BEP:	52
Title:	The BitTorrent Protocol Specification v2
Version:	7392d83f374b071ca9601728c27be2d05e199e0c
Last-Modified:	Thu Aug 31 15:47:00 2017 -0700
Author:	Bram Cohen <bram@bittorrent.com>
Status:	Draft
Type:	Standards Track
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 14-May-2017 (ssiloti@bittorrent.com, the8472.bep@infinite-source.de), v2 wire protocol and torrent format

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).

Note that in the context of bencoding strings including dictionary keys are arbitrary byte sequences (`uint8_t[]`).

BEP authors are encouraged to use ASCII-compatible strings for dictionary keys and UTF-8 for human-readable data. Implementations must not rely on this.

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.

`piece layers`

> A dictionary of strings. For each file in the file tree that is larger than the piece size it contains one string value. The keys are the merkle roots while the values consist of concatenated hashes of one layer within that merkle tree. The layer is chosen so that one hash covers `piece length` bytes. For example if the piece size is 16KiB then the leaf hashes are used. If a piece size of 128KiB is used then 3rd layer up from the leaf hashes is used. Layer hashes which exclusively cover data beyond the end of file, i.e. are only needed to balance the tree, are omitted. All hashes are stored in their binary format.
>
> A torrent is not valid if this field is absent, the contained hashes do not match the merkle roots or are not from the correct layer.

All strings in a .torrent file defined by this BEP that contain human-readable text are UTF-8 encoded.

An example torrent creator implementation can be found here.

info dictionary

`name`

> A display name for the torrent. It is purely advisory.

`piece length`

The number of bytes that each logical piece in the peer protocol refers to. I.e. it sets the granularity of `piece`, `request`, `bitfield` and `have` messages. It must be a power of two and at least 16KiB.

Files are mapped into this piece address space so that each non-empty file is aligned to a piece boundary and occurs in the same order as in the file tree. The last piece of each file may be shorter than the specified piece length, resulting in an alignment gap.

`meta version`

An integer value, set to 2 to indicate compatibility with the current revision of this specification. Version 1 is not assigned to avoid confusion with BEP3. Future revisions will only increment this value to indicate an incompatible change has been made, for example that hash algorithms were changed due to newly discovered vulnerabilities. Implementations must check this field first and indicate that a torrent is of a newer version than they can handle before performing other validations which may result in more general messages about invalid files.

`file tree`

A tree of dictionaries where dictionary keys represent UTF-8 encoded path elements. Entries with zero-length keys describe the properties of the composed path at that point. 'UTF-8 encoded' in this context only means that if the native encoding is known at creation time it must be converted to UTF-8. Keys may contain invalid UTF-8 sequences or characters and names that are reserved on specific filesystems. Implementations must be prepared to sanitize them. On most platforms path components exactly matching '.' and '..' must be sanitized since they could lead to directory traversal attacks and conflicting path descriptions. On platforms that require valid UTF-8 path components this sanitizing step must happen after normalizing overlong UTF-8 encodings.

The `file tree` root dictionary itself must not be a file, i.e. it must not contain a zero-length key with a dictionary containing a `length` key.

File tree layout

Example:

```
{
  info: {
    file tree: {
      dir1: {
        dir2: {
          fileA.txt: {
            "": {
              length: <length of file in bytes (integer)>,
              pieces root: <optional, merkle tree root (string)>,
              ...
            }
          },
          fileB.txt: {
            "": {
              ...
            }
          }
        },
        dir3: {
          ...
        }
      }
    }
  }
}
```

Bencoded for fileA only:

```
d4:infod9:file treed4:dir1d4:dir2d9:fileA.txtd0:d5:lengthi1024e11:pieces root32:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaeeeee
```

`length`

Length of the file in bytes. Presence of this field indicates that the dictionary describes a file, not a directory. Which means it must not have any sibling entries.

`pieces root`

For non-empty files this is the the root hash of a merkle tree with a branching factor of 2, constructed from 16KiB blocks of the file. The last block may be shorter than 16KiB. The remaining leaf hashes beyond the end of the file required to construct upper layers of the merkle tree are set to zero. As of `meta version` 2 SHA2-256 is used as digest function for the merkle tree. The hash is stored in its binary form, not as human-readable string.

Note that identical files always result in the same root hash.

Interpreting paths:

`file tree: {name.ext: {"": {length: ...}}}`

a single-file torrent

`file tree: {nameA.ext: {"": {length: ...}}, nameB.ext: {"": {length: ...}}, dir: {...}}`

a rootless multifile torrent, i.e. a list of files and directories without a named common directory containing them. implementations may offer users to optionally prepend the torrent name as root to avoid file name collisions.

