How to Write a Bittorrent Client – Part 2

Posted on **November 27, 2012**

This is part 2 of how to write a Bittorrent client. If you have not read Part 1 yet, you should take a look at it first.

Picking up where we left off, we now have successful connections to our peers with the torrent file of interest and have performed the initial handshake. Moving on to the message passing sections of the Bittorrent protocol:

1. Message Passing – Overview
   Now we get to the meat of the Bittorrent protocol. The unofficial spec describes 11 types of messages that bittorrent supports: keep-alive, choke, unchoke, interested, not-interested, have, bitfield, request, piece, cancel, and port. Descriptions can be found here.

   Messages consist of a 4-byte length, followed by a single byte message id (except in the case of keep-alive messages, which just have a 4-byte length of zero and no id – we will ignore these going forward). Some messages also have a payload of fixed or variable length following the id. Each type of message has its own unique message id, so, for instance, whenever the 5th byte of a message is '4', you know you just received a 'Have' message. The length bytes encode the length of the message that follows those first 4 bytes (so they do not include themselves in the length count).

   You will need to have a way to consistently create and parse messages into their respective types and access their payload sections as appropriate for each type.

   One more thing to note about message passing – there seems to be no guarantee that messages will come in discrete packets containing only a single entire message. This means that you might end up with a long bytestring from a peer containing several messages, or you might end up with a bytestring from a peer that only has the length prefix for a message and the rest of the message will arrive in a later packet. You will need some way to deal with this inconsistency. The length prefix can be very helpful here to determine how much data you expect to have.

2. Message Passing – Bitfield and Have
   Once you and your peer have exchanged handshakes, you will need to know what pieces of the file your peer has in order to know which ones you can request from this peer. There are two message types that a peer can use to tell you about what pieces they have. The 'Have' message type is simple, consisting of a 4-byte length prefix and single byte message id as mentioned above, followed by a 4-byte payload representing the piece number (0-based index).

A client can send you a series of Have messages, one for each piece it has. Alternatively, at the start of a connection, the peer can send a 'Bitfield' message. Bitfield messages are optional and can only be sent as the message immediately following the handshake message. Again, the message consists of the 4-byte length prefix, 1-byte message ID, and a variable-length payload. The payload for the Bitfield message is a way to succinctly describe the pieces that a peer has. Bits set to 1 indicate pieces the peer has, unset bits (0) indicate missing pieces. The bitfield payload comes through as a set of bytes. You can think of each byte as made up of eight individual bits indicating which pieces a peer has (i.e. '\xfe\xff' = 1111111011111111 (pieces 0-15, piece 7 is missing). Any spare bits at the end of the last byte are left unset (0).

Some clients, even if they have all pieces, will send an incomplete Bitfield message and then follow it up with Have messages for the pieces missing from the Bitfield. You should create some representation of this data for each peer you are connected to, so that you can check if the peer has a piece before requesting some part of it. In python, the bitstring.BitArray class is useful for this (pip install bitstring).

3. Message Passing – Choke/Unchoke and Interested/Not Interested
   The four message types 'Choke', 'Unchoke', 'Interested', and 'Not Interested' are used to indicate whether a peer will send you files (and vice versa).

   'Interested' means that the downloading client (that's you) would like to download from the peer. 'Choke' means that the peer serving the file will not send it to you until they 'Unchoke' you. All connections start off as 'Not Interested' and 'Choked'. In order to get to a state where you can receive files, you need to send your peer an 'Interested' message, and they need to send you an Unchoke message. You should wait for this Unchoke message from your peer before requesting pieces. Once you are unchoked, a client can still send you a Choke message at any time, at which point you should refrain from requesting pieces from that peer.

   Note that these choking and interested states work both directions. If you are serving a file for which you have some or all of the pieces, you also have to track states for whether you will serve the peer files. If the peer sends you an Interested message, you can decide whether or not you want to send an Unchoke and send them files. This is also why you might not initially want to send both an Interested message and an Unchoke message unless you are willing to serve files – sending the peer an Unchoke tells them you will serve requests.

   Upon receiving the peer's handshake (or after the Bitfield/Have(s)), you should send your peer an Interested message to let them know you would like to request and receive files from them. You should also wait until they send you an Unchoke message before sending any requests for pieces.

4. Message Passing – Request
   Once you have sent your Interested message and received an Unchoke message, you can start

requesting pieces! It turns out that the ‘piece_length’ assigned in the .torrent metafile is usually too long for a single piece to be sent all at once. As a result, pieces are split and sent in smaller chunks. Confusingly, the official Bittorrent Protocol specification also calls these portions of a piece, ‘pieces’. In an effort to be less confusing, I will call these sub-pieces ‘blocks’, as the unofficial specification referenced throughout this post does.

