

BitTorrent Client

Dependencies

```
pip install bencode.py alive-progress bitstring
```

- Bencode - Lightweight, binary encoding format in the BitTorrent protocol for encoding metadata associated with torrents.
- Alive-progress - Live progress bar for monitoring download.
- Bitstring - Provides bit manipulation for bitfields.

List of supported features

Client Options

Run the client script with the `-h` / `--help` option to view the entire BitTorrent client option set:

```
python3 bt-client.py -h
```

Mandatory Options

Complete Example:

```
python3 bt-client.py -t artofwar.torrent -p 6888
```

- `-t TORRENT` - Specied .torrent file to parse and file to download
- `-p [1024, 49151]` - Specied port which your client is listening on. BitTorrent ports are *typically* [6881, 6889]

Optional Options

Complete Example:

```
python3 bt-client.py -t artofwar.torrent -p 6888 -c -w 40 -u -d -q
```

- `-c` - Indicates that the client accepts a compact response from the tracker
- `-w NUMWANT` - Number of peers that the client would like to receive from the tracker. If omitted, defaults to 50 peers.
- `-u` - User may manually opt for support using a UDP-tracker protocol.
- `-d` - Outputs real-time log of client behavior into console for details.
- `-q` - Informs the client to automatically disconnect from the tracker once complete download has finished; otherwise, it will remain part of the swarm as a seeder for peer downloads.

Design and implementation choices that we made

UDP Tracker

I thought that communicating with the tracker without an HTTP library was unnecessarily tedious; however, I believe that it was not nearly as involved as opting for a UDP-tracker protocol. Since UDP is 'unreliable' there was a series of measures that I had to take in order to ensure accuracy and reliability between my client and the tracker. However, in doing so, there was significantly less overhead. In following the online BitTorrent specifications, I essentially created a 'de facto' state-maintained protocol by sending additional requests as `action=connect`, allowing the tracker to know who I am via a unique `connection_id`. Although, I would have to refresh this id upon timeout or multiple ReX's of announcement or connection requests, it facilitated adequate communication between us. This protocol essentially acted as an upgrade to the 'compact' option, reducing traffic by a significant amount (a 'super-compact' if you will).

Blocks and Pieces

For managing pieces, we decided to have a work deque of piece indices, which keeps track of what piece is undownloaded and hands out the responsibility of a piece to some peer. We decided that each peer should be an object that handles its own socket, which means handling incoming data from that peer and sending requests to said peer. We found that having 1 thread doing the work for every peer is extremely hard, given how many things it would need to keep track of. Each peer would have a listening thread, designed to listen for and handle any messages, and a request thread. The request thread's only job is that while our client is unchoked and the download is not finished, it grabs an index from the work deque and check if the current peer has it. If the peer doesn't, the thread appends the index back to the work queue and wait 2 seconds before trying again. If the peer has the piece, the thread sends requests for blocks of size 16000 bytes in the piece. The maximum number of unresolved requests (meaning requests with no piece message response yet) is $\min(5, \text{numBlocks})$, where `numBlocks` is `pieceLength/16000`. This ensures that we play nice with BitTorrent 16KB message size specification while implementing pipelining.

Unchoke Strategy

For every 10 seconds, from all the peers whom we're downloading from, I take a note of the total data received within the last 10 seconds. Then, I rank the peers by the data received. I take the top 3 peers and unchoke them if they're not already unchoked. Every 30 seconds, I take a random peer who was also choked, and unchoke them. This is done repeatedly to ensure reciprocation with our biggest benefactors.

Problems that we encountered (and if/how we addressed them)


400 Error HTTP Mangled Header

When attempting to communicate with the class tracker by sending an HTTP GET request, I would consistently get a 400 Error response. I recognized that this was a problem with the request message my client was sending; however, I couldn't find out exactly what it was. I tried numerous different byte-string order combinations, but none would successfully fall through. This had me stuck for a while, even attempting to reference sample HTTP requests from online, but none worked. The solution I found to this problem was to install a working *Transmission* client and use WireShark to intercept the appropriate HTTP packets when downloading a file using the .torrent. I was able to reverse-engineer/construct a matching request - albeit one that is tailored for my client's needs - that was successfully recognized by the tracker, and I received a response.

