Bittorrent

Charles Jin, Jason Kim, Marvin Qian, Harvey Xia

Summary

This project is an implementation of the Bittorrent protocol. This includes a peer application that contacts a tracker server and exchanges data with other peers.

Design Goals

In this implementation, we sought to implement the peer-to-tracker and peer-to-peer Bittorrent protocols. Interestingly, the official Bittorrent specification offers only a high-level description of these protocols, so we've designed and implemented protocols that meet the specification's requirements. We describe these protocols in more detail in further sections.

Problems Encountered

- Peers would sometimes attempt to simultaneously connect with one another. When this occurred, it was possible for two connections (one in each direction) to exist between to peers simultaneously, which is incorrect. To fix this issue, we added a policy so that only newly joined peers are responsible for initiating connections. This eliminates the race condition. However, this requires that newly joined peers connect with all other peers. While this would be infeasible for very large peer lists, at the scale of this implementation this design decision causes no issues.
- The Bittorrent protocol specifies using a unique identifier for each peer. At first we thought this was only necessitated by local IP and NAT translation. Later we realized that we need a unique indentifier because each of the client's outgoing connections is bound to a random port that happens to be available. This means we can't rely on the values of `socket.getInetAddress()` and `socket.getPort()` to identify a peer. Instead throughout the application we identify peers by their IP and welcome socket port.
- Because the protocol relies on a lot of TIMEOUTs, the application was multi-threaded by necessity. This meant that we had to be very careful with concurrency and multiple threads all needing to access and alter the same data structures.
- Because the application runs a lot of threads (one for each message), we implemented the client with an executor with a fixed thread pool, in order to limit the overhead associated with thread creation/deletion.

How To Use

To make: `make`

To test: `./runtests.sh`

See section "Run" for how to spin up your own instances of seeders, leechers, tracker, and a sample run.

Project Structure

The project is implemented in two primary packages: core and tracker. The core package contains code for the client application and

peer-to-peer communication. The tracker package contains code for the tracker and peer-to-tracker communication.

metafile/Metafile.java - object representing torrent metafile, i.e. the .torrent file. Also parses a file and returns a Metafile object

tracker/Tracker.java - the server that responds to requests from peers

core/Client.java - the bittorrent peer

core/Unchoker.java - Runnable that periodically runs the unchoke algorithm and updates the list of unchoked peers

message/* - Contains various classes that build and parse bittorrent messages over TCP

utils/Datafile.java - object encapsulating data file, i.e. the file that is being downloaded and uploaded

utils/Logger.java - Logging object

Protocol

This implementation follows the protocol as described here. The protocol provides a way for users to share files in a peer-to-peer manner. Peers are divided into seeders or leechers. Seeders have the full file and only upload to those seeking to download the file. Those downloading the file are leechers, but they also help other leechers by uploading the parts of the file that they already own. Peers find each other via a tracker that keeps track of peers. The tracker is identified by a metadata file known as a torrent file.

Once peers are connected, they can be either interested or not interested in each other's data. If they are interested, then they send an interested message to the other peer. The peer can then decide whether to choke or unchoke them. If they decide to choke them, then they refuse to upload to them for now. They may choose to unchoke them later. When a peer decides to unchoke another peer, then it notifies the other peer. That peer can now start requesting data and the peer should respond with the requested data. These relationships are bidirectional, one for uploading and one for downloading, but each direction is independent of the other.

In order to deal with churn, i.e. drop offline peers and discover new peers, peers must also ping the tracker every so often to 1) reregister themselves as a viable peer, and 2) receive the updated peer list with dead peers removed and new peers added.

More detail about the protocol can be found at the link above.

Run

In order to run a basic tracker + seeder + leecher setup, run each of the following commands in a new terminal and in the order shown below:

Tracker

```
java tracker.Tracker 6789
```

Seeder

```
java -cp .:lib/json-20160212.jar core.Client Client1 6000 file/to/share trackerHostname/IP trackerPort
```

This also creates a torrent file for the file you're trying to share. This should be given to the leecher so that he can download the file.

