Paul Zbarcea, Harvey Sun
5/10/22

CMSC417 Final Project: zTorrent

List of supported features:

1. Compile, Run, and Run Tests in Maven:
    a. UnitTests in Maven run automatically on build
    b. Provides commands like mvn:exec java that make things easier
    c. Can be recompiled for different versions of Java
2. Can be configured in pom.xml
    a. For example, running the CLI with specific arguments
    b. Dependency list is both clear and configurable
3. Both a command line and graphical interface for the client
4. DHT, HTTP, and UDP Tracker all supported
5. Can upload and download files at the same time
6. Can connect to multiple peers, including both commercial clients and other zTorrent instances
7. Relatively fast download speeds. Around 2MB/s during initial testing, detailed below. However, increasing the maximum peer queue size improved download speeds significantly, which we didn't get to test. Through observation, new download speeds seem to be about 5-10 times faster, ranging from 10-20MB/s.

Charlie Chapin's 1914 "The Good for Nothing" (244.083 MB):

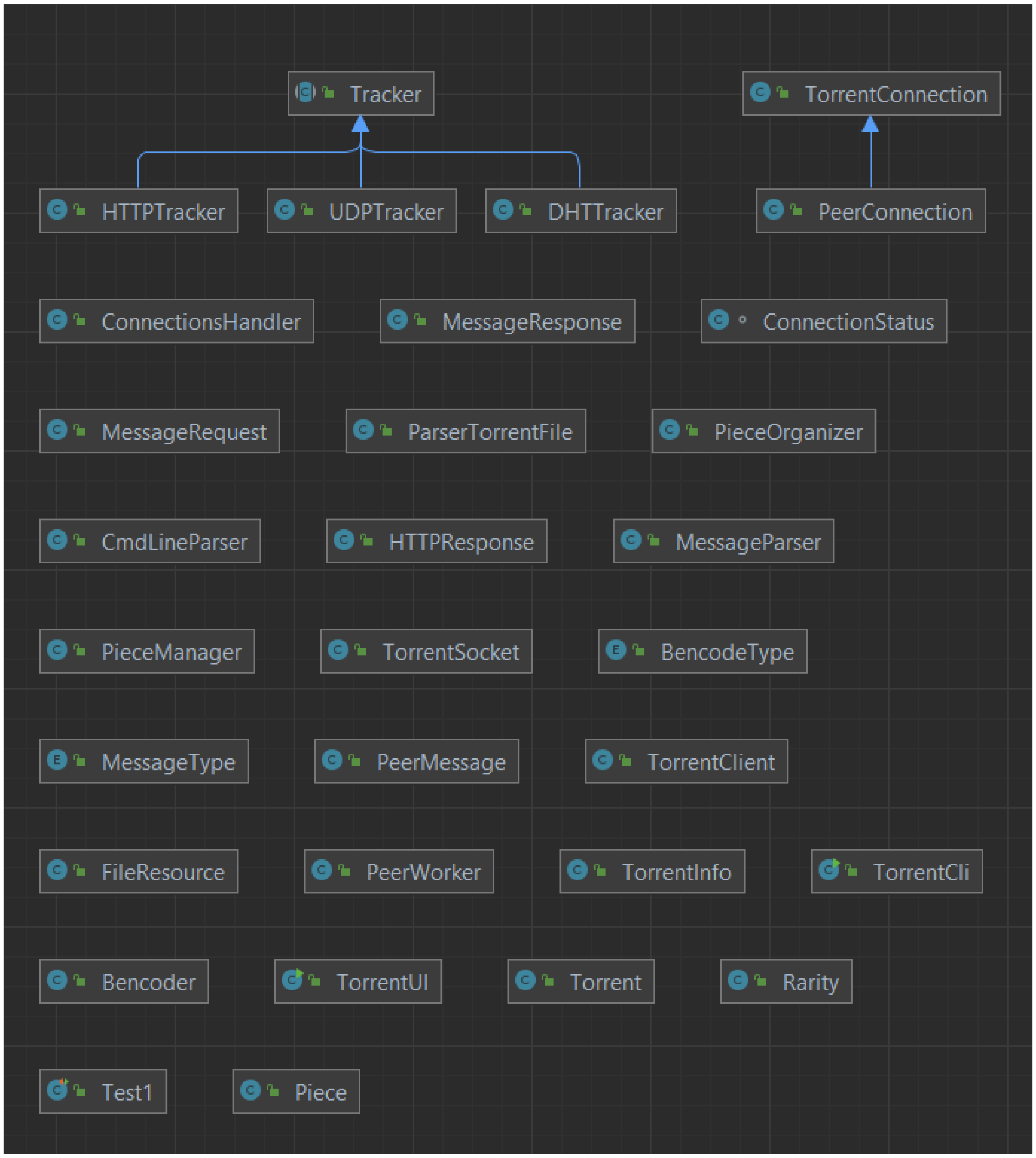
qBittorrent	Transmission	Our client
1:32.69	0:59.63	3:18.65
1:32.72	1:05.66	3:20.21
0:51.42	0:56.39	3:15.22
0:46.56	1:03.20	3:10.15
0:46.06	1:04.09	3:11.20

