BitTorrent Client Part 3 (SWAGclient):

by Nick Paoletti and Daniel Selmon

Special thanks to Rob Moore for the large amount of help he gave me in completing this project!

How it all works:

This program has seen great changes since part 1, and most of the original code has been rewritten. The program still starts by reading from the .torrent file to create a TorrentInfo object, is stored globally. This is handled by the Metadata class, which instead of an object, is now a collection of static methods. Using the torrent info, the Metadata class then makes an HTTP GET request to the tracker, and stores the list of peers held within the results of the request to primarily assemble a list of peers. From here, a globally stored list of 'approved' peers – ones that have the IP addresses given in the project is created. In this implementation, each peer not only holds an IP address and port, but their own bitfield to save which pieces they have, and a Socket used to connect through the peer. Before each peer is connected to, the main method checks a text file that corresponds to the file name given in the arguments. If it's shown you have already downloaded the file, your bitfield is updated by reading in the corresponding text file to see which pieces you have. Each piece is checked at this point.

For each of these peers, a new thread is created. Each thread's run method is held within Download.java, and handles the downloading process. At first, a handshake is made, which in the process, establishes the socket for each peer. The peer acts as a medium for Messages sent – to send a message, the Download thread will create a Message, give it to the peer, which will give it to the Message class to encode and send on the peer's socket. It is similar for receiving messages – to receive the next message from the peer, the Download thread asks the peer to receive the next message. When the peer gets the next message, it asks the Message class to encode it into a neatly wrapped Message, which can be sent and interpreted by the Download thread. From the messages the Download thread receives, it looks at their information, and decides what to do in return. For example, if a piece message is seen, it will download the data to the hard drive and send a new Message asking for the next piece. Requests for pieces are completed using a rarest piece first algorithm. If a Have message is receieved, that Peer's bitfield will be updated to consider it has that piece – and if that piece is one you don't have, you will send a message saying you are interested in the piece. The biggest change is the implementation of uploading: by sending Have and Bitfield messages when you receive pieces, or are resuming the application using a file you have downloaded, other peers will show interest in you. From here, you will unchoke them, receive their requests, and send pieces in return. At most, only 6 peers can be unchoked at a time. During the download phase, the unchoked peers are based on who is seeding the most, while during uploading, the unchoked peers are based on who is leeching the most. Instead of terminating on download completion, the program will continue to upload pieces to other peers until the user terminates the program.

KNOWN ISSUES:

- Graceful exit is not working – I did not have the time to fix my error.
- The program does not listen for incoming connections. A pretty big deal!
- The git repo might be really messy.

Classes Depth:

btclient package:

RUBTClient.java:

This is the main class. It takes in the command line arguments, and from them, reads in the .torrent file, extracting it's metainfo, and creating a TorrentInfo object, and then looks at the filename of what you are naming your download. If you already have that file, then the program looks for the progress text file stating which pieces you have, and populates your bitfield with these values. From here, a thread for each peer you aim to connect to is established, along with a separate thread for tracker announces on the given interval, GUI updates, and Peer choking and unchoking. The main method keeps the program running until the types in 'q', which will stop all the threads, mark the pieces you have downloaded in a text file, and exit the program. This is also responsible for initializing the GUI, which at the time of writing, remains barely functional.

Metadata.java:

This class, formerly an object, is responsible for reading the torrents metadata – a .torrent file, creating a TorrentInfo object with it, and then using this information to make an HTTP GET request. In the process of making an HTTP GET request, a Peer ID for the user is obtained (SWAG and 16 random alphanumeric values), and information on all the peers is obtained. From here, a TrackerInfo object is made, storing the IP values for these peers. The Metadata also takes note of 'approved peers' – ones with the IP addresses 128.6.5.130 and 128.6.5.131, and stores them in an ArrayList full of approved peers in the FileManager. It is from this list that connections with peers are made.

TrackerInfo.java:

The TrackerInfo java is one of the more simple classes in the project. It holds a TrackerInfo project – of which this part of the project only has one, since only one tracker is involved. It stores values obtained from the HTTP GET request – Incomplete, Downloaded, Intervals, and a list of Peers. It also contains a method in which creates the URL address so that tracker scrapes can be made.