Leecher

```
java -cp .:lib/json-20160212.jar core.Client Client2 7000 path/to/torrent downloads/directory
```

Design

Client

The client serves as the interface for sharing and downloading a file. If the user has a file to share, he can advertise the file to a tracker (Client.java:79) and then share a torrent file so downloaders can find the file. If the user wants to download a file, he just needs to initiate the client with the associated torrent file. Downloading or uploading multiple files simply involves running multiple client instances--one for each file.

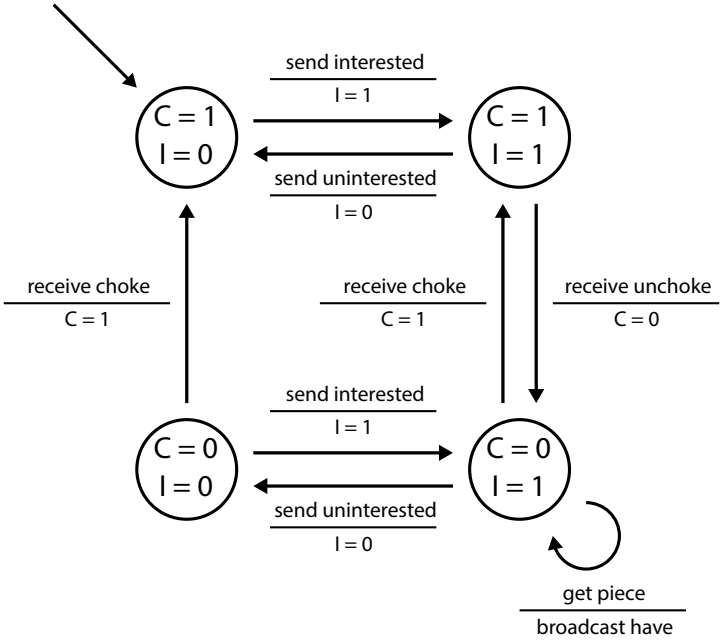
The implementation of the client consists of two main threads that can create separate tasks to be run on a thread pool. One of these threads simply presents a welcome socket and accepts new connections from peers (Client.java:66, Welcomer.java). The other thread parses messages from the connection sockets of all connected peers (Client.java:67, Responder.java). Once the messages are parsed, they are bundled into a task instance and passed to the thread pool for processing (Responder.java:54, RespondTask.java). The message responses are delegated to two objects based on whether the action is related to uploading to the peer (Uploader.java) or downloading from the peer (Downloader.java). In addition to these tasks, the client also has to query the tracker periodically for the updated peer list (Client.java:64, TrackerTask.java). This is done through scheduled tasks and allows the client to be robust to peer churn since it will quickly discover peers that have joined or left. The first time this task is executed, it connects to the available peers via a simple protocol handshake that involves exchanging bitfields for the file of interest. Finally, an unchoker task also runs periodically in order to update which peers the client has decided to unchoke (i.e., upload to them) (Client.java:63, Unchoker.java).

For each peer (IP-port pair), the client keeps a set of connection information (Connection.java). This information includes the associated socket, the peer's bitfield, his upload rate to the client and download rate from the client, and choke/unchoke and interested/uninterested state for the protocol.

We implemented the peer-to-peer protocol using two finite state machines, one for the download actions of a client and one for the upload actions of a client, as shown below. For each file, the client maintains a list of four peers. If the client hasn't completed downloading the file, then this list contains the four peers that have provided the fastest *download* rates. If the client has completed downloading the file (i.e., is now seeding), then the list contains the four peers that have provided the fastest *upload* rates. This tit-for-tat strategy provides incentive to upload quickly to your peers in order to receive more data in return.

