

# **BitTorrent Protocol Specification**

## **V 1.0**

CSI 5321  
Dec 12 2004

Submitted by  
**Arun Chokkalingam & Firasath Riyaz**



# 1 Torrent (MetaInfo) File

The torrent file contains the information about the file(s) we will be downloading. This is bencoded.

## 1.1 Bencoding

Bencoding is a way to specify the data in a terse format. The below table provides the format for the different types in Bencoding.

Type	Description	Format	Example
Strings	Normal Strings [series of continuous characters]	<i>&lt;string length&gt;:&lt;string data&gt;</i>	<i>7:network</i> Represents the string network
Integers	Normal integers	<i>i&lt;integer&gt;e</i>	<i>i3e</i> represents 3.
Lists	They are lists of types [strings, I integers, lists, dictionaries].	<i>l&lt;bencoded type&gt;e</i>	<i>l8:advanced7:networke</i> represents the list of two strings: ["advanced", "network"]
Dictionaries	They are a mapping of keys to values	<i>d&lt;bencoded string&gt;&lt;bencoded element&gt;e</i>  <b>Note:</b> The keys are bencoded strings	<i>d7network:7:csi53216:baylore</i>

## 1.2 Structure of Torrent(MetaInfo) File

- a. The following table gives the structure of a **single-file** torrent [does not have a sub-directory structure].

Key	Description
Info	A dictionary that describes the files
- length	Length of file in bytes.(integer)
-md5sum(optional)	A 32 character hexadecimal string corresponding to the MD5 sum of the file.
-name	The filename of a string(string)
-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.(raw binary encoded)
Announce	The announce URL of the tracker
Announce-list(optional)	This is an extension to the official specification, which is also backwards compatible. This key is used to implement lists of backup trackers. The full specification can be found at <a href="http://home.elp.rr.com/tur/multitracker-spec.txt">http://home.elp.rr.com/tur/multitracker-spec.txt</a> .
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 text comments.(string)
Created by(optional)	Name and version of the program used to create.

- b. The following table gives the structure of a **mutli-file** torrent [having a sub-directory structure].

Key	Description
Info	A dictionary that describes the files
files	a list of dictionaries, one for each file. Each dictionary in this list contains the following keys
- length	Length of file in bytes.(integer)
-md5sum(optional)	A 32 character hexadecimal string corresponding to the MD5 sum of the file.
-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".
oname	the name of the top-most directory in the structure -- the directory which contains all of the files listed in the above <b>files</b> list. (string)
opiece length	Number of bytes in each piece(integer)
opieces	String consisting of the concatenation of all 20-byte SHA1 hash values, one per piece.(raw binary encoded)
Announce	The announce URL of the tracker
Announce-list(optional)	This is an extension to the official specification, which is also backwards compatible. This key is used to implement lists of backup trackers. The full specification can be found at <a href="http://home.elp.rr.com/tur/multitracker-spec.txt">http://home.elp.rr.com/tur/multitracker-spec.txt</a> .
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 text comments.(string)
Created by(optional)	Name and version of the program used to create.

## 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, constrained by the fact that too small a piece size will result in a large .torrent metadata file, and piece sizes too large cause inefficiency. The general rule of thumb seems to be to pick the smallest piece size that results in a .torrent file no greater than approx. 50 - 75 kB. 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 bytes.

Attached is a sample metafile for a single file.



C:\Adv Net\Copy (2)  
of OurBitTorrent\c.tor

Attached is the sample metafile created by the tracker for a directory of files.



C:\nbt\  
filesetfolder.torrent

## 2 Tracker HTTP Protocol

The tracker is an HTTP 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 for details.). You can also specify the tracker to send the gzip and send the information back. We are investigating this.

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

Parameter	Description
<b>info_hash</b>	20-byte SHA1 hash of the <i>value</i> of the <b>info</b> key from the Metainfo file. Note that the <i>value</i> will be a bencoded dictionary, given the definition of the <b>info</b> key above.
<b>peer_id</b>	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. <i>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 <a href="#">peer_id</a> below for common client encodings of this field.</i>
<b>port</b>	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.
<b>uploaded</b>	The total amount uploaded so far, encoded in base ten ascii.
<b>downloaded</b>	The total amount downloaded so far, encoded in base ten ascii.
<b>left</b>	The number of bytes this client still has to download, encoded in base ten ascii.
<b>event</b>	If specified, must be one of <b>started</b> , <b>completed</b> , or <b>stopped</b> . If not specified, then this request is one performed at regular intervals.

