Ryan Couch
Daniel Torres
Phase 3 Write-up

Util.java

Contains additional tools used within other classes. Methods used to create random peerId's, print a list of peers with corresponding data, convert byte[]'s to hex and URL format, search for peers by a given IP address, covert bit arrays to boolean arrays, etc.

Message.java

The message class is used in communicating with the peers after handshaking. Types of messages are defined within this class (have, request, piece, bitfield, etc), and can be created/initialized from here. The main two methods are read(InputStream) and write(OutputStream, Message). In 'read', an int and a byte are read from the InputStream, and the corresponding types of Messages are generated. In write, a parameter Message is sent to the peer through the parameter OutputStream.

Tracker.java

The tracker class is the instance of the 'tracker' that we connect to as a peer. Within this class a tracker can be initialized, along with a URL connection made to it. The main methods are announceToTracker(int, int, int, event) and getPeers(Map<ByteBuffer, Object>). In announceToTracker, an announce URL is generated and sent to the tracker. In getPeers, a list of peers is retrieved from the response message the tracker sends.

Peer.java

The peer class deals with the interfacing with an actual peer. From here, a client can connect to a peer through a socket. Once connected, a handshake message will be generated and sent to the peer. The response will be validated, and message can then be sent and received, all of this done through the createHandshake and verifyHandshake methods. The handleMessage method takes in a message generated from the InputStream/Message class, and certain actions will take place for each type (have, interested, piece, request, etc). The peer will push all messages received from the peer into a queue, which will be ultimately handled by the Client. This class also sends periodic keep-alive messages to the peers notifying them to keep the connection open.

RUBTClient.java

The RUBTClient class is where the main() method is located. Two arguments are read in; the torrent file to be opened and the name of the file to save the download contents to. The torrent file is read in, and the corresponding data it contains is stored. The client is what facilitates all downloading and final handling of messages. From here the queue of messages are handled, sent by the peers. As pieces of the file are downloaded, the client verifies them and stores to file. The bitfield is updated and have messages are then sent.To select pieces the rubtclient uses a rarest piece algorithm. The way this works is that each time a call is made to choose piece the client runs through all current connected peers and sees what peers have what pieces, each time a peer has a piece the individual pieces 'count' goes up. To keep track of this we created an availability class which holds an index field and a count, this allowed us to keep track of what pieces had the least availability I.e the rarest pieces to be selected.

Overall, this project is turning out to be something that is both challenging and interesting. While it is on a larger scale than most projects we have done previously within Computer Science, it is also on an interested topic. We both actively use torrents so we find interest in it. However, we are used to smaller, simplified programs with a given outline. Thus, the BitTorrent client has opened our minds to more difficult, large projects and how they must be planned and executed. The easiest parts were to initiate a

a larger scale than most projects we have done previously within Computer Science, it is also on an interested topic. We both actively use torrents so we find interest in it. However, we are used to smaller, simplified programs with a given outline. Thus, the BitTorrent client has opened our minds to more difficult, large projects and how they must be planned and executed. The easiest parts were to initiate a tracker and connect to it, and create a list of peers. We ran into some issues getting responses back from the peer following the handshake, and ended up wasting a lot of time trying to get past it. Once successful in debugging, we had limited time to execute other aspects of the project. Implementing a multi-threaded, multi-connection instance of the BT client posed its own challenges and we ultimately had to start the assignment over from scratch.

We also had an issue running two peer connection threads at once. It was difficult for us to test the simultaneous-peer aspect of Phase 2, mostly due to the .130 peer being down and us being unable to test connections to .131 and .130 at the same time. We were unsuccessful in trying other methods to test this, but there should be no major issues.

The final problem area is dealing with peers that dont respond correctly, for example the 131 peer responds with an incorrect bitfield while the 130 peer simply only sends have messages and never sends a bifield, this was extremely difficult to design around but was useful in that it created robust code.Interacting with multiple peers is extremely cumbersome even though our design was meant to handle multiple peer connections we keep running into blockers.