Theory of Operation

In this project, we were assigned to build a simple BitTorrent(BT) client which will later serve as the foundation for a fully-featured BT client. Our client is able to open a .torrent file, contact the tracker as well as multiple peers and be able to upload and download to the peers simultaneously.

Assignment Overview

Your assignment should basically do the following:

1. Take as a command-line argument the name of the .torrent file to be loaded and the name of the file to save the data to. Your main class MUST be called "RUBTClient.java", but may reside in any package. For example:java RUBTClient somefile.torrent file.mp3
2. Open the .torrent file and parse the data inside. You may use the Bencoder2 or TorrentInfo classes to decode the data.
3. Contact the tracker via the announce URL, including all of the necessary key/value pairs in the request. The java.net.URL class is a convenient way to accomplish this.
4. Tracker announces should be performed no more frequently than the value of "min_interval" (or 1/2 "interval" if "min_interval" is not present), and no less frequently than twice the value of "interval" returned by the tracker.
5. Your client must correctly publish its connection information and status to the tracker. This includes the port, uploaded, downloaded, left, and event arguments, when applicable. Your client must accept incoming connections from other peers. It should not maintain more than one TCP connection with another peer.
6. Capture the response from the tracker and decode it in order to get the list of peers. From this list of peers, *use only the peers located at 128.6.171.130 and 128.6.171.131* . You must extract these IP addresses from the list, hard-coding it is not acceptable, except the comparison itself.
7. Open TCP connections to the peers and be able to download and upload **simultaneously** from several at the same time.
8. Time the download and output the total time of the download. You can make use of System.nanoTime();
9. Download one or more pieces of the file and verify its SHA-1 hash against the hash stored in the .torrent file.
10. After a piece is downloaded and verified, the peers are notified that you have completed the piece.
11. Other peers should be able to request pieces from your client that you have verified, and your client should send those pieces to the peers, according to the protocol. Remember that you should be able to serve multiple clients simultaneously.
12. Repeat steps 5-8 (using the same TCP connections) for the rest of the file, allowing new clients to connect. The client should continue uploading until the user provides the appropriate input to shut down the client.
13. When the client has downloaded and verified the entire file, be sure to contact the tracker with the *completed* event (even if it is shorter than the *interval* value).
14. When the client exits, you must contact the tracker and send it the *stopped* event and properly close all connections.

In addition to the above basic run-through of your program's execution, your program should be able to detect input from the user at any point (you can decide what the input should be), and

allow the user to exit the program, saving any verified pieces of the file. The user should be able to restart the program at a later time and resume the download. Any non-verified pieces/blocks should be discarded before exiting, and the program should also keep track of how much it has uploaded and downloaded across sessions for that torrent.
How you choose to implement saving the state of your program is up to you, but an intuitive approach might be to allocate the total space to disk before downloading, and then writing the appropriate pieces into the file as they complete.