CMSC417 Final Project BitTorrent Report

Patrick Logue
Brian Sherwood
Haylee Lee

1. List of supported features

- Communicate with an HTTP tracker
- Download files from Poole
- Download files from official BitTorrent clients
    - Tested with debian-11.5.0-amd64-netinst.iso.torrent
- Uses a rarest-first strategy for downloading pieces
- Discards slow connections
- Can seed indefinitely

2. Design and implementation choices that you made

To begin, we chose to implement our BitTorrent client in Python. Handling binary data and networking is simple with the struct and socket packages. We wanted to leverage these simple libraries alongside ease of use to streamline our development process.

In the design phase we decided to split our client based on it's primary functionalities into four different classes:

client.py - Serves as the program's main and controls the peer to peer protocol (including accepting new connections from other peers).

To start our BitTorrent program, run "python3 ./client <*path to torrent file*> <*compact*> <*port*> <*seed*>" where *path to torrent file* is the local path to a .torrent file, *compact* is either 0 (not compact) or 1 (compact formatted when communicating with the tracker), port is the port number to listen on, and seed is 0 (do not seed) or 1 (seed).

client.py initializes 3 classes: a tracker instance, a PeerList instance as peer_manager, and a FileDownloader instance as pieces, then informs the peer_manager to connect to the peers it received from the tracker. It then binds master_socket to port 6881 to listen for TCP connections from the peers and starts its while loop. The loop runs while pieces determines that it does not have the entire file. At the start of the loop the select timeout is initialized to the minimum of either the keep alive timer (set to one minute) or the tracker interval timer (set to the amount we receive from the tracker). Then it checks whether there are currently unchoked peers and how many outgoing requests exist. If within the threshold, choose a Piece and send requests for Blocks in that piece that have not been completed. Next we call select. If it detects readable data on the master_socket, it accepts the connection, initiates the handshake, and adds the peer to the

peer_manager list. If it detects readable data on any other socket, it reads in the data, processes the len and id of the message and passes it off to the peer it originated from to handle. If it detects the peer has closed the connection, it clears any requests sent to it, removes it from the list, and closes its socket. When the select call hits a timeout, it checks whether it should send keep alive packets to each peer or check back in with the tracker, and adjuncts the timers accordingly. The timeout also clears and removes any peers it has not seen in more than two minutes.

tracker.py - Controls sending, receiving, and processing requests to and from the tracker.

During initialization of its Tracker instance, it first parses the torrent file and constructs its HTTP GET request to the tracker. Then it processes the response and initializes a list with hostname/ip and port for each peer it received.

peer.py - Controls sending, receiving, and processing messages to and from each Peer as well as maintaining the state of each Peer.

This file contains two classes, PeerList and Peer. Each Peer instance represents one external peer our client has connected to. It keeps its own state, bitfield (maintained as a bitarray), socket, and time it was last seen. It has methods to initiate a handshake with a new peer, send each type of message, handle interested/choking statuses, and handle messages passed from *client*. The PeerList class maintains a Peers and has methods that call the initial connections to new peers received from the tracker and return peers from its list by certain characteristics. The Peer method that handles messages passed from *client* updates its own fields and calls *pieces* to update the local fields appropriately.

pieces.py - Constructs and maintains the state of each Piece in the file made of Blocks.

This file contains three classes, Block, Piece, and FileDownloader. Each Block instance represents one block, or segment of a Piece that gets requested from peers. Each Piece instance represents one piece, or segment of the file that has a corresponding hash. The Piece class maintains a list of its Blocks and has methods to initialize, update, reset, and check that list. The FileDownloader class maintains the file name and size, and controls a list of Pieces. It has methods to initialize the Piece list, update Pieces' blocks, retrieve the rarest piece, and write a piece to file.

In addition to the four primary functionalities, the ***message.py*** file contains classes for each type of message. Each class can be instantiated to send a new message of that type or to unpack received bytestreams.

3. Problems that you encountered (and if/how you addressed them)

The largest non-technical problem we encountered on this project was a lack of communication when we were unsure how certain aspects of the protocol worked. This resulted

in *pieces.py* being implemented with an incorrect understanding of Blocks and Pieces and their relationship to each other and the peer protocol. We solved this by rewriting *pieces.py* and helping our teammate with their misunderstanding.

Technically, we struggled with our client's download speed. Once we were able to successfully download the flatland file from poole, we noticed that it retrieved larger files at an incredibly slow rate. For instance, the first successful download of the debian iso file was over an hour long. We eventually improved our client through optimizations in the request pipeline, refusing connections with slow handshakes, and smart removal of slow peers to reach speeds of up to 3 minutes and at most 10 minutes, but have not achieved consistent download speeds comparable to official BitTorrent clients.

A second technical issue stemmed from clients passing data over the localhost. We had problems keeping a connection alive between two clients both living on the same localhost. We solved this by setting the new local host peer with .settimeout(0). This seemed to not break the pipe and allow for our data to be transferred between clients.

Finally, many hours went into debugging the networking protocol. We mainly solved pipeline errors and unpacking errors by setting the sockets to block instead of having them Timeout.

4. Known bugs or issues in your implementation

The biggest issue our program currently faces is its speed. It is able to completely download both poole and debian torrents, but the debian often takes several minutes.

Another issue we have noticed is that once our client gets very close to finishing the file, the number of requests in its pipeline becomes a negative number and it spams one seeder at the end to finish the file (it does end up completing the download though). This is not intended and the cause is unknown.

Lastly, our client has trouble connecting to other instances of itself. We are unsure if this is because of errors in our implementation, or because our method of testing it (using localhost) was causing undue environmental havoc.

5. Contributions made by each group member

Patrick is the MVP of this group. He developed tracker, message, and significant portions of client. He also completely rewrote pieces, helped with peer, and spent a significant amount of time debugging the whole program. Everyone else in this group owes Patrick lunch.

Brian developed peer, significant portions of client, and spent too many hours banging his head against the wall trying to debug the whole program. He also wrote the majority of this report.

Hayley developed the initial implementation of pieces.

References

https://blog.jse.li/posts/torrent/

http://www.kristenwidman.com/blog/33/how-to-write-a-bittorrent-client-part-1/

https://www.morehawes.co.uk/old-guides/the-bittorrent-protocol

https://wiki.theory.org/BitTorrentSpecification#peer_id