BitTorrentSpecification

From TheoryOrg

Bittorrent Protocol Specification v1.0

Contents

Identification

BitTorrent is a peer-to-peer file sharing protocol designed by Bram Cohen. Visit his pages at http://www.bittorrent.com BitTorrent is designed to facilitate file transfers among multiple peers across unreliable networks.

Purpose

The purpose of this specification is to document version 1.0 of the BitTorrent protocol specification in detail. Bram's protocol specification page (http://bittorrent.org/beps/bep_0003.html) outlines the protocol in somewhat general terms, and lacks behaviorial detail in some areas. The hope is that this document will become a **formal** specification, written in clear, unambiguous terms, which can be used as a basis for discussion and implementation in the future.

This document is intended to be maintained and used by the BitTorrent development community. Everyone is invited to contribute to this document, with the understanding that the content here is intended to represent the current protocol, which is already deployed in a number of existing client implementations.

This is not the place to suggest feature requests. For that, please go to the mailing list (http://lists.ibiblio.org/mailman/listinfo/bittorrent) .

Scope

This document applies to the first version (i.e. version 1.0) of the BitTorrent protocol specification. Currently, this applies to the torrent file structure, peer wire protocol, and the Tracker HTTP/HTTPS protocol specifications. As newer revisions of each protocol are defined, they should be specified on their own separate pages, **not here**.

Related Documents

- Official protocol specification (http://bittorrent.org/beps/bep_0003.html)
- Developer and user wishlist
- Tracker protocol extensions

Conventions

In this document, a number of conventions are used in an attempt to present information in a concise and unambiguous fashion.

- *peer* v/s *client*: In this document, a *peer* is any BitTorrent client participating in a download. The *client* is also a peer, however it is the BitTorrent client that is running on the local machine. Readers of this specification may choose to think of themselves as the *client* which connects to numerous *peers*.
- *piece* v/s *block*: In this document, a *piece* refers to a portion of the downloaded data that is described in the metainfo file, which can be verified by a SHA1 hash. A *block* is a portion of data that a *client* may request from a *peer*. Two or more *blocks* make up a whole *piece*, which may then be verified.
- *defacto standard*: Large blocks of text in *italics* indicates a practice so common in various client implementations of BitTorrent that it is considered a defacto standard.

In order to help others find recent changes that have been made to this document, please fill out the change log (last section). This should contain a brief (i.e. one-line) entry for each major change that you've made to the document.

bencoding

Bencoding is a way to specify and organize data in a terse format. It supports the following types: byte strings, integers, lists, and dictionaries.

byte strings

Byte strings are encoded as follows: *<string length encoded in base ten ASCII>:<string data>*
Note that there is no constant beginning delimiter, and no ending delimiter.

Example: *4:spam* represents the string "spam"

integers

Integers are encoded as follows: *i<integer encoded in base ten ASCII>e*
The initial **i** and trailing **e** are beginning and ending delimiters. You can have negative numbers such as *i-3e*. You cannot prefix the number with a zero such as *i04e*. However, *i0e* is valid.

Example: *i3e* represents the integer "3"

- *NOTE:* The maximum number of bit of this integer is unspecified, but to handle it as a signed 64bit integer is mandatory to handle "large files" aka .torrent for more that 4Gbyte

lists

Lists are encoded as follows: *l<bencoded values>e*
The initial **l** and trailing **e** are beginning and ending delimiters. Lists may contain any bencoded type, including integers, strings, dictionaries, and other lists.

Example: *l4:spam4:eggse* represents the list of two strings: ["spam", "eggs"]

dictionaries

Dictionaries are encoded as follows: *d<bencoded string><bencoded element>e*
The initial **d** and trailing **e** are the beginning and ending delimiters. Note that the keys must be bencoded strings. The values may be any bencoded type, including integers, strings, lists, and other dictionaries. Keys must be strings and appear in sorted order (sorted as raw strings, not alphanumerics). The strings should be compared using a binary comparison, not a culture-specific "natural" comparison.

Example: *d3:cow3:moo4:spam4:eggse* represents the dictionary { "cow" => "moo", "spam" => "eggs" }
Example: *d4:spaml1:a1:bee* represents the dictionary { "spam" => ["a", "b"] }
Example: *d9:publisher3:bob17:publisher-webpage15:www.example.com18:publisher.location4:homee* represents { "publisher" => "bob", "publisher-webpage" => "www.example.com", "publisher.location" => "home" }

Implementations

- C (http://funzix.svn.sourceforge.net/viewvc/funzix/trunk/bencode/bencode.c?view=markup)
- Perl (http://search.cpan.org/dist/Convert-Bencode/lib/Convert/Bencode.pm)
- Java (http://www.koders.com/java/fid47111A56F2466C232E09AEF75A39915EC70D3536.aspx#L52)
- Python by Hackeron
- Decoding encoding bencoded data with haskell by Edi
- Objective-C (http://www.stupendous.net/projects/bencoding-obj-c-class/) by Chrome

Metainfo File Structure

All data in a metainfo file is bencoded. The specification for bencoding is defined above.

The content of a metainfo file (the file ending in ".torrent") is a bencoded dictionary, containing the keys listed below. All character string values are UTF-8 encoded.

- **info**: a dictionary that describes the file(s) of the torrent. There are two possible forms: one for the case of a 'single-file' torrent with no directory structure, and one for the case of a 'multi-file' torrent (see below for details)
- **announce**: The announce URL of the tracker (string)
- **announce-list**: (optional) this is an extention to the official specification, offering backwards-compatibility. (list of lists of strings).
  - The official request for a specification change is here (http://bittorrent.org/beps/bep_0012.html) .
- **creation date**: (optional) the creation time of the torrent, in standard UNIX epoch format (integer, seconds since 1-Jan-1970 00:00:00 UTC)
- **comment**: (optional) free-form textual comments of the author (string)
- **created by**: (optional) name and version of the program used to create the .torrent (string)
- **encoding**: (optional) the string encoding format used to generate the **pieces** part of the **info** dictionary in the .torrent metafile (string)

Info Dictionary

This section contains the field which are common to both mode, "single file" and "multiple file".

- **piece length**: number of bytes in each piece (integer)

- **pieces**: string consisting of the concatenation of all 20-byte SHA1 hash values, one per piece (byte string, i.e. not urlencoded)
- **private**: (optional) this field is an integer. If it is set to "1", the client MUST publish its presence to get other peers ONLY via the trackers explicitly described in the metainfo file. If this field is set to "0" or is not present, the client may obtain peer from other means, e.g. PEX peer exchange, dht. Here, "private" may be read as "no external peer source".
  - **NOTE:** There is much debate surrounding private trackers.
  - The official request for a specification change is here (http://bittorrent.org/beps/bep_0027.html) .
  - Azureus was the first client to respect private trackers, see their wiki (http://www.azureuswiki.com/index.php /Secure_Torrents) for more details.

Info in Single File Mode

For the case of the **single-file** mode, the **info** dictionary contains the following structure:

- **name**: the filename. This is purely advisory. (string)
- **length**: length of the file in bytes (integer)
- **md5sum**: (optional) a 32-character hexadecimal string corresponding to the MD5 sum of the file. This is not used by BitTorrent at all, but it is included by some programs for greater compatibility.

