.jpg.track>" in the buffer used to compare commands. This automatically initiates the download process (since *strcmp(buf, "<GET>")* will evaluate to true). The client will then re-connect to tracker server, and download the "picture-wallpaper.jpg" tracker file.

Next, the client spins off threads for downloading. The *download()* function will first parse the tracker file for a port number of another client currently sharing a chunk. Using this port number, the client will then connect to this peer. It is here where our team had planned to implement the request of and download of chunks, but were unable to do to lack of time.

Client_support:

To meet the requirement outlines in the final demo we have to take a different approach to our file handling structure and scheme. Thus Xiao has wrote the client_support files which contain a series of convenient methods and functions to carry out the workload on the client for file handle, I/O and chunking data management to and from a tracker file. The client_support has the following key structures and variables:

1. ***chunk_struct***, a structure that stores and tracks the ip address, port number, starting byte, ending byte and time stamp for a file chunk in memory.
2. ***tracked_file_info_struct***, a structure that stores file name, file size, description and md5 for a sharing file in memory.
3. ***segment_struct***, a structure that stores information regarding a file segment for clients in seed mode.
4. ***rtn_val***, an enum object that defines all the return values of the functions/methods in client_support.
5. ***tracked_file_info***, an instance of *tracked_file_info_struct* that act as a buffer of detail information of a sharing file in memory.
6. ***live_chunks***, a vector of *chunks_struct* objects that act as a table to manage chunks that is currently being shared in the P2P network for a particular file of interest, either created from a client in seed or requested by a client in download mode.
7. ***pending_chunks***, a vector of *chunks_struct* objects that holds all chunks on a pending list to be added to *live_chunks* to be shared, used only by a client in seed mode.
8. ***file_segment***, a vector of *segment_struct* objects that stores information for each segment of a tracker file, it is used to dynamically append percentile of chunks in the tracker file to the *live_chunks* vector to achieve a dynamic seeding environment per demo requirement.

The client_support has the following methods and functions:

1. ***findNetChunk(),*** finds the next chunk available in the *live_chunks* vector with a input combination of *starting_byte* and *ending_byte* of said chunk and returns the index of a matching chunk in the vector if it exists, and returns *NO_NEXT_CHUNK* if no match was found.
2. ***tracker_file_parser(),*** parses out an existing tracker file, read in the *file name*, *file size*, *description* and *md5* information of that file to the corresponding buffers. It also parses out all the chunks present in the tracker file and save them into *live_chunks* vector.
3. ***commitPendingChunks(),*** commit all the chunks in *pending_chunks* vector and copy them into the *live_chunks* vector, also appends all the pending chunks into a preformatted (with header of file information) existing tracker file, then clears out *pending_chunks* vector to get ready for future modifications.
4. ***appendChunk(),*** append/add an additional *chunks_struct* object to the end of *pending_chunks* vector.
5. ***clearPendingChunks(),*** clears out the *pending_chunks* vector, it now has a length of 0.
6. ***clearLiveChunks(),*** clears out the *live_chunks* vector, it now has a length of 0.
7. ***isLiveChunk(),*** checks if a chunk is in the *live_chunk* vector with a *chunks_struct* object as input, returns the index of the matching chunk in the *live_chunks* vector if it is found, *NOT_LIVE_CHUNK* if it's not found.
8. ***initSegments(),*** initializes segment structure for an existing tracker file by its file size, by default it will divide the sharing file into 20 segments in which each represents 5% of the file by size as close as it can by number of chunks. Each segment can be referenced by its index in the segment vector.
9. ***appendSegment(),*** append a new segment of chunks to an existing tracker file with the tracker file name, file size of the sharing file, index of the segment to be appended, and the port number of the client.

Source Code Citation

Client.findIP() **(Deprecated)** was based on:
Baudis, Petr (2008) and Kerrisk, Michael (2009). *getifaddrs(3)*. Available at
http://man7.org/linux/man-pages/man3/getifaddrs.3.html (Accessed 11 November 2014).

Client.readConfig() was based on:
mbaitoff (2010). *C read file line by line*. Available at
http://stackoverflow.com/questions/3501338/c-read-file-line-by-line (Accessed 20 November 2014).

Basic server and client networking code was derived from previous work by Matthew Lindner (CS 284 - FS 2013), which was based on:
Ercal, Fikret. *Internet Socket Example - server code*. Avaiable at http://web.mst.edu/~ercal/284/server1.c (Accessed October 2013).

The peer struct, peers[] array (initialization, findOpening() functions), and accepting connection in a loop were based on:
jienew (2008). *Chatroom Server/Client Project*. Available at http://wiki.jienew.twbbs.org/project:chatroom (Accessed 16 September 2014) .

compute_md5() was implemented with OpenSSL (https://www.openssl.org/) and was based on:
askyb (2013). *OpenSSL MD5 Hashing Example in C++*. Avaialbe at
http://www.askyb.com/cpp/openssl-md5-hashing-example-in-cpp/ (Accessed 5 November 2014).

compute_md5() was also based on:
askovpen (2012). *How to calculate the MD5 hash of a large file in C?*. Available at
http://stackoverflow.com/questions/10324611/how-to-calculate-the-md5-hash-of-a-large-file-in-c/10324904#10324904
(Accessed 5 November 2014).