Design and Implementation choices:

1. Choosing the Language (Why Java?)
    a. The project specification says that the speed of our client is important, so we considered Java and C++ over Python (which we used for Project 4) since they're both compiled languages. Java is also well-documented and supports lots of libraries. Finally, we were both extensively familiar with Java and its principles, which would speed development and (hopefully) result in less-buggy code.
2. Understanding the protocol, designing the program:
    a. The protocol has a lot of moving parts. To understand how each part works and how they would interoperate, we:
        i. Inspected uTorrent packets through Wireshark
        ii. Looked at open source implementations to get an idea of how things work:
            1. tTorrent: https://github.com/mpetazzoni/ttorrent
            2. http://www.kristenwidman.com/blog/33/how-to-write-a-bittorrent-client-part-1/
            3. https://allenkim67.github.io/programming/2016/05/04/how-to-make-your-own-bittorrent-client.html
            4. https://blog.jse.li/posts/torrent/
        iii. Made sure to note what similar programs use in terms of classes and hierarchy, such as this documentation https://hackage.haskell.org/package/bittorrent-0.0.0.3/docs/
3. Creating the structure
    a. We used a template from Maven to create the project, a pom.xml, and the directory structure that would be needed to host our project
4. Designing our program:
    a. The next step was to create a list of necessary classes and model our base implementation
    b. The final project ended up being larger than anticipated, with multiple inner classes. Our design exemplified features of encapsulation, combining tools we would need in singular classes. For example, we wrote a utility class MessageParser, which parses incoming messages from peers, creates PeerMessages, and adds them to a queue. We also implement inheritance through the UDPTracker, DHTTracker, and HTTPTracker classes.
    c. Much of the program required parsing messages, so that is where we began in our implementation. We started with the Bencoder class, which is needed for everything to work since reading the torrent file and sending messages would all need to be encoded and decoded properly. After that was working and tested, we moved on to the MessageParser, and Trackers, since they were core to the implementation. Afterwards, we started adding more classes as we realized they were needed, such as the Piece class that is needed to represent one piece of the

file, and then the PieceOrganizer which represents data about the piece. Everything branched out from these initial classes, using the Wikipedia protocol as a guideline.

d. A picture of our class diagram is below (generated from intelliJ):



Problems we encountered:

1. HTTP Responses include the header, not just the body, so we needed to parse through that before we got to anything useful.
    a. Initially we were unsure why we were not getting useful data, since we expected to only receive the message body
    b. After some debugging, we finally realized the issue, and realized that we had to parse the message header to get to the message body, which we implemented in HTTPResponse.java
2. We had trouble uploading to other clients at first. Again, the issue was not immediately obvious (we thought it was an issue with our implementation) so we had to debug and eventually realized the problem after opening a different client (uTorrent) and being prompted to allow access through the firewall. We ended up using port forwarding to get uploading to work properly. A superior implementation would prompt the user for a firewall exception but we deemed this was too hard to implement and did not include it in our final submission.
3. Before we increased the maximum peer queue size, we were concerned about the relative speed of our client. A piazza post specifies that 1 gigabyte in twenty minutes is an acceptable benchmark, so we did lengthy testing to figure out whether we could consistently meet this criteria. During this testing, it was difficult to find torrents that would communicate with us because (we believe) of our lack of encryption. Eventually, we found a few suitable torrents: the odyssey.txt file, the raspberry pi OS, and a public domain Charlie Chapin movie.
4. Writing to file was extremely tedious and difficult, and led to a lot of problems for us. At first, we tried writing into a single file, and it never worked properly. Finally, we decided to try first writing to a temp file, then writing output to a completed file when the download finished. This led to issues because we were not updating the file pointer correctly. After extensive testing, we were able to make the file writing work, but this took multiple days to correct.

Known bugs and issues:

1. Popular torrent clients allow users to delete two ways: only the .torrent file, or the .torrent file and the associated data. Our implementation deletes both the downloaded file, and the .torrent on clicking “Remove”. This is an issue that we’d like to resolve in the future by including another button to remove only .torrent and keep the downloaded file.
2. Currently, we do not support magnet links. This is due to an issue we had parsing the links, and we found it difficult to configure, so we removed this from our implementation. Most commercial clients support this, so we consider this an issue with our implementation.
3. On testing, we noticed that if a file is deleted manually (from command line using rm or from File Explorer on windows) then we get a nullPointerException that causes our GUI to hang and our program to crash. We did not fix this edge case, and it is a known bug in our implementation. To fix it we would have to find where the exception is thrown and surround it in a try catch.
4. Another known issue is that when we delete the .torrent and downloaded file, we leave the directory intact and also leave the file.temp which is used to store pieces as they are downloaded. This is an issue because it means we do not clean up properly and our client can leave remnants between runs. These remnants affect the %downloaded display on the GUI, causing some confusion.

Contributions:

We worked together on the design, structure, and implementation of the project. In terms of individual contributions, Paul configured the Maven setup and Harvey cobbled the GUI together.