Info in Multiple File Mode

For the case of the **multi-file** mode, the **info** dictionary contains the following structure:

- **name**: the filename of the directory in which to store all the files. This is purely advisory. (string)
- **files**: a list of dictionaries, one for each file. Each dictionary in this list contains the following keys:
  - **length**: length of the file in bytes (integer)
  - **md5sum**: (optional) a 32-character hexadecimal string corresponding to the MD5 sum of the file. This is not used by BitTorrent at all, but it is included by some programs for greater compatibility.
  - **path**: a list containing one or more string elements that together represent the path and filename. Each element in the list corresponds to either a directory name or (in the case of the final element) the filename. For example, a the file "dir1/dir2 /file.ext" would consist of three string elements: "dir1", "dir2", and "file.ext". This is encoded as a bencoded list of strings such as *l**4**:dir1**4**:dir2**8**:file.exte*

Notes

- The **piece length** specifies the nominal piece size, and is usually a power of 2. The piece size is typically chosen based on the total amount of file data in the torrent, and is constrained by the fact that too-large piece sizes cause inefficiency, and too-small piece sizes cause large .torrent metadata file. Historically, piece size was chosen to result in a .torrent file no greater than approx. 50 - 75 kB (presumably to ease the load on the server hosting the torrent files).
  - Current best-practice is to *keep the piece size to 512KB or less,* for torrents around 8-10GB, even if that results in a larger .torrent file. This results in a more efficient swarm for sharing files. The most common sizes are 256 kB, 512 kB, and 1 MB.
  - Every piece is of equal length except for the final piece, which is irregular. The number of pieces is thus determined by 'ceil(total length / piece size)'.
  - For the purposes of piece boundaries in the multi-file case, consider the file data as one long continuous stream, composed of the concatenation of each file in the order listed in the *files* list. The number of pieces and their boundaries are then determined in the same manner as the case of a single file. Pieces may overlap file boundaries.
- Each piece has a corresponding SHA1 hash of the data contained within that piece. These hashes are concatenated to form the *pieces **value in the above*** **info** dictionary. Note that this is **not** a list but rather a single string. The length of the string must be a multiple of 20.

Tracker HTTP/HTTPS Protocol

The tracker is an HTTP/HTTPS service which responds to HTTP GET requests. The requests include metrics from clients that help the tracker keep overall statistics about the torrent. The response includes a peer list that helps the client participate in the torrent. The base URL consists of the "announce URL" as defined in the metadata (.torrent) file. The parameters are then added to this URL, using standard CGI methods (i.e. a '?' after the announce URL, followed by 'param=value' sequences separated by '&').

Note that all binary data in the URL (particularly info_hash and peer_id) must be properly escaped. This means any byte not in the set 0-9, a-z, A-Z, '.', '-', '_' and '~', must be encoded using the "%nn" format, where nn is the hexadecimal value of the byte. (See RFC1738 (http://www.faqs.org/rfcs/rfc1738.html) for details.)

For a 20-byte hash of \x12\x34\x56\x78\x9a\xbc\xde\xf1\x23\x45\x67\x89\xab\xcd\xef\x12\x34\x56\x78\x9a,
The right encoded form is %124Vx%9A%BC%DE%F1%23Eg%89%AB%CD%EF%124Vx%9A

Tracker Request Parameters

The parameters used in the client->tracker GET request are as follows:

- **info_hash**: urlencoded 20-byte SHA1 hash of the *value* of the *info* key from the Metainfo file. Note that the *value* will be a bencoded dictionary, given the definition of the *info* key above.
- **peer_id**: urlencoded 20-byte string used as a unique ID for the client, generated by the client at startup. This is allowed to be any value, and may be binary data. *There are currently no guidelines for generating this peer ID. However, one may rightly presume that it must at least be unique for your local machine, thus should probably incorporate things like process ID and perhaps a timestamp recorded at startup. See peer_id below for common client encodings of this field.*
- **port**: The port number that the client is listening on. Ports reserved for BitTorrent are typically 6881-6889. Clients may choose to give up if it cannot establish a port within this range.
- **uploaded**: The total amount uploaded (since the client sent the 'started' event to the tracker) in base ten ASCII. While not explicitly stated in the official specification, the concensus is that this should be the total number of bytes uploaded.
- **downloaded**: The total amount downloaded (since the client sent the 'started' event to the tracker) in base ten ASCII. While not explicitly stated in the official specification, the consensus is that this should be the total number of bytes downloaded.
- **left**: The number of bytes this client still has to download, encoded in base ten ASCII.
- **compact**: Setting this to 1 indicates that the client accepts a compact response. The peers list is replaced by a peers string with 6 bytes per peer. The first four bytes are the host (in network byte order), the last two bytes are the port (again in network byte order). It should be noted that some trackers only support compact responses (for saving bandwidth) and either refuse requests without "compact=1" or simply send a compact response unless the request contains "compact=0" (in which case they will refuse the request.)
- **no_peer_id**: Indicates that the tracker can omit peer id field in peers dictionary. This option is ignored if compact is enabled.
- **event**: If specified, must be one of *started*, *completed*, *stopped*, (or empty which is the same as not being specified). If not specified, then this request is one performed at regular intervals.
  - **started**: The first request to the tracker *must* include the event key with this value.
  - **stopped**: Must be sent to the tracker if the client is shutting down gracefully.
  - **completed**: Must be sent to the tracker when the download completes. However, must not be sent if the download was already 100% complete when the client started. Presumably, this is to allow the tracker to increment the "completed downloads" metric based soley on this event.
- **ip**: Optional. The true IP address of the client machine, in dotted quad format or rfc3513 defined hexed IPv6 address. *Notes: In general this parameter is not necessary as the address of the client can be determined from the IP address from which the HTTP request came. The parameter is only needed in the case where the IP address that the request came in on is not the IP address of the client. This happens if the client is communicating to the tracker through a proxy (or a transparent web proxy/cache.) It also is necessary when both the client and the tracker are on the same local side of a NAT gateway. The reason for this is that otherwise the tracker would give out the internal (RFC1918) address of the client, which is not routeable. Therefore the client must explicitly state its (external, routeable) IP address to be given out to external peers. Various trackers treat this parameter differently. Some only honor it only if the IP address that the request came in on is in RFC1918 space. Others honor it unconditionally, while others ignore it completely. In case of IPv6 address (e.g.: 2001:db8:1:2::100) it indicates only that client can communicate via IPv6.*
- **numwant**: Optional. Number of peers that the client would like to receive from the tracker. This value is permitted to be zero. If omitted, typically defaults to 50 peers.
- **key**: Optional. An additional identification that is not shared with any users. It is intended to allow a client to prove their identity should their IP address change.
- **trackerid**: Optional. If a previous announce contained a tracker id, it should be set here.