There is some dispute about what the range of acceptable sizes of blocks should be. See here for a portion of the discussion. The short answer is that if you use 2^14 (16384) bytes as your requested block length, you should be able to successfully request and download pieces from peers. Note that if the piece_length is not evenly divisible by the block request size, the last block of each piece will be smaller than the normal block size. It is likely that at least the last block of the last piece will be smaller than the block request size (and the last piece smaller than the piece_length) because the total torrent length is unlikely to be evenly divisible by the piece size. You will need to take this into account when requesting the end of the torrent. When serving files, if you wish to accept a larger range of block request sizes, see the discussion mentioned above for guidelines.

The ‘Request’ message type consists of the 4-byte message length, 1-byte message ID, and a payload composed of a 4-byte piece index (0 based), 4-byte block offset within the piece (measured in bytes), and 4-byte block length (probably 2^14, as mentioned in previous paragraph). You can be creative in the algorithms you use to determine the order of blocks to request (in order, most rare first, random, etc). Before requesting a block from a peer, you should first check whether the peer has the piece to which that block belongs. Requesting blocks from a peer who does not have them will not be a good strategy for downloading your torrent!

You may also wish to track the blocks which you have requested and (/or) those which you have received in order refrain from making duplicate requests. The bitstring.BitArray class mentioned under the Bitfield/Have section can be useful here as well. One option which can speed up your torrent downloads is to add a periodic check for requested pieces which have been outstanding for more than some set period of time, and re-request those pieces from other peers.

5. Message Passing – Piece
   A peer should respond to a Request message with a ‘Piece’ message that includes the block requested. Though the message type is called ‘Piece’, it includes the information for a block, not necessarily a full piece.

   A Piece message consists of the 4-byte length prefix, 1-byte message ID, and a payload with a 4-byte piece index, 4-byte block offset within the piece in bytes (so far the same as for the Request message), and a variable length block containing the raw bytes for the requested piece. The length of this should be the same as the length requested.

   The bytes provided in the block section should be stored somewhere in your program. When all blocks for a piece have been received, you should perform a hash check to verify that the piece matches what is expected and you have not been sent bad or malicious data. The ‘pieces’ element in the .torrent metafile includes a string of 20-byte hashes, one for each piece in the torrent. Note that this is NOT a list, but is a single long string. You should perform a SHA1 hash on the downloaded piece contents and compare that to the hash provided for that particular piece. If they do not match, you should discard the downloaded piece and request the blocks for it again.

If you wish to serve files as well as download them, you should send a Have message for the piece to all connected peers once you have the full and hash-checked piece.

6. Message Passing – Cancel and Port

   The last two message types are ‘Cancel’ and ‘Port’. Neither is strictly necessary to implement to have a workable Bittorrent client. Cancel messages are sent when one wishes to inform peers that a block you requested from them is no longer needed (i.e. you have received it from another peer). This message is most often used in what is called the End-Game: when a torrent is almost completely downloaded, you can optionally send requests for the last few outstanding blocks to many peers at once. To be polite, you should let those peers know when you have received the block by sending Cancel messages.

   The Port message type is used by clients which support a Distributed Hash Table (DHT) approach to finding peers rather than using a tracker. As I did not implement a client using DHT, I will say no more about it.

7. Writing to file

   One interesting question in writing a Bittorrent client is when to write data to file rather than holding it in memory. This is an implementation decision and can be done immediately upon receiving every block (minimal RAM usage), after hash-checking each piece, or after fully downloading the torrent file. This last option may slow down significantly if you have insufficient RAM to store the entire file in memory. At any rate, at some point in your Bittorrent download process, you will need to write data to a file.

   If the torrent consists of a single file, writing the data into a single file is easy and the name for the downloaded file will be given in the ‘name’ field of the .torrent metafile. If the torrent consists of multiple files, the .torrent metafile will contain a ‘files’ element within the info dictionary which contains information for each file. Each file will have a ‘path’ element which is a list representing the path and filename, with the filename being the last element of the list. Each file will also have a ‘length’ element that specifies the length of the file. Since the data for the torrent is run together in a continuous byte sequence, pieces can hold data for multiple files. The data will be written in the order of the files listed in the ‘files’ element. You will need the length field for each file in order to know how much data to write out to that file.

8. Algorithms and Extensions

> There is an interesting listing of potential algorithms to use in the unofficial specification here, and another section on official extensions to the protocol which you can optionally choose to support here. Both sections are interesting and worth a look.

You should now have a Bittorrent client that can download a torrent file. Stick with legal torrents, please, and happy torrenting!

This entry was posted in **Uncategorized** by **kristenwidman**. Bookmark the **permalink [http://www.kristenwidman.com/blog/71/how-to-write-a-bittorrent-client-part-2/]** .