<b>-started</b>	The first request to the tracker <b>must</b> include the event key with the <b>started</b> value.
<b>-stopped</b>	Must be sent to the tracker if the client is shutting down gracefully.
<b>-completed</b>	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 solely on this event
<b>ip</b>	Optional. The true IP address of the client machine, in dotted quad format or rfc3513 defined hexed IPv6 address. <i>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.</i>
<b>numwant</b>	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.



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

Key	Description
<b>failure reason</b>	If present, then no other keys may be present. The value is a human-readable error message as to why the request failed (string).
<b>interval</b>	Interval in seconds that the client should wait between sending regular requests to the tracker
<b>complete</b>	number of peers with the entire file, i.e. seeders (integer)
<b>incomplete</b>	number of non-seeder peers, aka "leechers" (integer)
<b>peers</b>	The value is a list of dictionaries, each with the following keys
<b>-peer id</b>	peer's self-selected ID, as described above for the tracker request (string)
<b>-ip</b>	peer's IP address (either IPv6 or IPv4) or DNS name (string)
<b>-port</b>	peer's port number (integer)

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.*

### 3 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)

<code>http://spam.com/announce</code>	-> <code>http://spam.com/scrape</code>
<code>http://spam.com/x/announce</code>	-> <code>http://spam.com/x/scrape</code>
<code>http://spam.com/announce.php</code>	-> <code>http://spam.com/scrape.php</code>
<code>http://spam.com/a</code>	-> (scrape not supported)
<code>http://spam.com/announce?x=2%0644</code>	-> <code>http://spam.com/scrape?x=2%0644</code>
<code>http://spam.com/announce?x=2/4</code>	-> (scrape not supported)
<code>http://spam.com/x%064announce</code>	-> (scrape not supported)

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

The response of this HTTP GET method is a "text/plain" 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 value. The value of that key is yet another nested 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:filesd20:.....d8:completei5e10:downloadedi50e10:incompletei10eeee
```

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

## 4 Peer wire protocol

### Overview:

The peer protocol facilitates the exchange of pieces as described in the **meta-info** 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 meta-info 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 block is typically lesser in size than the piece.***

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. **Simply saying "when you are choked by the peer you can not download pieces from the peer until you are unchoked".**
- **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 it. **Typically the peer will send this message after it had received a Bit-Field message from the client telling the peer the list of pieces it has.**

*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.

- **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. 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 the same peer\_id that is transmitted in tracker requests.

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

### **A typical Hand Shake message in java will look like this**

"out" is the output-stream we are writing to

Then Handshake is :

```
out.write(19);

out.write("BitTorrent protocol".getBytes());

byte[] reservedbytes = new byte[8];

out.write(reservedbytes);

out.write(info_hash);

