Writeup for Project 3

Name: Robin Onsay

Netid: onsayrob

Date: 05/01/2022

Question 1: Explain your code and how it works.

List all of your source code files for your BitTorrent Lite. For each file list all of the functions you wrote. Under each filename explain what the code in the file does from a high level. Under each function name explain what the function does and how it does it. Example setup below.

File: peer-main.cpp

This source code file holds the main method for the peer implementation of BitTorrent Lite.

Function: signalHandler(int sigNum)

The signal handler cleanly exits the program on a SIGINT. This is useful so the socket is closed properly.

Function: main (int argc, char *argv[])

This is the main method of the peer program. It opens the correct files, obtains the owned chunks in a map, mapping the hash value of the chunk with the chunk, and then starts the peer.

File: peer.cpp

This file holds the implementation of the peer for BitTorrent Lite.

Function: comp(CHUNK& lhs, CHUNK& rhs)

This function compares via their index to determine which is greater. This is used for sorting later on.

Function: Peer::Peer(char *myIP, char *trackerIP, std::map<uint32_t, CHUNK> *owndChunks, std::ofstream *outFile, std::ofstream *log)

This is the constructor for the Peer object. It first connects to the tracker server and obtains the torrent file. It then parses the file and loads the contents into memory where they can be used. It also forms an object wide packet for the chunks that it owns (owndChunksPkt). Finally it binds the main receiving socket to the port and sets it as nonblocking.

Function: Peer::chunkInq()

This function performs a inquiry into what chunks are owned by which peers. It iterates through all the known peers and tries to connect to them. Once successful, it makes a request to the peer for its owned chunks file. It then parses the owned chunks file and creates a mapping of the chunk ID and the IP addresses that hold the chunk.

Function: Peer::reqChunk(std::string peerIP, uint32_t hash, CHUNK *chunk)

This function requests a chunk from a peer. Given the peer IP address, the hash of the chunk, and a pointer to where it should store the chunk, it forms a chunk request and sends it to the peer. It then checks if the received chunk matches the expected hash, and if it does it write the chunk to memory. This function return the bytesRead, or -1 on error.

Function: Peer::run()

This is the main thread, requesting chunks from peers. It first spawns the server thread that will handle all incoming requests from other peers. Then it pushed all the chunks that it owns to a linked list of chunks. Next, this function determines the rarest chunk and using the chunk map from Peer::chunkInq(), it requests the rarest chunk. Finally it sorts the chunks based on index using the comp function, and writes the chunks to the output file.

Function: Peer::closePeer()

This function closes the peer, waiting for all threads to finish and ultimately closing the socket listening for incoming requests.

Function: Peer::server()

The server function is spawned to listen for incoming requests. Once there is an incoming request, it spawns another thread to handle via the connHandler method. The server stops if the atomic variable end is set to true.

Function: Peer::connHandler(int cliSockfd, IP_ADDR cliAddr)

The connection handler accepts the connection from the other peer. It polls the file descriptor for incoming data. Once data is available, it determines the request type and passes it along the the proper handler. Once it's complete it closes the connection with the peer.

Function: Peer::chunkInqReqHandler(int cliSockfd)

This function is very simple, it writes the owned chunks packet to the client.

Function: Peer::chunkReqHandler(int cliSockfd, PACKET *pkt)

This function gets the hash of the requested chunk, finds the chunk via the owned chunks map, and responds to the peer with the requested chunk.

Function: Peer::write_p(int fd, char *buffer, size_t size)

This write function write data to the socket file descriptor. Because the write function can be interrupted at any moment, a looped write is necessary in order to send all the data through the socket.

Function: Peer::read_p(int fd, char *buffer, size_t size)

This function is the same as the write function, only it reads from the file descriptor.

File: tracker-main.cpp

This source code file holds the main method for the tracker implementation of BitTorrent Lite.

Function: signalHandler(int sigNum)

The signal handler cleanly exits the program on a SIGINT. This is useful so the socket is closed properly.

Function: main (int argc, char *argv[])

This is the main method of the tracker program. It opens the correct files, and starts the tracker server.

File: tracker.cpp

This file holds the implementation details of the Tracker class.

Function: Tracker::Tracker(char *pListPath, char *tFilePath, char *inFilePath, std::ofstream *log)

This is the constructor for the Tracker class. It parses the peers list, and creates the torrent file while also writing the torrent file into a packet in memory. Finally it binds the socket for the tracker server.

Function: Tracker::run()

The run function listens and accepts incoming requests, spawning a new thread to handle the request. This is handled by the connHandler method.

Function: Tracker::closeTracker()

The closeTracker function cleanly closes the tracker server. It does so by waiting for other threads to finish, then it closes the socket.

Function: Tracker::connHandler(int sockfd, IP_ADDR cliAddr)

The connHandler handles the connection request from a peer. It polls the socket for incoming data, and once data is available, it reads the socket. It then validates the torrent file request, and sends the file over the socket. Once complete, it closes the connection with the peer.

Question 2: Did you finish the entire Project? If not what was complete and what is incomplete?

Project Completely Finished.

Question 3: What did you learn doing this project?

I learned a lot about multithreaded programming in C/C++. Additionally, I was able to learn about handling file descriptors, atomic objects, and mutexes.

Question 4: What was the hardest part of this project?

The hardest part of this project was implementing the peer. Specifically, getting the peer to find out which chunks other peers had, and obtaining those chunks.

Question 5: If you were to start this project again, would you do anything differently?

I would have started the project sooner.

Question 6: Do you have any suggestions for improving this project?

This project was very fun, and I hope it stays as a part of this course for future students.