Tracker Response

The tracker responds with "text/plain" document consisting of a bencoded dictionary with the following keys:

- **failure reason**: If present, then no other keys may be present. The value is a human-readable error message as to why the request failed (string).
- **warning message**: (new, optional) Similar to failure reason, but the response still gets processed normally. The warning message is shown just like an error.
- **interval**: Interval in seconds that the client should wait between sending regular requests to the tracker
- **min interval**: (optional) Minimum announce interval. If present clients must not reannounce more frequently than this.
- **tracker id**: A string that the client should send back on its next announcements. If absent and a previous announce sent a tracker id, do not discard the old value; keep using it.
- **complete**: number of peers with the entire file, i.e. seeders (integer)
- **incomplete**: number of non-seeder peers, aka "leechers" (integer)
- **peers**: (dictionary model) The value is a list of dictionaries, each with the following keys:
  - **peer id**: peer's self-selected ID, as described above for the tracker request (string)
  - **ip**: peer's IP address either IPv6 (hexed) or IPv4 (dotted quad) or DNS name (string)
  - **port**: peer's port number (integer)
- **peers**: (binary model) Instead of using the dictionary model described above, the **peers** value may be a string consisting of multiples of 6 bytes. First 4 bytes are the IP address and last 2 bytes are the port number. All in network (big endian) notation.

As mentioned above, the list of peers is length 50 by default. If there are fewer peers in the torrent, then the list will be smaller. Otherwise, the tracker randomly selects peers to include in the response. *The tracker may choose to implement a more intelligent mechanism for peer selection when responding to a request. For instance, reporting seeds to other seeders could be avoided.*

Clients may send a request to the tracker more often than the specified interval, if an event occurs (i.e. stopped or completed) or if the client needs to learn about more peers. However, it is considered bad practice to "hammer" on a tracker to get multiple peers. If a client wants a large peer list in the response, then it should specify the **numwant** parameter.

***Implementer's Note**: Even 30 peers is **plenty**, the official client version 3 in fact only actively forms new connections if it has less than 30 peers and will refuse connections if it has 55. **This value is important to performance**. When a new piece has completed download, HAVE messages (see below) will need to be sent to most active peers. As a result the cost of broadcast traffic grows in direct proportion to the number of peers. Above 25, new peers are highly unlikely to increase download speed. UI designers are **strongly** advised to make this obscure and hard to change as it is very rare to be useful to do so.*

Tracker 'scrape' Convention

By convention most trackers support another form of request, which queries the state of a given torrent (or all torrents) that the tracker is managing. This is referred to as the "scrape page" because it automates the otherwise tedious process of "screen scraping" the tracker's stats page.

The scrape URL is also a HTTP GET method, similar to the one described above. However the base URL is different. To derive the scrape URL use the following steps: Begin with the announce URL. Find the last '/' in it. If the text immediately following that '/' isn't 'announce' it will be taken as a sign that that tracker doesn't support the scrape convention. If it does, substitute 'scrape' for 'announce' to find the scrape page.

Examples: (announce URL -> scrape URL)

```
~http://example.com/announce          -> ~http://example.com/scrape
~http://example.com/x/announce        -> ~http://example.com/x/scrape
~http://example.com/announce.php      -> ~http://example.com/scrape.php
~http://example.com/a                 -> (scrape not supported)
~http://example.com/announce?x2%0644 -> ~http://example.com/scrape?x2%0644
~http://example.com/announce?x=2/4    -> (scrape not supported)
~http://example.com/x%064announce     -> (scrape not supported)
```

Note especially that entity unquoting is *not* to be done. This standard is documented by Bram in the BitTorrent development list archive: http://groups.yahoo.com/group/BitTorrent/message/3275

The scrape URL may be supplemented by the optional parameter *info_hash*, a 20-byte value as described above. This restricts the tracker's report to that particular torrent. Otherwise stats for all torrents that the tracker is managing are returned. Software authors are strongly encouraged to use the *info_hash* parameter when at all possible, to reduce the load and bandwidth of the tracker.

You may also specify multiple info_hash parameters to trackers that support it. While this isn't part of the official specifications it has become somewhat a defacto standard - for example:

```
http://example.com/scrape.php?info_hash=aaaaaaaaaaaaaaaaaaaa&info_hash=bbbbbbbbbbbbbbbbbbbb&info_hash=cccccccccccccccccccc
```

The response of this HTTP GET method is a "text/plain" or sometimes gzip compressed document consisting of a bencoded dictionary, containing the following keys:

- **files**: a dictionary containing one key/value pair for each torrent for which there are stats. If *info_hash* was supplied and was valid, this dictionary will contain a single key/value. Each key consists of a 20-byte binary *info_hash*. The value of each entry is another dictionary containing the following:
  - **complete**: number of peers with the entire file, i.e. seeders (integer)
  - **downloaded**: total number of times the tracker has registered a completion ("event=complete", i.e. a client finished downloading the torrent)
  - **incomplete**: number of non-seeder peers, aka "leechers" (integer)
  - **name**: (optional) the torrent's internal name, as specified by the "name" file in the info section of the .torrent file

Note that this response has three levels of dictionary nesting. Here's an example:

d5:*files*d20:....................d8:*complete*i**5**e10:*downloaded*i**50**e10:*incomplete*i**10**eeee

Where .... is the 20 byte info_hash and there are 5 seeders, 10 leechers, and 50 complete downloads.

Unofficial extensions to scrape

Below are the response keys are being unofficially used. Since they are unofficial, they are all optional.

- **failure reason**: Human-readable error message as to why the request failed (string). Clients known to handle this key: Azureus.
- **flags**: a dictionary containing miscellaneous flags. The value of the flags key is another nested dictionary, possibly containing the following:

- **min_request_interval**: The value for this key is an integer specifying how the minimum number of seconds for the client to wait before scraping the tracker again. Trackers known to send this key: BNBT. Clients known to handle this key: Azureus.

Peer wire protocol (TCP)

Overview

The peer protocol facilitates the exchange of pieces as described in the '*metainfo* file.

Note here that the original specification also used the term "piece" when describing the peer protocol, but as a different term than "piece" in the metainfo file. For that reason, the term "block" will be used in this specification to describe the data that is exchanged between peers over the wire.

A client must maintain state information for each connection that it has with a remote peer:

- **choked**: Whether or not the remote peer has choked this client. When a peer chokes the client, it is a notification that no requests will be answered until the client is unchoked. The client should not attempt to send requests for blocks, and it should consider all pending (unanswered) requests to be discarded by the remote peer.
- **interested**: Whether or not the remote peer is interested in something this client has to offer. This is a notification that the remote peer will begin requesting blocks when the client unchokes them.

Note that this also implies that the client will also need to keep track of whether or not it is interested in the remote peer, and if it has the remote peer choked or unchoked. So, the real list looks something like this:

- **am_choking**: this client is choking the peer
- **am_interested**: this client is interested in the peer
- **peer_choking**: peer is choking this client
- **peer_interested**: peer is interested in this client

Client connections start out as "choked" and "not interested". In other words:

- **am_choking** = 1
- **am_interested** = 0
- **peer_choking** = 1
- **peer_interested** = 0

A block is downloaded by the client when the client is interested in a peer, and that peer is not choking the client. A block is uploaded by a client when the client is not choking a peer, and that peer is interested in the client.