Peer.java:

The Peer object is one of the most robust objects in this application. Not only does it keep track of a peers IP, Peer ID, and port obtained from the HTTP GET request, but it keeps track of their bitfield – which pieces they have, and maintains a socket connection with Input and Output streams so messages can be sent, these being established at the time of the TCP Handshake, which the Peer class also handles. The Peer class acts as a medium between the Download and Message class – reading in Messages made by the Download class and using the Message class to send them, or reading in data from the InputStream, having Message decode it, and sending a decoded message back to the Download class.

Download.java:

The Download class implements Runnable, and it's main method, run(), reads in Messages from a Peer, interpreting the messages, and sending out corresponding messages in certain scenarios. Each time Download is ran, each Thread has its own corresponding designated peer, allowing each thread to handle uploading and downloading to a different peer. Download.java also features the methods for handling and sending messages: for example, makePiece creates a PieceMessage when a peer sends a RequestMessage. Download.java also has a method, shaHash, to hash a byte array and compare it against another byte array – which in this case is given in the TorrentInfo.

TrackerAnnounce.java:

The TrackerAnnounce class is very simple. It is a Runnable that on the interval given in the tracker info, makes announces to the tracker – in particular, amount uploaded and downloaded (the other information stays the same throughout). This is it's own thread that terminates when the rest of the program does.

PeerChoking.java:

Like TrackerAnnounce, PeerChoking is a Runnable on every thirty seconds analyzes the download speeds (during your uploading phase) or upload speeds (during your downloading phase) and ranks peers based on them. It will unchoke the top 6 peers (a number specified in the FileManager), and choke any other than those 6.

SpeedCalculation.java:

Like the previous simple Runnables, SpeedCalculation is in charge of updating something on a given interval – in this case, your upload and download speed on a ten second interval. However, the GUI implementation is incomplete.

FileManager.java:

FileManager is the glue that connects a large amount of threads and peers, and all the various classes and objects in this file into one cohesive collection of static variables... which need getters and setters added to them. Indeed, the direct access occurring from this file is something that certainly needs to be cleaned up for the last part of the project. FileManager holds many important static fields accessed by many different classes – your bitfield keeping track of which full pieces you have, a subpiece bitfield keeping track of which little 16kB pieces you have, the .torrent files information, the TrackerInfo for this projects tracker, the file in which pieces are downloaded to, a list of Peers you are connected to, your download and upload tally, and which pieces are currently in process of downloading. Keeping these fields in one global location helps the many threads of the program interact decently seamlessly. The FileManager also has a couple methods to store your bitfield onto a text file upon program exit - storeFileProgress, so on program resume, it can read this progress – readFileProgress – so you can pick up where you left off downloading, or be able to seed the files you have to the list of Peers.

btclient.messaging package:

This package contains 5 classes, each of which are objects – the Message class, and four specific message types which extend the Message class.

Message.java:

The Message class holds Message objects – convenient ways of storing the information from messages downloaded from the peer, and takes allows Messages made by the user to be passed on the output stream without the Download class worrying of those specifics. It contains final values corresponding to the type of the eight types of messages, and also contains the messages which have one value every time – choke, unchoke, keep alive, interested, and uninterested.

The decode method is responsible for reading raw bytes from within the InputStream established from the Peer, and turning those bytes into a message – for example, reading in the bytes of a bitfield and turning it into a boolean array of true and false. The encode method takes a Message, usually made by the Download method of the application, and takes the values of the fields to write bytes upon the OutputStream. All types of Messages contain a Length and a Type – this is inherited through the four unique types of messages.

BitfieldMessage.java:

Retaining the length and type from the super class, Message, a Bitfield message also contains a boolean array representing a bitfield – a 1 being true, a 0 being false.

HaveMessage.java:

Retaining the length and type from the super class, Message, a Have message contains the index in which a Peer 'has'.

PieceMessage.java:

Retaining the length and type from the super class, Message, a Piece message contains the piece index of a piece of data, an integer containing it's byte offset, and the actual bytes of that piece, stored in a byte array.

PieceMessage.java:

Retaining the length and type from the super class, Message, a Request message contains the piece index of a piece of data being requested, the byte offset where the request starts, and the length of the desired piece from that request.