Dylan Murray 128-00-3091
Jonas Schrieber 131-00-7364
Charles Zuppe 137-00-8995

RUBT Client – Part II

Project Overview

Our program starts in the main method of RUBTClient. RUBTClient verifies command line arguments and creates a Manager to manage the overall flow of the program. The Manager interfaces with Tracker to retrieve a peer list, creates Peers from this list that store all of a unique peer's state information (choke, interested, etc) and constants (ip address, port, etc), and creates a MessageHandler to interface with each Peer by sending Messages (including specific kinds of messages, eg Have, Piece, etc). See the attached UML Diagram.

Type Description

Our program makes use of various classes. We strove to be as modular and organized, as well as logical, as possible. The names and descriptions of these classes are listed below:

RUBTClient:

RUBTClient contains our program's main() method. It is responsible for verifying the command line arguments, populating a TorrentInfo object with information extracted from the .torrent file, generating a peerID, and creating a Manager object to handle the torrent download. We chose to separate the RUBTClient and Manager classes like this to add extensibility for multiple managers handling multiple torrents.

Manager:

The Manager class controls the overall flow of the application. Upon creation, the Manager sets all its associated fields, then proceeds to create a Peer object for each peer referenced by the tracker. For each Peer, there is a MessageHandler Thread which interfaces with one, and only one, peer. It generates a SocketListener object, which, as its name suggests, listens on a socket for any incoming connections. Manager also creates a FileAccess object, which contains a RandomAccessFile attached to the file specified in the command line arguments.

FileAccess:

FileAccess is a class responsible for all file operations. When a Peer requests a Piece, FileAccess is responsible for fetching that piece. When a MessageHandler receives a Piece, FileAccess is doing the work. When we resume our client after quitting without finishing the download, FileAccess tallies the pieces we already have and informs Manager of its findings.

SocketListener:

SocketListener is a class which is responsible for listening for incoming connections on a specific port. If utilizes knock knock protocol, and when it receives a handshake, it parses that handshake to verify hash info. If this checks out, it adds the client initiating the handshake to an ArrayList of Peers, named potentialPeers. This is passed back to Manager, which decides whether the Peer is already in our list of Peers. If not, it is added to the list.

RuntimeCommands:

RuntimeCommands is a class which implements Runnable responsible for taking user input. In this case, there is only one valid command, "quit." When this command is received, there is a call to a Manager method responsible for disconnecting all peers, notifying the FileAccess class that the file must be closed, killing off all threads currently running, and exiting the JVM.

Tracker:

The Tracker class handles all tracker connection related processes. It generates the GET request sent to the tracker, receives the tracker response, and decodes the response and extracts all its information: namely, an interval time to which we need to wait to contact tracker, and a list of peers in the swarm.

Peer:

The Peer class contains all relevant Peer state information (such as choked, interested, is choking, is interested), unique Peer information, such as the peer's socket, ip address, and port number, and most importantly contains what pieces the peer has that the client needs.

MessageHandler:

The MessageHandler, which implements Runnable, is instantiated on a one-to-one basis with Peer Threads, from the Manager Class. It generates, and verifies, handshakes, fields messages to and from a Peer to whom it has been attached, and also divides pieces larger than 16KB into smaller sub-Pieces.

Message:

The Message class represents a message that's sent between peer to peer. It also has static methods that can be used to easily encode and decode data from an output or input stream. The message class as an object is only used to represent Choke, Unchoke, Interested, and Uninterested methods. The other types of messages are specialized and are subclasses of Message.

BitfieldMessage:

The Message that represents a p2p bitfield message. A subtype of Message, a byte[] bitfield was added, and it will eventually override the method that generates a byte[] torrent protocol encoding of this message (it does not for Part I because we do not have to upload any pieces, and thus do not have to send a BitfieldMessage).

HaveMessage:

The Message that represents a p2p have message. A subtype of Message, an int index field was added. It overrides the method that generates the byte[] torrent protocol encoding of this message.

RequestMessage:

The Message that represents a p2p request message. A subtype of Message, fields were added to store index, offset (begin), and download length. It overrides the method that generates the byte[] torrent protocol encoding of this message.

PieceMessage:

The Message the represents a p2p piece message. A subtype of Message, fields were added to store index, offset (begin), and block. It overrides the method that generates the byte[] torrent protocol encoding of this message.

Other Classes:

In addition to these classes, we also used the classes provided to us on the Sakai page: Bencoder2, BencodingException, TorrentInfo, and ToolKit.