It is important for the client to keep its peers informed as to whether or not it is interested in them. This state information should be kept up-to-date with each peer even when the client is choked. This will allow peers to know if the client will begin downloading when it is unchoked (and vice-versa).

Data Types

Unless specified otherwise, all integers in the peer wire protocol are encoded as four byte big-endian values. This includes the length prefix on all messages that come after the handshake.

Message flow

The peer wire protocol consists of an initial handshake. After that, peers communicate via an exchange of length-prefixed messages. The length-prefix is an integer as described above.

Handshake

The handshake is a required message and must be the first message transmitted by the client. It is (49+len(pstr)) bytes long.

handshake: <pstrlen><pstr><reserved><info_hash><peer_id>

- **pstrlen**: string length of <pstr>, as a single raw byte
- **pstr**: string identifier of the protocol
- **reserved**: eight (8) reserved bytes. All current implementations use all zeroes. Each bit in these bytes can be used to change the behavior of the protocol. *An email from Bram suggests that trailing bits should be used first, so that leading bits may be used to change the meaning of trailing bits.*
- **info_hash**: 20-byte SHA1 hash of the info key in the metainfo file. This is the same info_hash that is transmitted in tracker requests.

- **peer_id**: 20-byte string used as a unique ID for the client. This is usually the same peer_id that is transmitted in tracker requests (but not always e.g. an anonymity option in Azureus).

In version 1.0 of the BitTorrent protocol, pstrlen = 19, and pstr = "BitTorrent protocol".

The initiator of a connection is expected to transmit their handshake immediately. The recipient may wait for the initiator's handshake, if it is capable of serving multiple torrents simultaneously (torrents are uniquely identified by their info*hash). However, the recipient must respond as soon as it sees the info_hash part of the handshake. The tracker's NAT-checking feature does not send the peer_id field of the handshake*.

If a client receives a handshake with an info_hash that it is not currently serving, then the client must drop the connection.

If the initiator of the connection receives a handshake in which the peer_id does not match the expected peer*id, then the initiator is expected to drop the connection*. Note that the initiator presumably received the peer information from the tracker, which includes the peer_id that was registered by the peer. The peer_id from the tracker and in the handshake are expected to match.

peer_id

The peer_id is exactly 20 bytes (characters) long.

There are mainly two conventions how to encode client and client version information into the peer_id, Azureus-style and Shadow's-style.

Azureus-style uses the following encoding: '-', two characters for client id, four ascii digits for version number, '-', followed by random numbers.

For example: '-AZ2060-'...

known clients that uses this encoding style are:

- 'AG' - Ares (http://aresgalaxy.sourceforge.net/)
- 'A~' - Ares (http://aresgalaxy.sourceforge.net/)
- 'AR' - Arctic (http://dev.int64.org/arctic.html)
- 'AT' - Artemis (http://www.cyberartemis.com)
- 'AX' - BitPump (http://www.analogx.com/contents/download/network/bitpump.htm)
- 'AZ' - Azureus (http://azureus.sf.net)
- 'BB' - BitBuddy (http://www.btvampire.com)
- 'BC' - BitComet (http://www.bitcomet.com)
- 'BF' - Bitflu (http://bitflu.workaround.ch/)
- 'BG' - BTG (uses Rasterbar libtorrent) (http://btg.berlios.de/)
- 'BP' - BitTorrent Pro (Azureus + spyware) (http://www.intelpeers.com)
- 'BR' - BitRocket (http://www.bitrocket.org)
- 'BS' - BTSlave (http://btslave.sourceforge.net)
- 'BW' - BitWombat (http://bitwombat.com)
- 'BX' - ~Bittorrent X
- 'CD' - Enhanced CTorrent (http://www.rahul.net/dholmes/ctorrent/)
- 'CT' - CTorrent (http://ctorrent.sourceforge.net)
- 'DE' - DelugeTorrent (http://www.deluge-torrent.org)
- 'DP' - Propagate Data Client (http://www.propagatedata.com/)
- 'EB' - EBit (http://dywt.com.cn/)
- 'ES' - electric sheep (http://electricsheep.org)
- 'FC' - FileCroc (http://www.filecroc.com)
- 'FT' - FoxTorrent (http://www.foxtorrent.com)
- 'GS' - GSTorrent (http://sourceforge.net/projects/gstorrent)
- 'HL' - Halite (http://www.binarynotions.com/halite.php)
- 'HN' - Hydranode (http://hydranode.com)
- 'KG' - KGet (http://kget.sourceforge.net)
- 'KT' - KTorrent (http://ktorrent.org)
- 'LC' - LeechCraft (http://deviant-soft.ws/tiki-index.php?page=LeechCraft)
- 'LH' - LH-ABC (http://code.google.com/p/lh-abc)
- 'LP' - Lphant (http://www.lphant.com)
- 'LT' - libtorrent (http://libtorrent.sf.net)
- 'lt' - libTorrent (http://libtorrent.rakshasa.no)
- 'LW' - LimeWire (http://www.limewire.org)
- 'MO' - MonoTorrent (http://monotorrent.blogspot.com/)
- 'MP' - MooPolice (http://www.moopolice.de)
- 'MR' - Miro (http://www.getmiro.com)

- 'MT' - MoonlightTorrent (http://www.moonlighttorrent.com)
- 'NX' - Net Transport (http://www.xi-soft.com)
- 'OT' - OmegaTorrent (http://www.omegatorrent.com)
- 'PD' - Pando (http://www.pando.com)
- 'qB' - qBittorrent (http://www.qbittorrent.org)
- 'QD' - QQDownload (http://im.qq.com/cyclone/)
- 'QT' - Qt 4 Torrent example
- 'RT' - Retriever (http://www.halogenware.com/software/retriever.html)
- 'S~' - Shareaza alpha/beta (http://shareaza.sourceforge.net)
- 'SB' - ~Swiftbit
- 'SS' - SwarmScope
- 'ST' - SymTorrent (http://symtorrent.aut.bme.hu/)
- 'st' - sharktorrent (http://sharktorrent.com)
- 'SZ' - Shareaza (http://shareaza.sourceforge.net)
- 'TN' - TorrentDotNET
- 'TR' - Transmission (http://www.transmissionbt.com)
- 'TS' - Torrentstorm (http://www.torrentstorm.com)
- 'TT' - TuoTu (http://www.tuotu.com)
- 'UL' - uLeecher!
- 'UT' - μTorrent (http://www.utorrent.com)
- 'VG' - Vagaa (http://www.vagaa.com)
- 'WT' - BitLet (http://www.bitlet.org)
- 'WY' - FireTorrent (http://www.wyzo.com/firetorrent/)
- 'XL' - Xunlei (http://www.xunlei.com)
- 'XT' - XanTorrent (http://www.xantorrent.pwp.blueyonder.co.uk/xantorrent.zip)
- 'XX' - Xtorrent (http://www.xtorrent.com)
- 'ZT' - ZipTorrent (http://www.ziptorrent.com)

Clients which have been seen in the wild and need to be identified:

- 'BD' (example: -BD0300-)
- 'NP' (example: -NP0201-)
- 'SD' (example: -SD0100-)
- 'wF' (example: -wF2200-)
- 'hk' (example: -hk0010-) Chinese IP address, unrequestedly sends info dict in message 0xA, reconnects immediately after being disconnected, reserved bytes = 01,01,01,01,00,00,02,01

Shadow's style uses the following encoding: one ascii alphanumeric for client identification, up to five characters for version number (padded with '-' if less than five), followed by three characters (commonly '---', but not always the case), followed by random characters. Each character in the version string represents a number from 0 to 63. '0'=0, ..., '9'=9, 'A'=10, ..., 'Z'=35, 'a'=36, ..., 'z'=61, '.'=62, '-'=63.