`file tree: {dir: {nameA.ext: {"": {length: ...}}, nameB.ext: {"": {length: ...}}}}`

multiple files rooted in a single directory

infohash

The infohash is calculated by applying a hash function to the bencoded form of the info dictionary, which is a substring of the metainfo file. For `meta version` 2 SHA2-256 is used.

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 implementations must either reject invalid metainfo files or extract the substring directly. They must not perform a decode-encode roundtrip on invalid data.

For some uses as torrent identifier it is truncated to 20 bytes.

When verifying an infohash implementations must also check that the `piece layers` hashes outside the info dictionary match the `pieces root` fields.

trackers

Tracker GET requests have the following keys:

info_hash

The 20byte truncated infohash as described above. This value will almost certainly have to be escaped.

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 and BEP 7.

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 nineteen (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 truncated infohash. 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
- 16 - reject
- 21 - hash request
- 22 - hashes
- 23 - hash reject

'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.

'hash request' messages contain a pieces root, base layer, index, length, and proof layers. The pieces root is the root hash of a file. The base layer defines the lowest requested layer of the hash tree. It is the number of layers above the leaf layer that the hash list should start at. A value of zero indicates that leaf hashes are requested. Clients are only required to support setting the base layer to the leaf and piece layers. Index is the offset in hashes of the first requested hash in the base layer. Index MUST be a multiple of length, this includes zero. Length is the number of hashes to include from the base layer. Length MUST be equal-to-or-greater-than two and a power of two. Length SHOULD NOT be greater than 512. Proof layers is the number of ancestor layers to include. Note that the limits imposed on index and length above mean that at-most one uncle hash is needed from each proof layer. Hash requests MUST be answered with either a 'hashes' or 'hash reject' message.

Clients must be able to service requests for hash blocks covering pieces which they have announced through bitfield or have messages. They may be able to service additional requests if they have access to the full layers from a metadata file but requesting implementations should try to prioritize requests where they can be certain that the other party can must have the necessary data.

Hash requests may be sent to a peer regardless of its choke state. For unchoked peers, hash requests should be subject to the same rate limiting policy as piece requests, except that all hash requests are not necessarily rejected after a peer is choked. Clients may impose a separate rate limit on hash requests received from choked peers. Clients MUST NOT reject a hash request with a base layer of zero if it immediately follows a request for one of the chunks in the requested range and the client services the chunk request with a piece message.

'hashes' messages contain a pieces root, base layer, index, length, proof layers, and hashes. This message MUST correlate with a 'hash request' message. Hashes starts with the base layer and ends with the uncle hash closest to the root. A proof layer is omitted if the requested hashes include the entire child layer. In other words, the first log2(length)-1 proof layers are ommitted. The ommitted layers are still counted towards the requested proof layers.

'hash reject' messages have the same payload as 'hash request' messages. They indicate that a peer will not service a hash request.

'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 a 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. cancel messages do not relieve the other side from the duty of responding to a request. They must either send a piece or a reject message as response.

'reject' messages have the same payload as request messages. They indicate that a peer will not service a request. They must be sent after a choke message to cancel all pending requests.

'piece' messages contain an index, begin, and piece. Note that they are correlated with request messages must be explicitly rejected by the remote after an unchoke. This means a request is answered with either a piece or reject messages. If an unsolicited piece is received a peer MUST close the connection.

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.

Note that the original version of the peer protocol had no reject message. That message has been adopted from the Fast Extension which specifies further optional messages.

Upgrade Path

For interoperability with BEP 3 a torrent can be created to contain the necessary data for both formats. To do so the 'pieces' field and 'files' or 'length' in the info dictionary must be generated to describe the same data in the same order. Since the old format did not align files to piece boundaries a multifile torrent must use BEP 47 padding files to achieve identical alignment.

Implementations supporting both formats can join both swarms by calculating the new and old infohashes and downloading them to the same storage. Before doing so they must validate that the content (file names, order, piece alignment) is identical. During the download they must also verify that pieces match both piece hash formats. If any inconsistency is detected they may either abort or fall back to downloading one of the two formats as if the other were not present.

When initiating a connection and sending the sha1 infohash of such a hybrid torrent a peer can set the 4th most significant bit in the last byte of the reserved bitfield to indicate that it also supports the new format. The remote peer may then respond with the new infohash to upgrade the connect to the new format.

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