BitTorrent.org

Home For Users For Developers Developer mailing list Forums (archive)

BEP:	3
Title:	The BitTorrent Protocol Specification
Version:	0e08ddf84d8d3bf101cdf897fc312f2774588c9e
Last-Modified:	Sat Feb 4 12:58:40 2017 +0100
Author:	Bram Cohen <bram@bittorrent.com>
Status:	Final
Type:	Standard
Created:	10-Jan-2008
Post-History:	24-Jun-2009 (arvid@bittorrent.com), clarified the encoding of strings in torrent files. 20-Oct-2012 (arvid@bittorrent.com), clarified that info-hash is the digest of en bencoding found in .torrent file. Introduced some references to new BEPs and cleaned up formatting. 11-Oct-2013 (arvid@bittorrent.com), correct the accepted and de-facto sizes for request messages 04-Feb-2017 (the8472.bep@infinite-source.de), further info-hash clarifications, added resources for new implementors

BitTorrent is a protocol for distributing files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over plain HTTP is that when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load.

A BitTorrent file distribution consists of these entities:

- An ordinary web server
- A static 'metainfo' file
- A BitTorrent tracker
- An 'original' downloader
- The end user web browsers
- The end user downloaders

There are ideally many end users for a single file.

To start serving, a host goes through the following steps:

1. Start running a tracker (or, more likely, have one running already).
2. Start running an ordinary web server, such as apache, or have one already.
3. Associate the extension .torrent with mimetype application/x-bittorrent on their web server (or have done so already).
4. Generate a metainfo (.torrent) file using the complete file to be served and the URL of the tracker.
5. Put the metainfo file on the web server.
6. Link to the metainfo (.torrent) file from some other web page.

7. Start a downloader which already has the complete file (the 'origin').

To start downloading, a user does the following:

1. Install BitTorrent (or have done so already).
2. Surf the web.
3. Click on a link to a .torrent file.
4. Select where to save the file locally, or select a partial download to resume.
5. Wait for download to complete.
6. Tell downloader to exit (it keeps uploading until this happens).

bencoding

- Strings are length-prefixed base ten followed by a colon and the string. For example `4:spam` corresponds to 'spam'.
- Integers are represented by an 'i' followed by the number in base 10 followed by an 'e'. For example `i3e` corresponds to 3 and `i-3e` corresponds to -3. Integers have no size limitation. `i-0e` is invalid. All encodings with a leading zero, such as `i03e`, are invalid, other than `i0e`, which of course corresponds to 0.
- Lists are encoded as an 'l' followed by their elements (also bencoded) followed by an 'e'. For example `l4:spam4:eggse` corresponds to ['spam', 'eggs'].
- Dictionaries are encoded as a 'd' followed by a list of alternating keys and their corresponding values followed by an 'e'. For example, `d3:cow3:moo4:spam4:eggse` corresponds to {'cow': 'moo', 'spam': 'eggs'} and `d4:spaml1:a1:bee` corresponds to {'spam': ['a', 'b']}. Keys must be strings and appear in sorted order (sorted as raw strings, not alphanumerics).

metainfo files

Metainfo files (also known as .torrent files) are bencoded dictionaries with the following keys:

announce

> The URL of the tracker.

info

> This maps to a dictionary, with keys described below.

All strings in a .torrent file that contains text must be UTF-8 encoded.

info dictionary

The `name` key maps to a UTF-8 encoded string which is the suggested name to save the file (or directory) as. It is purely advisory.

`piece length` maps to the number of bytes in each piece the file is split into. For the purposes of transfer, files are split into fixed-size pieces which are all the same length except for possibly the last one which may be truncated. `piece length` is almost always a power of two, most commonly 2 18 = 256 K (BitTorrent prior to version 3.2 uses 2 20 = 1 M as default).

`pieces` maps to a string whose length is a multiple of 20. It is to be subdivided into strings of length 20, each of which is the SHA1 hash of the piece at the corresponding index.

There is also a key `length` or a key `files`, but not both or neither. If `length` is present then the download represents a single file, otherwise it represents a set of files which go in a directory structure.

In the single file case, `length` maps to the length of the file in bytes.

For the purposes of the other keys, the multi-file case is treated as only having a single file by concatenating the files in the order they appear in the files list. The files list is the value `files` maps to, and is a list of dictionaries containing the following keys:

`length` - The length of the file, in bytes.

