Robert Sehringer
Rodrigo Pacheco Curro
John Jordan

Our program uses a TorrentHandler instance to coordinate pretty much everything in the process of taking a single-file torrent and getting the file we want. This TorrentHandler is created right in RUBTClient's main method, and will control the flow of execution once start() is called. The TorrentHandler needs to have a broad view of the entire process because it is ultimately in charge of making sure we have a downloaded file at the end. To this end, it will create instances of classes that will abstract parts of the process (like peer connections, tracker communication, and message types). The general flow is:

1) TorrentHandler decodes the info file using GivenTools package, then creates a Tracker object from the extracted info, a buffer to hold completed pieces, and a Writer object to write those completed pieces to disk.
2) The Tracker builds up the announce URL, establishes the HTTP connection, and then stores the trackers response in a decoded Map for easy access.
3) Using the Map, TorrentHandler creates a peer list. It then plucks out the peer info with the RU id, and creates a Peer object. The Peer and the TorrentHandler are linked through the PeerDelegate interface that allows the Peer to access certain TorrentHandler methods on reaching certain events.
4) Peer creates a Handshake object and tries to implement the handshake with the remote peer. If it succeeds, it tells the TorrentHandler using the interface and now our send/receive loop can begin.
5) Peer starts reading on it's socket buffer. When it successfully reads a message it uses the interface so that TorrentHandler comes up with a response to send. The TorrentHandler will check the SH1 hash of any completed pieces the Peer has passed up to it. Upon sending the response, execution ends up at the beginning of the startRead() loop.
6) Repeat step 5 until we've received all our pieces. TorrentHandler lets the Peer disconnect, uses Writer to write the file in its entirety (will be changed for future phases), and then uses Tracker to announce the completion. We're done at this point!

Classes

Message.java

Holds an enum Message types. Handles everything we need to encode and decode peer messages. Encoding is done by constructing a byte array using the type of Message we want (HAVE, PIECE, UNCHOKE, etc.) and the “payload” array (referred to as “tail” in the code). Decoding is done by creating a MessageData object using the types defined here and then reading from MessageData members.

MessageData.java

Holds messageData objects and constructors for creating them. Uses methods from Message to accomplish this encoding. We use instances of this class to shuttle around messages to the Peers and TorrentHandlers. This will definitely be changed in the future to a more elegant solution, but as of now this solution works.

Peer.java

Abstracts our P2P connection. A Peer maintains state about the connection, is in charge of sending and receiving messages, and has an interface to communicate to the TorrentHandler that spawned (known as a PeerDelegate). This way, decision making is done by the TorrentHandler which will eventually be serving and requesting from multiple Peers. Peers start by creating and sending a Handshake to the peer and then verifying the response Handshake before letting the TorrentHandler take control of the decision making. From there, it enters a feedback loop with the TorrentHandler in startReading(); upon successfully reading a Message from the peer, execution passes to the TorrentHandler (specifically in didPeerReceiveMessage()) which evaluates how to respond. Once it has generated the right Message response, TorrentHandler calls send() on the Peer with the appropriate Message, and then execution jumps back to the startReading() loop.

Handshake.java

Nothing too out there, just holds a Handshake object, provides constructors, and encode/decode. Similar enough to Messages.

PeerDelegate.java

An Interface that TorrentHandler implements to allow “communication” between it and the Peer. This allows our Peers to invoke methods in TorrentHandler upon certain conditions, like upon successfully performing the handshake or upon reading a Message. For future phases, this will probably be replaced with an event system to allow for multiple Peers communicating with a TorrentHandler in a threaded manner.

TorrentHandler.java

Our “Torrent” abstraction. Given a path to a .torrent file, this is in charge of making sure we have the file(s) described within downloaded at the end of the day. It will decode the .torrent using the tools provided, create a Tracker instance to communicate with the tracker, create Peers from the peer-list, implement the peer-messaging protocol to download the file, verify the hashes of each downloaded piece, use a Writer object to write the file to disk, and then tells the tracker we’ve got the complete file. That’s a lot.

Tracker.java

Communicates with the tracker. An instance of this object is created by the TorrentHandler using the info extracted from the .torrent. This object is in charge of building the announce URL, opening a URLConnection, and then extracting the trackers response into a map. The TorrentHandler will use this class to communicate with the tracker at the beginning and then at completion to notify that the file is complete. For the next phases, the TorrentHandler will announce more frequently since there will be more peers.

RUBTClient.java

Pretty simple, it creates a TorrentHandler from the shell arguments and then lets TorrentHandler do the rest. It also has a generatePeerID() method that will be called during TorrentHandler’s initialization.

Writer.java

Used by TorrentHandler to write our completed and verified pieces to disk. It’s pretty flexible at the moment, it can take any given PIECE message and write the payload block to disk in the right location(since we can assume that block size = piece size for this phase). Otherwise, it can write a ByteBuffer input, or even a byte at a time. In the future this will also handle reading in order to upload. For future phases we’ll also have to decide how often to

write, how to pipeline writes to avoid excessive seeking, and other challenges. But for now, TorrentHandler just writes the file all in one go.