BitTorrent

A technical description of the BitTorrent protocol

1 Introduction

File sharing is a phenomen that is well known in the entire community. Many people are using it, and a large amount of bandwith on the Internet is used up by different file sharing applications/protocols. CacheLogic suggests that about 30% of all Internet bandwith was used by BitTorrent at the end of 2004[1].

BitTorrent is one of the most popular ways of sharing and downloading files. Podcasts are often distributed with it, as well as open source projects, such as Red Hat Linux. Besides legal distribution of files, BitTorrent is often used to illegaly share and download copyrighted material.

The technology within the protocol makes it possible to distribute large amounts of data without the need of a high capacity server, and expensive bandwith. This is probably the main reason for using this protocol instead of traditional downloading from HTTP or FTP servers. For those sharing and downloading copyrighted files, BitTorrent is benificial due to the fact that it is decentralized.

2 Overview of BitTorrent

BitTorrent is a file sharing protocol, implementing peer-to-peer technology. The main idea is to releive a single server from heavy demand, as the number of users increase, by dividing files into smaller pieces. A piece that is downloaded by a peer is then shared for uploading by that peer. The key components in the protocol are:

- Metainfo file. A metainfo file is used to describe metadata about the file(s) being shared, and the tracker, so that clients and trackers can administrate downloads. Also referred to as torrent file, and has the file ending .torrent.
- Tracker. A central server keeping track of peers and seeds sharing and downloading the file(s).
- Peer. A user on the network downloading the file(s), and sharing the parts that have been successfully downloaded.
- Seed. Basically a peer that has the complete file or files available for sharing. At least one seed is needed in the beginning for the sharing to be possible.
- Client. An application implementing the BitTorrent protocol.

The procedure of sharing a file is rouhly implemented as follows:

1. A metainfo file (also called torrent) containing information about the file and a central server called a tracker is published, usually on a web page.
2. The user downloads the metainfo file, and opens it with a BitTorrent client application. The client informs the tracker of its existance, and receives a list of other peers downloading or sharing the file in question.
3. The peers exchange information about who has which pieces, and download those pieces that are interesting. When a piece is successfully downloaded by a peer, it is announced as available to other peers.
4. When the complete file has been downloaded, the peer may choose to remain online, to share the file.

This describes the behaviour of the original BitTorrent 1.0 protocol. Currently, there are some modifications and extensions that may differ in behaviour, but those are discussed later in the document.

3 BitTorrent protocol 1.0

This is a description of the original BitTorrent 1.0 protocol, designed by Bram Cohen [2], as well as some defacto standards and implementations [3]. Included in this description is the metafile structure, tracker protocol and peer wire protocol.

3.1 Metainfo file structure

All data in metainfo (torrent) files are bencoded. The definition of bencoding is beyond the scope of this paper, for more information see [3]. Suffice to say, bencoding is a way to organize data. Types supported are: byte strings, integers, lists and dictionaries, where the keys in the dictionary are bencoded strings, and the values are of any bencoded type, including dictionaries.

A metainfo file can be in one of two modes:

- Single-file mode. One single file is described by the torrent, and there is no directory structure.
- Multi-file mode. Multiple files are described, with possible directory structure described as well.

A torrent file always contains a single dictionary that consists of several keys.

One key is *info*, which is structured differently depending on the mode. In single file mode, the dictionary has following entries:

- *length:* The length of the file in bytes
- *name:* The name of the file
- *piece length:* The piece length in bytes
- *pieces:* A 20 byte SHA1 hash is calculated for each piece of the file, and all hash codes are then concatenated to one byte string

In multi file mode the *info* dictionary looks as follows:

- *files:* A list of dictionaires, each corresponding to each file. Each dictionary has the keys:
    - *length:* The length of the file in bytes
    - *path:* A list of strings, where each element corresponds to a directory name, except the last string, which is the file name
- *name:* The name of the root directory of the files
- *piece length:* The piece length in bytes
- *pieces:* A 20 byte SHA1 hash is calculated for each piece of the files, and all hash codes are then concatenated to one byte string

The other key in the main dictionary is *announce*, and it corresponds to a string of the URL of the tracker.