A full explanation by Shad0w about the encoding style (including information about existing conventions on how the three characters after the version string are used) can be found here (http://forums.degreez.net/viewtopic.php?t=7070) .

For example: 'S58B-----'... for Shadow's 5.8.11

known clients that uses this encoding style are:

- 'A' - ABC (http://pingpong-abc.sourceforge.net/)
- 'O' - Osprey Permaseed (http://osprey.ibiblio.org/)
- 'Q' - BTQueue (http://btqueue.sourceforge.net/)
- 'R' - Tribler (http://www.tribler.org/)
- 'S' - Shadow's client (http://bt.degreez.net/)
- 'T' - BitTornado (http://bittornado.com)
- 'U' - UPnP NAT Bit Torrent (http://aaron2003.myftp.org/upnpclient.html)

Bram's client now uses this style... 'M3-4-2--' or 'M4-20-8-'.

BitComet (http://www.bitcomet.com/) does something different still. Its peer_id consists of four ASCII characters 'exbc', followed by two bytes x and y, followed by random characters. The version number is x in decimal before the decimal point and y as two decimal digits after the decimal point. BitLord (http://www.bitlord.com/) uses the same scheme, but adds 'LORD' after the version bytes. An unofficial patch (http://solidox.org/bc/) for BitComet once replaced 'exbc' with 'FUTB'. The encoding for BitComet Peer IDs changed to Azureus-style as of BitComet version 0.59.

XBT Client (http://xbtt.sourceforge.net/client/) has its own style too. Its peer_id consists of the three uppercase characters 'XBT' followed by three ASCII digits representing the version number. If the client is a debug build, the seventh byte is the lowercase character 'd', otherwise it is a '-'. Following that is a '-' then random digits, uppercase and lowercase letters. Example: 'XBT054d-' at the beginning would indicate a debug build of version 0.5.4.

Opera 8 previews and Opera 9.x releases (http://www.opera.com/) use the following peer_id scheme: The first two characters are 'OP' and the next four digits equal the build number. All following characters are random lowercase hexdecimal digits.

MLdonkey (http://mldonkey.sourceforge.net/Main_Page) use the following peer_id scheme: the first characters are '-ML' followed by a dotted version then a '-' followed by randomness. e.g. '-ML2.7.2-kgjjfkd'

Bits on Wheels (http://www.bitsonwheels.com/) uses the pattern '-BOWxxx-yyyyyyyyyyyy', where y is random (uppercase letters) and x depends on the version. Version 1.0.6 has xxx = A0C.

Queen Bee (http://queenbee.se/) uses Bram's new style: 'Q1-0-0--' or 'Q1-10-0-' followed by random bytes.

BitTyrant (http://bittyrant.cs.washington.edu/) is an Azureus fork and simply uses 'AZ2500BT' + random bytes as peer ID in its 1.1 version. Note the missing dashes.

TorrenTopia (http://www.torrentopia.org/) version 1.90 pretends to be or is derived from Mainline 3.4.6. Its peer ID starts with '346------'.

BitSpirit (http://www.167bt.com/intl/) has several modes for its peer ID. In one mode it reads the ID of its peer and reconnects using the first eight bytes as a basis for its own ID. Its real ID appears to use '\0\3BS' (C notation) as the first four bytes for version 3.x and '\0\2BS' for version 2.x. In all modes the ID may end in 'UDP0'.

Rufus (http://rufus.sourceforge.net/) uses its version as decimal ASCII values for the first two bytes. The third and fourth bytes are 'RS'. What then follows is the nickname of the user and some random bytes.

G3 Torrent (http://g3torrent.sourceforge.net/) starts its peer ID with '-G3' and appends up to 9 characters of the nickname of the user.

FlashGet (http://www.flashget.com/) uses Azureus style with 'FG' but without the trailing '-'. Version 1.82.1002 still uses the version digits '0180'.

BT Next Evolution (http://www.btnext.com/) is derived from BitTornado but tries to mimic Azureus style. The result is that its peer ID starts with '-NE', continues with a 4 digit version number and then directly goes on with the three characters that describe the type of client in Shad0w's peer ID style.

AllPeers (http://www.allpeers.com/) takes the sha1 hash of a user dependent string and replaces the first few characters with "AP" + version string + "-".

Qvod (http://www.qvod.com/) starts its id with the four letters "QVOD" and continues with its build number in four decimal digits (currently "0054"). The remaining 12 characters are random uppercase hexdecimal digits. There appears to be a popular modified client in China that replaces the four characters in the beginning with random bytes.

Many clients are using all random numbers or 12 zeroes followed by random numbers (like older versions of Bram's client (http://www.bittorrent.com/)).

Messages

All of the remaining messages in the protocol take the form of <length prefix><message ID><payload>. The length prefix is a four byte big-endian value. The message ID is a single decimal byte. The payload is message dependent.

keep-alive: <len=0000>

The **keep-alive** message is a message with zero bytes, specified with the length prefix set to zero. There is no message ID and no payload. Peers may close a connection if they receive no messages (**keep-alive** or any other message) for a certain period of time, so a keep-alive message must be sent to maintain the connection *alive* if no command have been sent for a given amount of time. This amount of time is generally two minutes.

choke: <len=0001><id=0>

The **choke** message is fixed-length and has no payload.

unchoke: <len=0001><id=1>

The **unchoke** message is fixed-length and has no payload.

interested: <len=0001><id=2>

The **interested** message is fixed-length and has no payload.

not interested: <len=0001><id=3>

The **not interested** message is fixed-length and has no payload.

have: <len=0005><id=4><piece index>

The **have** message is fixed length. The payload is the zero-based index of a piece that has just been successfully downloaded and verified via the hash.

Implementer's Note: That is the strict definition, in reality some games may be played. In particular because peers are extremely unlikely to download pieces that they already have, a peer may choose not to advertise having a piece to a peer that already has that piece. At a minimum "HAVE supression" will result in a 50% reduction in the number of HAVE messages, this translates to around a 25-35% reduction in protocol overhead. At the same time, it may be worthwhile to send a HAVE message to a peer that has that piece already since it will be useful in determining which piece is rare.

*A malicious peer might also choose to advertise having pieces that it knows the peer will never download. Due to this attempting to model peers using this information is a **bad idea**.*

bitfield: <len=0001+X><id=5><bitfield>

The **bitfield** message may only be sent immediately after the handshaking sequence is completed, and before any other messages are sent. It is optional, and need not be sent if a client has no pieces.