`path` - A list of UTF-8 encoded strings corresponding to subdirectory names, the last of which is the actual file name (a zero length list is an error case).

In the single file case, the name key is the name of a file, in the muliple file case, it's the name of a directory.

trackers

Tracker GET requests have the following keys:

info_hash

> The 20 byte sha1 hash of the bencoded form of the info value from the metainfo file. This value will almost certainly have to be escaped.
>
> Note that this is a substring of the metainfo file. The info-hash must be the hash of the encoded form as found in the .torrent file, which is identical to bdecoding the metainfo file, extracting the info dictionary and encoding it *if and only if* the bdecoder fully validated the input (e.g. key ordering, absence of leading zeros). Conversely that means clients must either reject invalid metainfo files or extract the substring directly. They must not perform a decode-encode roundtrip on invalid data.

peer_id

> A string of length 20 which this downloader uses as its id. Each downloader generates its own id at random at the start of a new download. This value will also almost certainly have to be escaped.

ip

> An optional parameter giving the IP (or dns name) which this peer is at. Generally used for the origin if it's on the same machine as the tracker.

port

> The port number this peer is listening on. Common behavior is for a downloader to try to listen on port 6881 and if that port is taken try 6882, then 6883, etc. and give up after 6889.

uploaded

> The total amount uploaded so far, encoded in base ten ascii.

downloaded

> The total amount downloaded so far, encoded in base ten ascii.

left

> The number of bytes this peer still has to download, encoded in base ten ascii. Note that this can't be computed from downloaded and the file length since it might be a resume, and there's a chance that some of the downloaded data failed an integrity check and had to be re-downloaded.

event

> This is an optional key which maps to `started`, `completed`, or `stopped` (or `empty`, which is the same as not being present). If not present, this is one of the announcements done at regular intervals. An announcement using `started` is sent when a download first begins, and one using `completed` is sent when the download is complete. No `completed` is sent if the file was complete when started. Downloaders send an announcement using `stopped` when they cease downloading.

Tracker responses are bencoded dictionaries. If a tracker response has a key `failure reason`, then that maps to a human readable string which explains why the query failed, and no other keys are required. Otherwise, it must have two keys: `interval`, which maps to the number of seconds the downloader should wait between regular rerequests, and `peers`. `peers` maps to a list of dictionaries corresponding to `peers`, each of which contains the keys `peer id`, `ip`, and `port`, which map to the peer's self-selected ID, IP address or dns name as a string, and port number, respectively. Note that downloaders may rerequest on nonscheduled times if an event happens or they need more peers.

More commonly is that trackers return a compact representation of the peer list, see BEP 23.

If you want to make any extensions to metainfo files or tracker queries, please coordinate with Bram Cohen to make sure that all extensions are done compatibly.

It is common to announce over a UDP tracker protocol as well.

peer protocol

BitTorrent's peer protocol operates over TCP or uTP.

Peer connections are symmetrical. Messages sent in both directions look the same, and data can flow in either direction.

The peer protocol refers to pieces of the file by index as described in the metainfo file, starting at zero. When a peer finishes downloading a piece and checks that the hash matches, it announces that it has that piece to all of its peers.

Connections contain two bits of state on either end: choked or not, and interested or not. Choking is a notification that no data will be sent until unchoking happens. The reasoning and common techniques behind choking are explained later in this document.

Data transfer takes place whenever one side is interested and the other side is not choking. Interest state must be kept up to date at all times - whenever a downloader doesn't have something they currently would ask a peer for in unchoked, they must express lack of interest, despite being choked. Implementing this properly is tricky, but makes it possible for downloaders to know which peers will start downloading immediately if unchoked.

Connections start out choked and not interested.

When data is being transferred, downloaders should keep several piece requests queued up at once in order to get good TCP performance (this is called 'pipelining'.) On the other side, requests which can't be written out to the TCP buffer immediately should be queued up in memory rather than kept in an application-level network buffer, so they can all be thrown out when a choke happens.

The peer wire protocol consists of a handshake followed by a never-ending stream of length-prefixed messages. The handshake starts with character ninteen (decimal) followed by the string 'BitTorrent protocol'. The leading character is a length prefix, put there in the hope that other new protocols may do the same and thus be trivially distinguishable from each other.

All later integers sent in the protocol are encoded as four bytes big-endian.