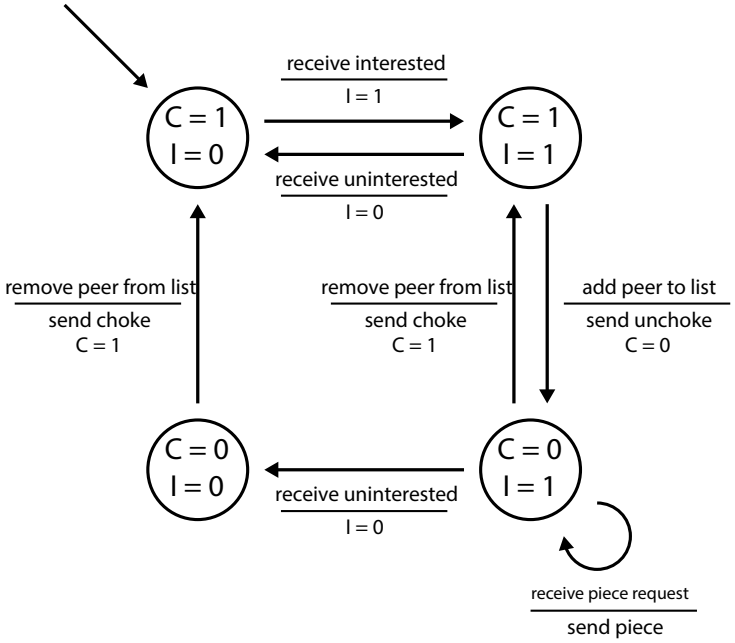
Downloader FSM

Note: C = peerChoking and I = amInterested.



Uploader FSM

Note: C = amChoking and I = peerInterested.



In summary - each Client (Client.java) is responsible for a single file. - peers are represented by Peer objects (Peer.java). Each peer also has an associated State (State.java) and Connection (Connection.java). - the Client has multiple Connections (Connection.java), one for each peer it is talking to (uploading and downloading from a peer is a single connection). - the Client has a Downloader

(Downloader.java) object that handles messages sent from peers uploading to it. The Downloader also stores download metrics (like speed) in the Connection object associated with the peer (used for tit-for-tat unchoking algorithm). - the Client has an Uploader (Uploader.java) object that handles messages sent from peers downloading from it. The object also maintains the state associated with which peers are currently unchoked. - the Client has a Welcomer (Welcomer.java) that listens for new download requests on a public welcome socket. - the Client has a Responder (Responder.java) that iterates over all current connections and spawns a new RespondTask for each new message. - the RespondTask (RespondTask.java) picks either the Uploader or Downloader object to handle the response. - The Client also schedules an Unchoker task (Unchoker.java) that periodically picks new peers to unchoke based on who is uploading the fastest (tit-for-tat).

Tracker

The Tracker protocol was as follows. Its main job was to maintain a mapping from a file to a dictionary of peers currently downloading / uploading that file. For the purposes of the tracker, peers were stored as IP / port of their welcome socket, since that is how the peers would contact each other.

At a high level, every peer was required to register with the tracker in order to get a list of peers. The tracker response included this dictionary of peers, as well as a timeout value during which the peer was expected to re-ping the tracker for an updated peer list. This pinging also served the purpose of notifying the tracker that the peer was still up and running, and if the peer did not respond within the timeout period, the the tracker assumed that the peer had died and removed it from its own peer list.

Specifically, the tracker implemented the following messages, as specified in the BitTorrent protocol: - STARTED The client sends this message when it wants to begin downloading a file. The tracker registers the peer in its map and sends a response with all the other peers for the file. The tracker also passes a TIMEOUT parameter that tells the client how often to PING the tracker. - COMPLETED The client sends this message when it is done downloading a file. Because our tracker does not keep track of seeders and leechers, this message essentially doesn't do anything on the tracker end, except in the special case that the peer is trying to upload a file for the first time. In this case, the peer will send the tracker the COMPLETED message as its FIRST message, and the tracker will create a new entry in its map for the file (otherwise, all leechers for files that do not exist yet are just given a message to try again after TIMEOUT seconds). - STOPPED The client sends this message when it is undergoing a graceful shut down. On receiving this message, the tracker removes the peer from the peer list for the file. Note that because the tracker removes all peers that do not respond within a certain period, this message is optional (as according to the BitTorrent protocol.) - PING The client sends this message onces every ~TIMEOUT seconds to 1) let the tracker know that it is still up and running (and thus the tracker can pass out its information to peers for downloading), and to 2) get an updated peer list from the tracker.