The **bitfield** message is variable length, where X is the length of the bitfield. The payload is a bitfield representing the pieces that have been successfully downloaded. The high bit in the first byte corresponds to piece index 0. Bits that are cleared indicated a missing piece, and set bits indicate a valid and available piece. Spare bits at the end are set to zero.

A bitfield of the wrong length is considered an error. Clients should drop the connection if they receive bitfields that are not of the correct size, or if the bitfield has any of the spare bits set.

request: <len=0013><id=6><index><begin><length>

The **request** message is fixed length, and is used to request a block. The payload contains the following information:

- **index**: integer specifying the zero-based piece index
- **begin**: integer specifying the zero-based byte offset within the piece
- **length**: integer specifying the requested length.

This section is under dispute! Please use the discussion page (http://wiki.theory.org /Talk:BitTorrentSpecification#Messages:_request) to resolve this!

View #1 According to the official specification, "All current implementations use 2^15 (32KB), and close connections which request an amount greater than 2^17 (128KB)." As early as version 3 or 2004, this behavior was changed to use 2^14 (16KB) blocks. As of version 4.0 or mid-2005, the mainline disconnected on requests larger than 2^14 (16KB); and some clients have followed suit. Note that block requests are smaller than pieces (>=2^18 bytes), so multiple requests will be needed to download a whole piece.

Strictly, the specification allows 2^15 (32KB) requests. The reality is near all clients will now use 2^14 (16KB) requests. Due to clients that enforce that size, it is recommended that implementations make requests of that size. Due to smaller requests resulting in higher overhead due to tracking a greater number of requests, implementers are advised against going below 2^14 (16KB).

The choice of request block size limit enforcement is not nearly so clear cut. With mainline version 4 enforcing 16KB requests, most clients will use that size. At the same time 2^14 (16KB) is the semi-*official (only* semi *because the official protocol document has not been updated) limit now, so enforcing that isn't wrong. At the same time, allowing larger requests enlarges the set of possible peers, and except on very low bandwidth connections (<256kbps) multiple blocks will be downloaded in one choke-timeperiod, thus merely enforcing the old limit causes minimal performance degradation. Due to this factor, it is recommended that only the older 2^17 (128KB) maximum size limit be enforced.*

View #2 This section has contained falsehoods for a large portion of the time this page has existed. This is the third time I (uau) am correcting this same section for incorrect information being added, so I won't rewrite it completely since it'll probably be broken again... Current version has at least the following errors: Mainline started using 2^14 (16384) byte requests when it was still the only client in existence; only the "official specification" still talked about the obsolete 32768 byte value which was in reality neither the default size nor maximum allowed. In version 4 the request behavior did not change, but the maximum allowed size did change to equal the default size. In latest mainline versions the max has changed to 32768 (note that this is the first appearance of 32768 for either default or max size since the first ancient versions). "Most older clients use 32KB requests" is false. Discussion of larger requests fails to take latency effects into account.

piece: <len=0009+X><id=7><index><begin><block>

The **piece** message is variable length, where X is the length of the block. The payload contains the following information:

- **index**: integer specifying the zero-based piece index
- **begin**: integer specifying the zero-based byte offset within the piece
- **block**: block of data, which is a subset of the piece specified by index.

cancel: <len=0013><id<=8><index><begin><length>

The **cancel** message is fixed length, and is used to cancel block requests. The payload is identical to that of the "request" message. It is typically used during "End Game" (see the Algorithms section below).

port: <len=0003><id=9><listen-port>

The **port** message is sent by newer versions of the Mainline that implements a DHT tracker. The listen port is the port this peer's DHT node is listening on. This peer should be inserted in the local routing table (if DHT tracker is supported).

Algorithms

Queuing

This section is under dispute! Please use the discussion page (http://wiki.theory.org /Talk:BitTorrentSpecification#Algorithms:_Queuing) to resolve this!

View #1 In general peers are advised to keep a few unfullfilled requests on each connection. This is done because otherwise a full round trip is required from the download of one block to begining the download of a new block (round trip between PIECE message and next REQUEST message). On links with high BDP (bandwidth-delay-product, high latency or high bandwidth), this can result in a substantial performance loss.

Implementer's note: This the 'most crucial performance item. *A static queue of 10 requests is reasonable for 16KB blocks on a 5mbps link with 50ms latency. Links with greater bandwidth are becoming very common so UI designers are urged to make this readily available for changing. Notably cable modems were known for traffic policing and increasing this might of aleviated some of the problems caused by this.*

View #2 NOTE: much of the information in this "Queuing" section is false or misleading. I'll just note that the "defaults to 5 outstanding requests" hasn't been true for a long time, "32 KB blocks" is misleading since you normally don't use 32 KB blocks, and tuning queue length by changing it and trying to measure the effects is a bad idea.

Super Seeding

(This was not part of the original specification)

The super-seed feature in S-5.5 and on is a new seeding algorithm designed to help a torrent initiator with limited bandwidth "pump up" a large torrent, reducing the amount of data it needs to upload in order to spawn new seeds in the torrent.

When a seeding client enters "super-seed mode", it will not act as a standard seed, but masquerades as a normal client with no data. As clients connect, it will then inform them that it received a piece -- a piece that was never sent, or if all pieces were already sent, is very rare. This will induce the client to attempt to download only that piece.

When the client has finished downloading the piece, the seed will not inform it of any other pieces until it has seen the piece it had sent previously present on at least one other client. Until then, the client will not have access to any of the other pieces of the seed, and therefore will not waste the seed's bandwidth.

This method has resulted in much higher seeding efficiencies, by both inducing peers into taking only the rarest data, reducing the amount of redundant data sent, and limiting the amount of data sent to peers which do not contribute to the swarm. Prior to this, a seed might have to upload 150% to 200% of the total size of a torrent before other clients became seeds. However, a large torrent seeded with a single client running in super-seed mode was able to do so after only uploading 105% of the data. This is 150-200% more efficient than when using a standard seed.

Super-seed mode is 'NOT *recommended for general use. While it does assist in the wider distribution of rare data, because it limits the selection of pieces a client can downlad, it also limits the ability of those clients to download data for pieces they have already partially retrieved. Therefore, super-seed mode is only recommended for initial seeding servers.*

```
Why not rename it to e.g. "Initial Seeding Mode" or "Releaser Mode" then?
```

Piece downloading strategy

Clients may choose to download pieces in random order.

A better strategy is to download pieces in rarest first *order. The client can determine this by keeping the initial bitfield from each peer, and updating it with every* 'have *message. Then, the client can download the pieces that appear least frequently in these peer bitfields. Note that any Rarest First strategy should include randomization among at least several of the least common pieces, as having many clients all attempting to jump on the same "least common" piece would be counter productive.*

End Game

When a download is almost complete, there's a tendency for the last few blocks to trickle in slowly. To speed this up, the client sends requests for all of its missing blocks to all of its peers. To keep this from becoming horribly inefficient, the client also sends a cancel to everyone else every time a block arrives.

There is no documented thresholds, recommended percentages, or block counts that could be used as a guide or Recommended Best Practice here.