There are some additional parts of the metainfo file not bescrived in the official protocol description [2]. They are not required for a torrent file to be correct, but may be included if there is need for them. These are:

- *md5sum:* In single file mode, a key directly in the *info* dictionary, and in multi file mode a key in each of the dictionaries in the *files* list. The value is a 32 byte string, representing the hexadecimal MD5 sum of each file.
- *announce-list:* A key in the main dictionary of the torrent file. This is used for implementing multiple trackers [4]. The entry corresponding to this key is a list of list of strings, where each string is a tracker URL. Multiple trackers are described more in depth later in this document.
- *creation date:* A key in the main dictionary. Corresponding to the key is an integer describing time when the torrent was created, in standard UNIX epoch time.
- *comment:* A key in the main dictionary. Contains any string of text the creator of the metainfo file would like to add.
- *created by:* A key in the main dictionary. Corresponds to a string giving the name and version of the program that created the torrent file.

3.2 Tracker protocol

The tracker protocol is implemented on top of HTTP/HTTPS. This means that the machine running the tracker runs a HTPP or HTTPS server, and has the behaviour described belove:

1. The client sends a GET request to the tracker URL, with certain CGI variables and values added to the URL. This is done in the standard way, i.e if the base URL is “http://some.url.com/announce”, the full URL would be of this form: “http://some.url.com/announce?var1=value1&var2=value2&var3=value3”.
2. The tracker responds with a “text/plain” document, containing a bencoded dictionary. This dictionary has all the information required for the client.
3. The client then sends rerequests, either on regular intervals, or when an event occurs, and the tracker responds.

The CGI variables and values added to the base URL by the client sending a GET request are:

- **info_hash:** The 20 byte SHA1 hash calculated from whatever value the *info* key maps to in the metainfo file.
- **peer_id:** A 20 character long id of the downloading client, random generated at start of every download. There is no formal definition on how to generate this id, but some client applications have adapted some semiformal standards on how to generate this id. Refer to [3] for more information.
- **ip:** This is an optional variable, giving the IP address of the client. This can usually be extracted from the TCP connection, but this field is useful if the client and tracker are on the same machine, or behind the same NAT gateway. In both cases, the tracker then might publish an unroutable IP address to the client.
- **port:** The port number that the client is listening on. This is usually in the range 6881-6889.
- **uploaded:** The amount of data uploaded so far by the client. There is no official definition on the unit, but generally bytes are used
- **left:** How much the user has left for the download to be complete, in bytes.
- **event:** An optional variable, corresponding to one of four possibilities:

    - *started:* Sent when the client starts the download
    - *stopped:* Sent when the client stops downloading
    - *completed:* Sent when the download is complete. If the download is complete at start up, this message should not be sent.
    - *empty:* Has the same effect as if the **event** key is nonexistant. In either case, the message in question is one of the messages sent with regular intervals.

There are some optional variables that can be sent along with the GET request that are not specified in the official description of the protocol, but are implemented by some tracker servers:

- **numwant:** The number of peers the client wants in the response.
- **key:** An identification key that is not published to other peers. **peer_id** is public, and is thus useless as authorization. **key** is used if the peer changes IP number to prove it's identity to the tracker.
- **trackerid:** If a tracker previously gave its trackerid, this should be given here.

As mentioned earlier, the response is a "text/plain" response with a bencoded dictionary. This dictionary contains the following keys:

- *failure reason:* If this key is present, no other keys are included. The value mapped to this key is a human readable string with the reason to why the connection failed.
- *interval:* The number of seconds that the client should wait between regular rerequests.
- *peers:* Maps to a list of dictionaries, that each represent a peer, where each dictionary has the keys:
    - *peer_id:* The id of the peer in question. The tracker obtained this by the peer_id variable in the GET request sent to the tracker.
    - *ip:* The address of the peer, either the IP address or the DNS domain name.
    - *port:* The port number that the peer listens on.

These are the keys required by the official protocol specification, but here as well there are optional extensions:

- *min interval:* If present, the client must do rereqests more often than this.
- *warning message:* Has the same information as *failure reason*, but the other keys in the dictionary are present.
- *tracker id:* A string identificating the tracker. A client should resend it in the **trackerid** variable to the tracker.
- *complete:* This is the number of peers that have the complete file available for upload.
- *incomplete:* The number of peers that not have the complete file yet.

