**URTorrent**

Ben King

(bking11@u.rochester.edu)

**In this Project:**

In this project I implemented the URTorrent protocol mostly according to the spec on the website. It was a bitch to code but I think it works basically up to spec. Hopefully this README makes testing it pain free.

**Project Roadmap:**

This program handles only one torrent file at once. There is a server which detaches a thread for each new connection, and each peer is encapsulated in an object which contains its ip, port, choke status, connection, etc. One challenging item is that you don’t really know who a new connection is coming from, so at times I have two separate peer objects for the same peer which are later merged after I handshake.

The program asynchronously requests new pieces (in block size 2^16) every .5 seconds using a rarest first strategy. It keeps track of which peers have which pieces (not counting choking peers) and picks the rarest piece to download. It updates interest to its peers but it does not use this information. While it will not request a piece from a choking peer, it does not use any logic in choking/uncoking—it just unchokes everybody.

**Functions of Interest**

These are functions that I have commented that may be of some interest:

* **download\_from\_peers():** This function implements the rarest-first strategy in selecting a piece to download
* **Peer.listenToClient():** This function takes an input packet and executes the correct function according to the message id
* **Peer.download\_block():** This function splits the piece into 2^16 byte blocks and downloads them from the peer.
* **handshake\_new\_peers():** This function will check back with the tracker to see if there are any new peers that have joined. If there are it will handshake with them.
* **check\_hash():** This function verifies that the downloaded block matches the hash from the torrent. If all the pieces are downloaded it will reassemble them into the file.

**Libraries Used**:

I used bcode, a Python github repo for encoding/decoding bencode files. It is included in the project. I use MMT for generating metainfo files, not that urtorrent.py needs that. I use BitTornado for running a tracker rather than open tracker in testing, it is also included. For the http request to the tracker I use the requests module which exists on the cycles.

**How to Run:**

python3 <portno> <torrentfilename> [<debug level>]

* no debug level defaults to 1, which is only shows initial connections and the completion of the download. 2 lets you see every message that happens but you won’t really be able to get in commands. 0 prints nothing except for completion, you will need to use commands for debug.

**How to Test:**

**Recommended:** Open up a tmux session with five windows. In one cd to BitTornado/ and run:

python3 bttrack.py --port 6969 --dfile poop.txt

This launches the tracker. On the other five windows cd to u1/, u2/, u3/, u4/ etc. and run:

python3 urtorrent.py 700x treatmelikeapirate.mp3.torrent 2

Note that the port should be different for each one. Maybe wait a second between starting each peer in case there is garbage in the tracker’s saved state and the peers fail to connect at first. Debug level 2 is nice to watch but you can use 0 to test the commands.

Once the downloading is complete output\_treatmelikeapirate.mp3 will be in each peer’s folder for your listening enjoyment.

**Not recommended:** There is also a runscript in the main directory that you can run with:

python3 runscript.py [<N>]

* where N is the number of peers to create, default 4

Note that in this case, all the peers are backgrounded so you will not be able to interact, all the outputs will print to the same console, and to close them you must kill them e.g.:

pgrep -f urtorrent.py | xargs kill

pgrep -f bttrack.py | xargs kill

These commands are a bit sketch to run since they just match any process so be careful.