When to enter end game mode is an area of discussion. Some clients enter end game when all pieces have been requested. Others wait until the number of blocks left is lower than the number of blocks in transit, and no more than 20. There seems to be agreement that it's a good idea to keep the number of pending blocks low (1 or 2 blocks) to minimize the overhead, and if you randomize the blocks requested, there's a lower chance of downloading duplicates. More on the protocol overhead can be found here: http://hal.inria.fr/inria-00000156/en

Choking and Optimistic Unchoking

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 choked peers once every ten seconds.

Reciprocation and number of uploads capping is managed by unchoking the four peers which have the best upload rate and are interested. This maximizes the client's download rate. These four peers are referred to as *downloaders*, because they are interested in downloading from the client.

Peers which have a better upload rate (as compared to the *downloaders*) but aren't interested get unchoked. If they become interested, the *downloader* with the worst upload rate gets choked. If a client has a complete file, it uses its upload rate rather than its download rate to decide which peers 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. Newly connected peers are three times as likely to start as the current optimistic unchoke as anywhere else in the rotation. This gives them a decent chance of getting a complete piece to upload.

Anti-snubbing

Occasionally a BitTorrent peer will be choked by all peers which it was formerly downloading from. In such cases it will usually continue to get poor download rates until the optimistic unchoke finds better peers. To mitigate this problem, when over a minute goes by without getting any piece data while downloading from a peer, BitTorrent assumes it is "snubbed" by that peer and doesn't upload to it except as an optimistic unchoke. This frequently results in more than one concurrent optimistic unchoke, (an exception to the exactly one optimistic unchoke rule mentioned above), which causes download rates to recover much more quickly when they falter.

Official Extensions To The Protocol

Currently there are a few official extensions to the protocol.

Fast Peers Extensions

- Reserved Bit: The third least significant bit in the 8th reserved byte i.e. reserved[7] |= 0x04

These extensions serve multiple purposes. They allow a peer to more quickly bootstrap into a swarm by giving a peer a specific set of

pieces which they will be allowed download regardless of choked status. They reduce message overhead by adding HaveAll and HaveNone messages and allow explicit rejection of piece requests whereas previously only implicit rejection was possible meaning that a peer might be left waiting for a piece that would never be delivered.

The specificication is documented at the BitTorrent site here: http://bittorrent.org/beps/bep_0006.html

Distributed Hash Table

- Reserved Bit: The last bit in the 8th reserved byte i.e. reserved[7] |= 0x01

This extension is to allow for the tracking of peers downloading torrents without the use of a standard tracker. A peer implementing this protocol becomes a "tracker" and stores lists of other nodes/peers which can be used to locate new peers.

The specification is documented at the BitTorrent site here: http://bittorrent.org/beps/bep_0005.html

Connection Encryption

This extension allows the creation of encrypted connections between peers. This can be used to bypass ISPs throttling BitTorrent traffic.

The specification is documented at http://www.azureuswiki.com/index.php/Message_Stream_Encryption

The documentation is fairly complete, but ideally it would be clarified on several points including guidance on when encrypted connections should be attempted, fallback procedures to regular connections etc.

Unofficial Extensions To The Protocol

Azureus Messaging Protocol

- Reserved Bit: 1

A protocol in its own right - if two clients indicate they support the protocol, then they should switch over to using it. It allows normal BitTorrent as well extension messages to be sent over it, and is documented here (http://www.azureuswiki.com/index.php /Azureus_messaging_protocol) . Currently implemented by Azureus and Transmission.

It is not possible to use both this protocol and the LibTorrent extension protocol at the same time - if both clients indicate they support both, then they should follow the semantics defined by the Extension Negotiation Protocol (http://www.azureuswiki.com/index.php /Extension_negotiation_protocol) .

WebSeeding

The possibility to seed a torrent via a web server is generally called WebSeeding. It allows the HTTP server to work as a peer in the BitTorrent network.

There are at least two specification for how to combine a torrent download with a HTTP download. The first standard, implemented by BitTornado is quite easy to implement in the client, but is intrusive on the HTTP in that it requires a script handling requests on the server side. i.e. A plain HTTP server that just serves plain files isn't enough. The benfits is that the script can be more abuse resistant. This specification is found here: http://bittornado.com/docs/webseed-spec.txt

The second specification requires slightly more from the client, but downloads from plain HTTP servers. It is specified here: http://www.getright.com/seedtorrent.html. It has been implemented by GetRight, libtorrent and Mainline.

Extension protocol

- Reserved Bit: 44, the fourth most significant bit in the 6th reserved byte i.e. reserved[5] |= 0x10

This is a protocol for exchanging extension information and was derived from an early version of azureus' extension protocol. It adds one message for exchanging arbitrary handshake information including defined extension messages, mapping extensions to specific message IDs. It is documented here: http://www.libtorrent.org/extension_protocol.html and is implemented by libtorrent, uTorrent and Mainline.

It is not possible to use both this protocol and the Azureus Messaging Protocol at the same time - if both clients indicate they support both, then they should follow the semantics defined by the Extension Negotiation Protocol (http://www.azureuswiki.com/index.php /Extension_negotiation_protocol) .

Extension Negotiation Protocol

- Reserved bits: 47 and 48

These bits are used to allow two clients that support both the Azureus Messaging Protocol and LibTorrent's extension protocol to decide which of the two extensions should be used for communication, and is defined here (http://www.azureuswiki.com/index.php /Extension_negotiation_protocol) .

BitTorrent Location-aware Protocol 1.0

- Reserved Bit: 21

A Protocol, considering peers location (in geographical terms) for better performance. Specification can be found here (http://wiki.theory.org/BitTorrent_Location-aware_Protocol_1.0_Specification) .

SimpleBT Extension Protocol

- Reserved Bits: fist reserved byte = 0x01, following bytes may need to be set to zero

An extension using message id 9 to add peer exchange and connection statistics exchange. The specification can be found here (http://web.archive.org/web/20031002201124/btfans.3322.org/simplebt/ProtocalExtension.txt) . The extension was in use in SimpleBT 0.32 to 0.36.1. Later versions of SimpleBT were called BitComet and used the similar but incompatible BitComet Extension Protocol.

BitComet Extension Protocol

- Reserved Bits: first two reserved bytes = "ex"

There appears to be no official documentation.

In this protocol a peer announces the supported extensions by sending a message <len=0001+X><id=0xA0><extension 1>...<extension X> where <extension n> is (usually) the message id of the supported extension. When an extension consists of multiple messages, all ids need to be mentioned.

