# Project 2: BitTorrent

**Description:**

This project implements a simple BitTorrent, allowing users to download and upload files from their connected peers. To begin downloading, it is important to acquire a torrent file, which is a small file that tells the client the information necessary to download the content desired. The client (which is implemented in our RUBTClient class) will receive data on the tracker by parsing the torrent file. Once connected with the tracker (which is mimicked in the tracker class), the client should extract the list of peers from the tracker response (exectuted in the DownloadManager class ). In order to achieve simultaneous downloads and uploads, it was important for us to use multithreading for multiple-peer connections, which is handled by Incoming\_Controller class. The Peer class holds all of the peer’s information and extends Thread. The messages that peers send to one another are stored in the Messages class. The biggest addition to this version of the project, was fulfilling two more functionalities. The first change required pieces to be requested based on piece rarity and this is done in the Rarity class. The second change was establishing a choke strategy in the Optimistic\_Unchoker class which chokes all, but 3 connections based on uploading rates.

**Classes:**

***RUBTClient.java***

This class is the main class. It creates a client and parses the torrent file. In other words, a TorrentInfo object is created with the meta data of the torrent file. This way, necessary values are initialized to execute the program (e.i. peer ID, torrent file name, output file name, downloaded/uploaded information, etc). It also declares and initializes the tracker. Using the data extracted from the TorrentInfo object, the client makes a connection with the tracker. Upon successful connection, it starts the DownloadManager. It also includes methods allows saving the completed file into the disk.

**Tracker.java**

This class resembles a tracker. It has the necessary information and methods to connect the client with the tracker and decode the tracker’s response. This is done by using a method which creates the HTTP URL connection with the data provided by the meta data (TorrentInfo object) and tracker’s URL announce. This class also has a method which extracts the interval and peer list from the tracker’s response. The biggest change from the previous version is making this class a Thread and making it sleep for the tracker interval amount of time.

**Peer.java**

This class resembles a peer. It holds all of the peer’s information, such as the peer ID, peer IP, peer Port, etc. It also houses methods which take care of handshaking between peers. Moreover, this class extends the Thread Class. Each peer will be spun in a thread, so that peers can simultaneously download or upload. It begins with executing the handshake between client and peer. It also sends an interested message if the connecting peer has a piece that we are interested in, otherwise an uninterested message is sent. (The choking and un-choking is taken care of in Optimistic\_Unchoker). If downloading, the file is downloaded sequentially thereafter by sending messages for pieces and have messages when the piece is completed. The output file is regularly updated as the pieces are downloaded. When all is done, all sockets are closed. To be precise, these messages, like pieces, have, etc, are used from the Message Class. This keeps the Peer class from being too cluttered and helps organize our code. Upon successful execution, the peers would either have uploaded or downloaded the torrent without failure.

**DownloadManager.java**

This class takes care of going through the tracker peer list and extracting the peers into an a String Array. It makes sure that all peer threads has a bitfield message before allowing them to request pieces. This class is using the Rarity class to request pieces based on rarity of piece. It also contains various methods that help facilitate downloads, such as the ability to determine whether or not a piece of file has been downloaded, the ability to broadcast to all peers that they have a piece, the ability to save a piece, etc.

**Incoming\_Controller.java**

This class takes care of the incoming connections from peers. It also checks the peer list and reads in the handshake from new peers. It is running a Thread and connecting through a Server Socket. Its main purpose is to act like a mini server to handle connections. This is where all the messages from peers are handled and tells the program what to execute based on what message was received.

**Messages.java**

This class includes all the messages the peers will use to interact with one another, including, keep-alive, choke, unchoke, interested, uninterested, have, request, and piece. They are static functions that are called from Peers.java.

**Optimistic\_Unchoker.java**

This class takes care of choking and unchoking peers. It chokes all but 3 connections based on upload rate. Thus our “worst” peers are those that have lower upload rates. It also randomly unchokes a peer. Priority, however, is still given to peers with the highest upload speeds while downloading.

**Rairty.java**

Rarity class helps organize peer requests based on piece rarity, where the rarest piece should be given the higher priority. This is used in the DownloadManager class. It creates array list of peers who share a piece with the same index. This helps keep track of how many peers has the piece at a particular index. Thus the rarity of a piece is based on the counter that is calculated by summing up (keeping track) all the peers that have that one specific piece and comparing it to the amount of peers that have another piece. The smaller count of the two, would be the “rarer” one.