From an implementation perspective, there are three main components to the Tracker.java file:

- run (Tracker.java:54) The main event loop. The tracker just waits on the welcome socket for incoming connections
- processReq (Tracker.java:82) Accepts requests from run as they come in, and responds according to the message type, as detailed above
- stopTimer (Tracker.java:173), startTimer (Tracker.java:187), CheckTimeout (Tracker.java:203) Takes care of timeout for peer PINGs. The tracker registers a timeout with startTimer every time it receives a valid STARTED or PING message. The tracker cancels the outstanding timeout for every valid STOPPED message. Finally, CheckTimeout merely takes the filename and peer pair, and removes that entry from the peerlist if the event actaully fires.

The Tracker also relies on some helper files:

- TrackerRequest.java is a helper file for creating / parsing requests sent from the client to the tracker.
- TrackerResponse.java is a helper file for creating / parsing responses sent from the tracker to the client.
- TrackerTask.java is a thread that the Client schedules (Client.java:64) in order to ping the Tracker. Note that in order to account

for network latency, we do not set the TIMEOUT to the actual advertised TIMEOUT value, but some slightly smaller value. We stress that this is in line with the protocol, since the advertised TIMEOUT value is only saying to contact the tracker at MOST after TIMEOUT seconds.
- TrackerClient.java is a wrapper class for the client's interface with the tracker. In particular, update (TrackerClient.java:25) takes parameters (such as what type of message to send) and forms the TrackerRequest, sends it to the tracker, and returns a TrackerResponse message.

Testing

TestRunner.java - The class for running the tests. For this project we used the Junit framework.

MessageBuilderTest.java - Checks to see whether the MessageBuilder class is implemented correctly. The MessageBuilder is a helper that creates all the message types.

MessageParserTest.java - Checks to see whether the MessageParser class is implemented correctly. The MessageParser is a helper that parses all the message types.

MetaFileUtilsTest.java - Checks to see whether the MetaFileUtils class properly parses metafiles.

SeederLeecherTest.java - Runs end-to-end tests with one leecher, three leechers, and three leechers with one of them failing in the middle of uploading to other peers. Checks to see that all leechers manage to get a full copy of the file.

TrackerTest.java - Checks to see whether the Tracker is properly implemented - testNewFile (TrackerTest.java:45) sees whether peers can properly register new files with the COMPLETED message - testPeerList (TrackerTest.java:87) makes sure that peers sending STARTED and STOPPED messages are properly registered by the tracker. This is end-to-end tested by seeing whether the peer list returned by the Tracker in response to PING messages properly reflects the updates. - testTimeOut (TrackerTest.java:195) makes sure that peers that do not PING within the specified TIMEOUT are removed by the tracker. This is end-to-end tested by seeing whether the peer list returned by the Tracker in response to PING messages properly reflects the updates.

Future Directions

Our implementation simplifies the protocol in a few ways:

- Instead of downloading a piece from a peer one block at a time, we download a piece all at once. Extending our implementation would involve sending and receiving data at block granularity.
- We currently do not implement SHA1 verification of pieces. Implementing this feature would involve the receiving client, upon receiving a complete piece, to verify the integrity of that piece by computing its SHA1 hash and comparing it against the SHA1 hash for that piece as defined in the torrent metafile.
- We request pieces in simple sequential order instead of seeking the rarest piece first.
- We do not implement optimistic unchoking. This does not affect files with at most 4 peers, which suffices for the tests that we've written.
- Upon joining, a peer connects will all other peers. We could instead contact a fixed number of peers and allow any peer to initiate a connection with any other peer.
- We assume that all the nodes (tracker + peers) have to be on the same network, otherwise the ports will be all wrong due to network address translation.

Implementing any of these features would improve the speed, scalability, or robustness of the application.