Bittorrent Project #2

Group 12

Rohit Kumar (*rsk120*), Justin Rokisky (*jrokisk*), Akhilesh Maddali (*amaddali*)

High Level Explanation

Program execution begins in where a Torrent object is created, and Torrent's initialization method is called. At this point, the program checks if the downloaded file already exists on the disk, and if it does, it rehashes segments of the file in order to discern what pieces have already been completed. In addition, a list of Peers is created based off the response from the Tracker. All peers are handshaked, connected to, and begin running on their own Threads. At this point, any uploading can happen if a piece requests something we have. A Piece —> List { Peer } is created for downloading. Downloading works by checking if a piece has a Peer that is not already requesting some pieces, and if an idle peer is found for this piece, a the requests are started. All writing to a Peer's output is done a seperate thread, and Messages are communicated to this thread with a synchronized queue.

Class descriptions

Message.java The Message class is an abstract class that gets extended by all of our message objects. It is responsible for decoding incoming messages and encoding outgoing messages. It takes care of all of the reading and writing to/from the connection we have with each peer.

BitfieldMessage.java HaveMessage.java PieceMessage.java RequestMessage.java These classes extend the Message class and represent specific messages. Each has getter methods to pull the needed information out of each message.

Protocol.java This class contains constants. (we should probably merge this somewhere).

RUBTUtil.java This class contains helper methods that are used for generating random-ish Peer IDs, escaping bytes, and dumping data.

Tracker.java This class is responsible for all communication with the tracker. It has methods to send announces to the tracker, decode the trackers response, and to pull a peer list out of the response. It also has a method for getting the tracker's request interval.

Piece.java This class is responsible for storing all subpiece data of a piece until the client receives the whole piece. When this occurs, it has a method for comparing the SHA–1 hash of the piece to the correct SHA–1 hash. It also has accessor methods to get the length of the piece, the current buffer, and "clean up" methods for when we have finished downloading a piece or when the download or hash verification has failed.

FileHandler.java This class is responsible for all file IO in the client. When the program is launched, it checks if the file where we wish to save downloaded data to exists. If it does exist, the file is rehashed to

check which pieces the client has already downloaded. FileHandler also has methods for writing pieces to a file, and for getting subpieces from the file. FileHandler is also responsible for writing and reading to/from the statistics file, which is used for storing Uploaded and Downloaded between runs of the program.

RUBTClient.java RUBTClient is the main class of our program. It is responsible for printing usage, reading user input, and running the client's download/upload methods.

Peer.java This class is responsible for all interactions with a specific peer. Each peer that we are connected to is represented by its own Peer object. Peer has two different constructors: creating a peer object from tracker information and creating a peer object from an incoming connection. Peer's main method is responosible for handling all incoming messages. Peer has methods for handshaking, sending messages, connecting and disconnecting, and numerous getter methods. Also, each peer has a "request queue" where messages are stored until they are added to the Peer's peerOutThread message queue. This is necessary for handling problems with requesting pieces.

PeerOutThread.java This class is responsible for sending messages to a peer. Each peer has its own PeerOutThread. PeerOutThread has a private keepAliveTimer class for sending keep-alive messages to a Peer at specified intervals. Also, there are messages for shutting down the PeerOutThread and for getting the request queue.

Torrent.java Torrent is responsible for doing the majority of the work in our program. It has methods to: pull data from the given torrent file, schedule regular tracker announces, download the desired file, organize the order in which messages are sent to peers, and respond to peer messages. It also has numerous getter methods and a shutdown method that is invoked when the user enters "q" or when the program is shutdown.