Write Up

Alex Aguilera – 144005503

Chris McDonough – 154004779

Kyung Kim – 134007741

RUBT Client:

How it works-

To start our program, you must enter the commands

```
Javac –cp . RUBTClient.java
Java –cp . RUBTClient.java Phase2.torrent save.file
```

Start

After our program starts, the first thing it does is contact the tracker. It sends a GET Requests and receives information back which is decoded by the GivenTools and a method in our program which gives us all of the torrent info. It also creates a new RandomAccess file with the size of the File specified from the file_length in the TorrentInfo

Memory Setup

From there our program begins to set up our pieces in memory so that when we connect with peers, we have a structure to our memory too. It sets up however many pieces there are in memory from the class Piece.java and Memcheck.java, which we used as global memory in this assignment. Now that this is done, the program begins to set up each of the Peers. Our Peers each get their own instance of the class Peer, which implements runnable. Each peer when created opens up data streams and sends a handshake message which gets received and acknowledged by the other peer. Once this is done, we now move on to the run method in each Peer which is where threading starts.

Download/Upload

This is also where our program starts to run into trouble. We start off receiving the Peer's bitfield message and writing it into memory to see which pieces that peer has available. After that, we send an interested message and receive an unchoke message back, meaning that we are now ready to download from this peer. We have a queue that is populated by what messages we need in memory, and we send out a maximum of 4 piece requests, and after that the thread sleeps for a second to allow time for network decongestion/flow control. Before doing this, after sending out about 20 requests, the pipe would break and the program would stop running. Once I made the thread sleep, my program started running the download sequence as expected. This also makes it a lot easier to see that multiple threads are running and interweaving for the download. Each time a piece message is received, it is added to memory where we check if the full piece has been downloaded. If it has, we verify it against the hash and if the hash checks out, we write the piece to the File. Our program runs into trouble at usually around piece 420-435 however, as both of the threads reach an EOF exception for the data input streams (This also happens earlier sometimes) and then the sockets throw an error saying there is a broken pipe. So the program does usually simultaneously download ~90% of the file correctly but we could not figure out why the pipe kept breaking. Any feedback as to why this was happening would be greatly appreciated for helping with the next phase. Also, we noticed that we never received any request messages to tell us if they wanted some pieces that we have, so we never really got to uploading the files, which we are not sure as to why that was happening.

Conclusion

We did make a lot of progress downloading the file and had connected to Multiple Peers and the tracker, but still have to figure out how to upload simultaneously and probably have to redesign our algorithm to work with the

Each persons work:

Alex – Peer connections/download

Chris – Memory setup/verification of pieces

Kyung – User input/Research