How to Write a Bittorrent Client, Part 1

Posted on **November 23, 2012**

I spent the first few weeks of Hacker School writing my own client utilizing the Bittorrent Protocol, and thought I would share some of the things that I learned on the way. This post will cover a general outline of how to approach the project, with a focus on downloading torrent files and a bias toward python.

This post will be broken into two parts, of which this is the first.

1. Read the unofficial specification here.
   No, really. Read it. There is an official spec as well, but it is vague and much less helpful. Understanding the spec will make this project far easier for you going forward.

2. Play around with Wireshark.
   I also recommend downloading an already-written bittorrent client (I can recommend utorrent or the official bittorrent client – both from the same code base). Then download a .torrent file. These can be found in many places online. Mininova is a good place to look for legal (I think) torrent files.

   Launch a Wireshark session and open your .torrent file in your bittorrent client (on the clients mentioned above, go to File, Open Torrent, and select your .torrent file). While your torrent downloads, you can watch the packets and messages being sent to/from peers with Wireshark. Filter the Wireshark results with the keyword 'bittorrent' and you can see just the bittorrent messages. When your torrent is finished downloading, you can stop wireshark and save your session for later analysis. This information can be helpful in both understanding the spec and also for comparison later when you run into message-passing bugs. Wireshark is cool tool to learn about other network traffic as well, and I encourage you to play around with it.

3. View your .torrent file in a text editor.
   You'll notice that there is not really a whole lot here. This is NOT the actual file you want to download. Instead it is a metafile containing information that you will need in order to download the real file.

   See how it starts with a 'd' and ends with an 'e' and has plenty of funny '#:word' sections? That's called bencoding. Bencoding is an encoding that translates a complex set of embedded dictionaries, lists, strings, and integers into a single string. There is an explanation of bencoding in the unofficial spec here. This is good to understand and be able to read.

   You'll likely want to decode the torrent file and save much of the information that is stored there

for later use. In particular, you will at least need the 'announce' url and 'info' dictionary, and within the info dictionary you will need the 'piece length', 'name', 'pieces' (hash list), and 'paths' and 'lengths' of all individual files. Note that the structure is slightly different for single file vs multiple file torrents. Again, the spec is helpful for explaining the different tags and structure. If you are using python, note that there is a good bencode 3rd party library that can do the encoding/decoding for you. (pip install bencode)

4. Connect to the tracker.
   The 'announce' key in the .torrent metafile gives you the url of the tracker. The tracker is an HTTP(S) service that holds information about the torrent and peers. The tracker itself does not have the file you want to download, but it does have a list of all peers that are connected for this torrent who have the file or are downloading the file. It responds to GET requests with a list of peers.

   To send a properly formatted request to the tracker, you take the announce key mentioned above as the base url and add certain parameters to the url in the format of 'announce-url?param=value&param=value&…'. The url must be properly percent encoded using the "%nn" format, where 'nn' is the hexadecimal value of the byte or reserved character. Unreserved characters need not be escaped (see link for reference). For example, the escaped form of the binary string '\xab' is '%AB' and the escaped form of '\x12\x34\x56\x78\x9a' is '%124Vx%9A'. In python, the Requests library will take care of this for you ('pip install requests'). The required parameters are listed in the unofficial spec here. Of note in the parameters are the:

   - 'info_hash' which you compute as a hash of the bencoded info dictionary from the .torrent metafile using the SHA1 hash algorithm. The python documentation for the hashlib library has more details about hash algorithms. Note that you should not compute this on either the bencoded full torrent file nor on the decoded info dictionary – this should be computed on the bencoded info dictionary only. You can parse the info dictionary out of the original torrent file or re-bencode the decoded info dictionary. If you are using a language with unordered dictionaries (such as python), be careful if you re-bencode the info dictionary that you make sure the dictionary values are appear in sorted order or you will get an incorrect SHA1 hash. The bencode python library will take care of this for you.
   - 'peer_id' can be anything you want that is 20 bytes long – there is a section in the spec for suggestions for peer id formats.
   - 'left' – when you are downloading a file for the first time, 'left' should be the total length of the file. The .torrent metafile does not give you total length if it is a multi-file torrent, but it does give length of every file expected (the 'length' fields) and you can compute total length from that.

5. Parse the tracker response
   Assuming your GET request is formatted correctly and contains the correct info_hash, the tracker should send you a response with a text document containing a bencoded dictionary. The expected keys of the dictionary can be found here. The 'peers' key will contain information about

the peers we can connect to for this file. Once we parse these into ip_address:port strings, we can use them to connect to the peers. Note if you get the peers in the binary model that the last two bytes together encode the port number (i.e. '\x1a\xe1' = 26 * 256 + 225 = 6881).

6. Connect to peers
   Peer connections are made through TCP to the appropriate host ip and port. Now might be a good time to consider how or if you want to deal with connecting to multiple peers at the same time, as this will influence how you connect to your peers. Some options are:

   - The Twisted framework if you are using python. This framework implements event-driven programming and abstracts away many of the lower-level details (pip install Twisted). This is what I used.
   - Create your own event-driven programming loop using sockets and select calls.
   - Multi-threaded sockets. Good luck!

   You can revisit this later and just work on connecting to one peer first, but eventually you will want to consider how you wish to handle multiple simultaneous peer connections.

7. Handshake with peers
   Once you have a connection to your peer(s), the first contact step is your responsibility. The first message you send should be a Handshake. Parameters for this message are here. The info_hash and peer_id we have seen before. For the current protocol version (1.0), 'pstrlen' = 19 and 'pstr' = 'BitTorrent protocol'. 'Reserved' is 8 bytes long. Unless you want to support extensions to the protocol, these bytes should all be zeroes ('\x00').

   The message you send the peers is the values for the 'pstrlen', 'pstr', 'reserved', 'info hash', and 'peer id' combined into one long byte string. Structs are a great way to deal with moving to and from bytes and numbers/strings in python.

   The peer should immediately respond with his own handshake message, which takes the same form as yours. If you received a peer_id in your tracker response, you should check that the peer_id provided by the peer in the handshake matches what you expect. If it does not, you should close the connection. When serving files, you should check the incoming peer's handshake to verify that the info_hash matches one that you are serving and close the connection if not.

Part 2 available here.