Extensions currently in use (TODO: reverse engineer semantics):

- 0xA0 (EXT_SUPPORT) see above, needs to be included in its parameter list
- 0xA1 (EXT_PEERREQ) ask for peer exchange, used in conjunction with EXT_PEERS
- 0xA2 (EXT_PEERS) in reply to EXT_PEERREQ and for updates afterwards
- 0xA3 (EXT_AUTH_SEED) appeared in BitComet 0.53, used in conjunction with EXT_AUTH_CRYPTOED
- 0xA4 (EXT_AUTH_CRYPTOED)
- 0xA5 (EXT_CONNGRANT) appeared in BitComet 0.48, used in conjunction with EXT_CONNACCEPT
- 0xA6 (EXT_CONNACCEPT)
- 0x06 (?) announced by BitSpirit instead of EXT_CONNACCEPT
- 0xA7 (EXT_CHAT_MESSAGE) appeared in BitComet 0.53, vanished in 0.71
- 0xA9 (EXT_HASH_REQ) appeared in BitComet 0.54, vanished in 0.71, used in conjunction with EXT_HASH
- 0xAA (EXT_HASH)
- 0xAB (EXT_REPORT_RATE_old) appeared in BitComet 0.54, was replaced by EXT_REPORT_RATE_new in 0.57
- 0xAC (EXT_REPORT_INFO) appeared in BitComet 0.54, vanished in 0.71, reappeared in 0.82
- 0xAD (EXT_REPORT_RATE_new) appeared in BitComet 0.57, vanished in 0.75, reappeared in 0.82
- 0xAE (EXT_BC_PASSPORT) appeared in BitComet 0.75
- 0xAF (EXT_DHE_PREFERRED) appeared in BitComet 0.75
- 0xB0 (?) appeared in BitComet 0.86
- 0xC0 (?) does not correspond to a message id, appeared in BitComet 0.49

A minimum implementation needs only accept EXT_SUPPORT, but EXT_PEERREQ and EXT_PEERS are supported by all known implementations.

Reserved Bytes

The reserved bits are numbered 1-64 in the following table for ease of identification. Bit 1 corresponds to the most significant bit of the first reserved byte. Bit 8 corresponds to the least significant bit of the first reserved byte (i.e. byte[0] |= 0x01). Bit 64 is the least significant bit of the last reserved byte i.e. byte[7] |= 0x01

An orange bit is a known unofficial extension, a red bit is an unknown unofficial extension.

Reserved Bits

Bit	Use	Azureus	BitComet	MainLine	MonoTorrent	µTorrent	libtorrent	KTorrent	BitLord	XBT	Transmission
1	Azureus Extended Messaging	Yes	?	?	?	?	No	No	No	No	Yes
1-16	BitComet Extension protocol	No	Yes	No	No	No	No	No	Yes	No	No
21	BitTorrent Location-aware Protocol 1.0	No	No	No	No	No	No	No	No	No	No
44	Extension protocol	Yes	?	Yes	Yes	Yes	Yes	No	No	Yes	Yes
47 - 48	Extension Negotiation Protocol	Yes	No	No	No	No	No	No	No	No	Yes
61	NAT Traversal	No	?	Yes	?	?	No	?	?	No	?
62	Fast Peers	No	?	Yes	Yes	?	Yes	Yes	No	No	?
63	XBT Peer Exchange	No	?	No	?	?	No	?	?	Yes	?
64	DHT	No	?	Yes	Yes	Yes	Yes	Yes	No	No	?
64	XBT Metadata Exchange	No	?	No	?	?	No	?	?	Yes	?

Change Log

Put your changes below this line, so that the most recent changes appear first. The change log should be purged from time to time. Please preserve the last month's worth of change logs.

HighInBC - 2008-02-12 - Minor change. Added parsed result for final example of bencoded dictionary, added <nowiki> tags to top example to avoid issues.

amc1 - 2007-09-12 - Azureus now supports the LibTorrent Extension Protocol (LTEP, as I refer to it).

amc1 - 2007-08-16 - Added description and links to Extension Negotiation Protocol, and formatted the reserved bits table a bit.

Denial - 2007-08-13 - Added description of the SimpleBT and BitComet extension protocols

amc1 - 2007-07-16 - Added SymTorrent's peer ID identifier, as well as some other clients I've come across which need to be identified.

amc1 - 2007-07-14 - Corrected information about Shadow's style of peer ID - existing text made incorrect assumptions.

mitchman - 2007-07-12 - Clarified the Opera peed-id

roee88- 2007-06-11 - Added LH-ABC peer_id

Boian V Petkantchin - April 26, 2007. Listed another unoffcial extension - BitTorrent Location-aware Protocol 1.0. Included it in the table of reserved bits.

daniel-gl at gmx.net - Corrected reserved bit numbers - Added NAT traversal and XBT extensions to table

stuge - 2006-11-20 - Added Queen Bee peer_id

EHeM - 2006-10-16 - Update of view #1 on *Message: request* to hopefully reflect a possible compromise position. - Trimming of changelog.

EHeM - 2006-10-11 - Firmly marked the *Message: request* section as being under dispute - Firmly marked the *Algorithms: Queuing* section as being under dispute - Removed personal insults by uau, could we be reasonable humans and try to resolve disputes peacefully? Perhaps using the attached talk/discussion (http://wiki.theory.org/Talk:BitTorrentSpecification) page?

Joris Guisson - 2006-09-12 - Added KTorrent specific information (peer id and extension table)

Arvid - 2006-09-12 - Added Extension protocol to the unofficial extensions section

Alan - 2006-09-12 -Removed some older history. Update the reserved bytes section.

Alan - 2006-09-11 - Added section about Extensions to Protocol

uau - 2006-08-17 - Added warning about the misleading information in the Queuing and requests packet sections.

WikiWordsAreEllFourEmmThree - 2006-07-08 - Added BitPump peer_id

daniel-gl at gmx.net - 2006-04-28 - Added Bits on Wheels & BitLord peer_id.

DennisHolmes - 2006-04-22 - Added Enhanced CTorrent peer_id

WikiWordsAre&#65533;berGay - 2006-04-16 - Anti-snubbing '*is* part of the official protocol. Check out the paper on BitTorrent economics at http://www.bittorrent.org

JoshElsasser - 2006-04-11 - Added Transmission peer_id

daniel-gl at gmx.net - 2006-03-23 - Added Tribler peer_id

MaSiniavine - Corrected dictionary example

Juanjo 2006-03-10 - Added Lphant peer ID

EHeM 2006-03-07 - Added mention of another parameter to the baseline for queuing. - Another link adjustment. - Added exposition on request block size. - Sample link changes, the domain "example.com" is explicitly reserved for examples, as such that should be used instead of spam.com. - Added qualification to recommended number of peers.

EHeM 2006-03-01 (minor) - Link changes, the official BitTorrent pages are no longer on bitconjurer.org, but bittorent.com. - Removed unneeded line breaks from paragraph. - Typo fixes in changelog (yeah, I suppose do have a bit of vanity)

EHeM 2006-03-01 - Restored the section on queueing, as it '*is* a highly crucial performance item. Feel free to rewrite me if you desire, uau. The developer's list, http://lists.ibiblio.org/mailman/listinfo/bittorrent is a better place for debates. - Trimmed changelog entries older than one year, the above specifies one month, but this is changing slowly so more history seems pertinent.

haylegend - 2006-01-08 - Added Retriever's peer id.

uau - 2005-12-13 - Fixed sizes in request message description AGAIN. They had been changed to incorrect values. - Removed "Queuing" section. It had so many errors and inaccuracies that it did more harm than good as it was, and I didn't feel like rewriting it.

```
Last edit: Tue, 12 Sep 2006 22:38:05 -0700
(TresNi)
Revisions: 158
```

Retrieved from "http://wiki.theory.org/BitTorrentSpecification"
Category: PhpWiki