After the fixed headers come eight reserved bytes, which are all zero in all current implementations. If you wish to extend the protocol using these bytes, please coordinate with Bram Cohen to make sure all extensions are done compatibly.

Next comes the 20 byte sha1 hash of the bencoded form of the info value from the metainfo file. (This is the same value which is announced as `info_hash` to the tracker, only here it's raw instead of quoted here). If both sides don't send the same value, they sever the connection. The one possible exception is if a downloader wants to do multiple downloads over a single port, they may wait for incoming connections to give a download hash first, and respond with the same one if it's in their list.

After the download hash comes the 20-byte peer id which is reported in tracker requests and contained in peer lists in tracker responses. If the receiving side's peer id doesn't match the one the initiating side expects, it severs the connection.

That's it for handshaking, next comes an alternating stream of length prefixes and messages. Messages of length zero are keepalives, and ignored. Keepalives are generally sent once every two minutes, but note that timeouts can be done much more quickly when data is expected.

peer messages

All non-keepalive messages start with a single byte which gives their type.

The possible values are:

- 0 - choke
- 1 - unchoke
- 2 - interested
- 3 - not interested
- 4 - have
- 5 - bitfield
- 6 - request
- 7 - piece
- 8 - cancel

'choke', 'unchoke', 'interested', and 'not interested' have no payload.

'bitfield' is only ever sent as the first message. Its payload is a bitfield with each index that downloader has sent set to one and the rest set to zero. Downloaders which don't have anything yet may skip the 'bitfield' message. The first byte of the bitfield corresponds to indices 0 - 7 from high bit to low bit, respectively. The next one 8-15, etc. Spare bits at the end are set to zero.

The 'have' message's payload is a single number, the index which that downloader just completed and checked the hash of.

'request' messages contain an index, begin, and length. The last two are byte offsets. Length is generally a power of two unless it gets truncated by the end of the file. All current implementations use 2^14 (16 kiB), and close connections which request an amount greater than that.

'cancel' messages have the same payload as request messages. They are generally only sent towards the end of a download, during what's called 'endgame mode'. When a download is almost complete, there's a tendency for the last few pieces to all be downloaded off a single hosed modem line, taking a very long time. To make sure the last few pieces come in quickly, once requests for all pieces a given downloader doesn't have yet are currently pending, it sends requests for everything to everyone it's downloading from. To keep this from becoming horribly inefficient, it sends cancels to everyone else every time a piece arrives.

'piece' messages contain an index, begin, and piece. Note that they are correlated with request messages implicitly. It's possible for an unexpected piece to arrive if choke and unchoke messages are sent in quick succession and/or transfer is going very slowly.

Downloaders generally download pieces in random order, which does a reasonably good job of keeping them from having a strict subset or superset of the pieces of any of their peers.

Choking is done for several reasons. TCP congestion control behaves very poorly when sending over many connections at once. Also, choking lets each peer use a tit-for-tat-ish algorithm to ensure that they get a consistent download rate.

The choking algorithm described below is the currently deployed one. It is very important that all new algorithms work well both in a network consisting entirely of themselves and in a network consisting mostly of this one.

There are several criteria a good choking algorithm should meet. It should cap the number of simultaneous uploads for good TCP performance. It should avoid choking and unchoking quickly, known as 'fibrillation'. It should reciprocate to peers who let it download. Finally, it should try out unused connections once in a while to find out if they might be better than the currently used ones, known as optimistic unchoking.

The currently deployed choking algorithm avoids fibrillation by only changing who's choked once every ten seconds. It does reciprocation and number of uploads capping by unchoking the four peers which it has the best download rates from and are interested. Peers which have a better upload rate but aren't interested get unchoked and if they become interested the worst uploader gets choked. If a downloader has a complete file, it uses its upload rate rather than its download rate to decide who to unchoke.

For optimistic unchoking, at any one time there is a single peer which is unchoked regardless of its upload rate (if interested, it counts as one of the four allowed downloaders.) Which peer is optimistically unchoked rotates every 30 seconds. To give them a decent chance of getting a complete piece to upload, new connections are three times as likely to start as the current optimistic unchoke as anywhere else in the rotation.

Resources

- The BitTorrent Economics Paper outlines some request and choking algorithms clients should implement for optimal performance
- When developing a new implementation the Wireshark protocol analyzer and its dissectors for bittorrent can be useful to debug and compare with existing ones.

Copyright