Variety of Tracker Response Formats

For most tracker responses, I was able to bencode-decode them and extract the appropriate information about the swarm from the peer list, etc. However, on some instances, the response that I received didn't fit the standard format. (In this context, I am not talking about 'Compact Responses'; I was able to perform some magic to appropriately decode and sort compact peers.) I was able to identify two different types (excluding the standard): 'Mangled Responses' and 'Chunked Responses'. I identified 'Mangled Responses' as those that sent incomplete, bencode-incompatible, and messy data. Here is a response I managed to capture:

```
b'HTTP/1.1 200 OK\r\nContent-Type: text/plain; charset=utf-8\r\nDate: Fri, 08 Dec 2023 05:24:56 GMT\r\nContent-Length: 1665\r\nConne
```



Notice how it ends abruptly and even truncates one of the peer's ip address. To handle these types of responses, I simply catch it, notify the user of the mistake, and attempt gracefully exit by sending an appropriate 'stopped' event. Any attempt at immediate retransmission would cause us to be booted because < response interval.

I identified 'Chunked Responses' as those that specified "Transfer-Encoding: chunked" as part of the message. Here is a response I managed to capture:

```
b'HTTP/1.1 200 OK\r\nContent-Type: text/plain; charset=utf-8\r\nDate: Fri, 08 Dec 2023 05:31:52 GMT\r\nConnection: close\r\nTransfer
```



Notice how the bencoded chunk is prefaced by its byte length (in this case it is 80c, which is 128 bytes for this chunk). To handle these types of responses, I continually receive chunked tracker responses until one of the chunks is of size 0, indicated the end of the data.

Selector Socket Management vs Multi-Threaded Functionality

Initially, we utilized a multi-socket selector with simultaneous data receiving and timeout; however, we realized that this would cause a variety of issues regarding the behavior of our client when receiving and sending messages. More specifically, it significantly complicated the interactions we maintained states we carried on a per-peer basis. So, we decided to reimplement this part to establish threads for each peer to isolate each message receive and response without worrying about any messy interactions.

Known bugs or issues in our implementation

Occasional Non-Graceful Tracker Disconnection

Although I had implemented a "graceful disconnect" from the tracker by transmitting an appropriate response (event=stopped), the program would still notify me of an errors (e.g., BrokenPipeError, etc.). I believe that this is due to the multi-threaded nature of our client, but I am not so sure. As long as our client gets removed from the swarm, I think we should be fine, although I wish it could be more smooth.

Hidden Terminal Cursor

I don't know if I'm just going crazy, but every time I finish running the client, the cursor in my terminal disappears lol. I suspect that this has something to do with the alive-progress module, although I am not certain.

Random Connection Refused

On some instances, the server refuses our connection? Running it identically after doesn't spit out the same exception. Weird.

The download is really slow

The download maxxex out at roughly 300KB/s

Multi-file Download

In attempting to download multi-file torrents, there was some completed arithmetic magic that was attempted; however, it was ultimately unrealized. There was too much work to be done with handling blocks and pieces that could be requested while also being split between files within the same block or piece. If there was more time, it would have definitely been finished.

Contributions made by each group member

Snehal Tamot

- Handshake messages
- Bitfield messages

Baudouin Zuzuh

- Tracker interaction using HTTP with support for compact responses
- A Peer class that implements the peering BitTorrent protocol
- Piece manager for splitting, verifying, reading, and marking pieces
- Torrent parser
- A file manager for reading and writing pieces -- for single-file torrents
- Some of my contributions were duplicate efforts, but were needed for a holistic view of the protocol

Phan Nguyen

- Handling Handshake Messages
 - recv handshake
 - send back handshake
 - send back bitfield
 - register new peer as a peer object
- Peer logic
 - Automatically determine if self is interested in peer
 - Send interested/ not interested messages
 - Send out have messages if a piece is received and hashed correctly
 - Set up a thread for listening to incoming messages
 - Set up a thread for requesting pieces
- Handle sending data to Peers:
 - Implemented a Choking Algorithm that unchokes peers with highest download speeds
 - Optimistic Unchoking
 - Sending data from requests and support cancel requests
- Sending stay alive/ disconnecting from peers if not alive

Joshua Nguyen

- Client Option Set
- User Display/Interface
- Tracker Interaction
 - Connection Medium
 - HTTP
 - UDP
 - Handling Tracker Requests & Responses
 - Compact Responses
 - Chunked Responses
 - Mangled Responses
 - Regular Responses
 - Announcing & Scraping
 - Detached Thread Periodic Messaging
 - Handling Tracker Failure/Warnings
 - User Disconnect
 - Graceful Disconnect
 - Automated Disconnect
- Bencode De/Encoding
 - Parsing torrent file information
 - Parsing tracker message data
- Downloading Piece Data
 - Managing Blocks in Pieces & Pieces in File
 - Hash Verification
 - Writing to file
 - Mangled Piece Data
 - Duplicate Blocks/Pieces