out.write(peer_id);
```

info\_hash and peer\_id are strings But while writing to the stream converting them to bytes is a good idea.

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.

### **HandShakeReply:**

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.*

**If everything matches the receiver responds with handshake message with peer\_id field modified to its own ID.**

**peer\_id:** There are mainly three conventions how to encode client and client version information into the peer\_id, Azureus-style, 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:

- 'AZ' -  [Azureus](#)
- 'BB' -  [BitBuddy](#)
- 'CT' -  [CTorrent](#)
- 'MT' -  [MoonlightTorrent](#)
- 'LT' -  [libtorrent](#)
- 'BX' - Bittorrent X
- 'TS' -  [Torrentstorm](#)
- 'TN' - TorrentDotNET
- 'SS' - SwarmScope
- 'XT' -  [XanTorrent](#)

Shadow's style uses the following encoding: one ascii alphanumeric for client identification, three ascii digits for version number, '----', followed by random numbers.

For example: 'S587----'

known clients that uses this encoding style are:

- 'S' -  [Shadow's client](#)
- 'U' -  [UPnP NAT Bit Torrent](#)
- 'T' -  [BitTornado](#)
- 'A' -  [ABC](#)

Bram's client now uses this style... 'M3-4-2--'.

[BitComet](#) does something different still. Its peer\_id consists of four ASCII characters 'exbc', followed by a null byte, followed by a single ASCII numeric digit, followed by random characters. The digit seems to denote the version of the software, though it appears to have no connection with the real version number. The digit is incremented with each new BitComet release.

Many clients are using all random numbers or 12 zeroes followed by random numbers (like older versions of [Bram's client](#)).

## Tracker Handshake: [added on dec 12 2005]

This was the major hurdle we faced while we were implementing the uploading part of our BitTorrentClient. Whenever we interact with the BitTorrent Tracker that we downloaded from the Bittorrent website the tracker sends a half handshake message → a handshake message without the peer id. Only if the bittorrent client responds to this with a handshake message the tracker considers the client as a peer that can be considered as one of the sources of the file. And only after this is our client listed to the other bittorrent clients as sources that they can upload the file from.

## 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 character. 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.

Java example: `outPutStream.write(0);`

### Choke: <len=0001><id=0>

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

Java example: `outPutStream.write(1);`

`outPutStream.writeByte(0);`



**Unchoke: <len=0001><id=1>**

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

Java example: `outPutStream.write(1);`

`outPutStream.writeByte(1);`

**Interested: <len=0001><id=2>**

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

Java example: `outPutStream.write(1);`

`outPutStream.writeByte(2);`

**Not interested: <len=0001><id=3>**

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

Java example: `outPutStream.write(1);`

`outPutStream.writeByte(3);`

**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 been successfully downloaded. Initially the peers tell each other about the pieces they have using Bit-Field message later when they have downloaded another piece, they use the “have” message to tell the other peers now it also has this piece.

Java example: `outPutStream.write(5);`

`outPutStream.writeByte(4);`

`outPutStream.write(piece_index);`

**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.*

Java example: `outPutStream.write(1 + 10);`

`outPutStream.writeByte(5);`

`outPutStream.write(bitfield);`

where, bitfield is a byte array of size 10.

**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 value must not exceed  $2^{17}$  bytes, typical values are  $2^{15}$  bytes.

The observant reader will note that a block is typically smaller than a piece (which is commonly  $\geq 2^{18}$  bytes). A client should close the connection if it receives a request for more than  $2^{17}$  bytes.

Java example: `outPutStream.write(13);`

`outPutStream.writeByte(6);`

`outPutStream.write(piece_index);`

`outPutStream.write(begin);`

`outPutStream.write(length);`

**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.

Java example: `outPutStream.write(9 + 256);`

`outPutStream.writeByte(7);`

`outPutStream.write(piece_index);`

`outPutStream.write(begin);`

`outPutStream.write(block);`

block is a byte array of size 256 for this example

**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).

Java example: `outPutStream.write(13);`

`outPutStream.writeByte(8);`

`outPutStream.write(piece_index);`

`outPutStream.write(begin);`

`outPutStream.write(length);`

## 4.1 Algorithms:

### 4.1.1 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 download, 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?*

#### **4.1.2 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.*

### 4.1.3 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.*

### 4.1.4 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.

#### 4.1.4.1 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 a single piece from a particular 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.

## 5 Suggestions:

- Have a multicast for each torrent. This however gives rise to the problem of not having enough multicast addresses. We can overcome this by having a set of multicast addresses for each tracker and the tracker maps the torrents to the multicast addresses. The maintenance is done by LRU.
- Anonymity of the producer and consumer is not there. We can overcome this using the anonymous multicast reception. We are “investigating this”
- **Ref: An efficient protocol for anonymous multicast and reception**
- If the tracker sends the peer list in this format ... completed peers, a blank peer, and the downloading peers then we can restrict the end game to the completed peers.
- <http://wiki.theory.org/BitTorrentWishList>
- We can add encryption to the data packets in case we want to restrict the file to a specific group of peers This might be useful incase when you are sharing the files and you don't want the people in the network to know its contents. Say your company is intercepting the packets you send.)

## 6 Reference:

<http://wiki.theory.org/BitTorrentSpecification>