3.4 Peer wire protocol

The peer wire (peer to peer) protocol runs over TCP. Message passing is symmetric, i.e. messages are the same sent in both directions.

When a client wants to initiate a connection, it sets up the TCP connection and sends a handshake message to the other peer. If the message is acceptable, the receiving side sends a

handshake message back. If the initiator accepts this handshake, message passing can initiate, and continues indefinitely.

All integers are encoded as four byte big-endian, except the first length prefix in the handshake.

Handshake message
The handshake message consists of five parts:

- A single byte, containing the decimal value 19. This is the length of the character string following this byte.
- A character string "BitTorrent protocol", which describes the protocol. Newer protocols should follow this convention to facilitate easy identification of protocols [2].
- Eight reserved bytes for further extension of the protocol. All bytes are zero in current implementations.
- A 20 byte SHA1 hash of the value mapping to the *info* key in the torrent file. This is the same hash sent to the tracker in the **info_hash** variable.
- The 20 byte character string representing the peer id. This is the same value sent to the tracker.

If a peer is the first recipient to a handshake, and the info_hash doesn't match any torrent it is serving, it should break the connection. If the initiator of the connection receives a handshake where the peer id doesn't match with the id received from the tracker, the connection should be dropped.

Each peer needs to keep the state of each connection. The state consists of two values, *interested* and *choking.* A peer can be either interested or not in another peer, and either choke or not choke the other peer. Choking means that no requests will be answered, and interested means that the peer is interested in downloading pieces of the file from the other peer.

This means that each peer needs four Boolean values for each connection to keep track of the state.
am_interested
am_choking
peer_interested
peer_choking

All connections start out as not interested and choking for both peers. Clients should keep the *am_interested* value updated continuously, and report changes to the other peer.

The messages sent after the handshaking are structured as:
[message length as an integer] [single byte describing message type] [payload]

Keep alive messages are sent with regular intervals, and they are simply a message with length 0, and no type or payload.

Type 0, 1, 2, 3 are *choke, unchoke, interested* and *not interested* respectively. All of them have length 1 and no payload. These messages simply describe changes in state.

Type 4 is a *have*. This message has length = 5, and a payload that is a single integer, giving the integer index of which piece of the file the peer has successfully downloaded and verified.

Type 5 is *bitfield.* This message is only sent directly after handshake. It contains a bitfield representation of which pieces the peer has. The payload is of variable length, and consists of a bitmap, where byte 0 corresponds to piece 0-7, byte 1 to piece 8-15 etc. A bit set to 1 represents having the piece. Peers that have no pieces can neglect to send this message.

Type 6 is a *request.* The payload consists of three integers, *piece index, begin* and *length*. The *piece index* decides within which piece the client wants to download, *begin* gives the byte offset within the piece, and *length* gives the number of bytes the client wants to download. *Length* is usually a power of two.

Type 7 is a *block.* This message follows a *request.* The payload contains *piece index, length* and the data itself that was requested.

Type 8 is *cancel.* This message has the same payload as *request* messages, and it is used to cancel requests made.

Peers should continuously update their interested status to neighbours, so that clients know which peers will begin downloading when unchoked.

4 New features

4.1 DHT (Distributed Hash Table)

Due to the extensive use of P2P file sharing systems, such as Napster, Gnutella and Freenet, more and more research in this area has been conducted to develop a more scalable and decentralized approach. Napster suffered from having a centralized directory service which led to the vulnerability for attacks. Gnutella uses query flooding which broadcasts a message over the whole network. This type of flooding is not suitable for large networks and it's not sufficient enough to the use of file sharing.
In order to avoid these problems, different systems using DHT has been developed. To mention some of them, Tapestry, Chord, Pastry and CAN [5].
The difference between these systems is the routing algorithms used. Scalability is straight-forward based on the efficiency of their routing algorithms. In DHT systems, keys are produced from files, using a hash function for instance, and every node in the network holds a range of keys. Nodes have a unique ID and also possess a routing table, providing neighbor nodes. The most important functionality of DHT is the *lookup(key)*, which gives the ID of the node having a match to the key. This gives the opportunity to *get* and *put* files based on the key. However, DHT only provides exact matches on keys. There are overlying techniques to accomplish more complex queries. There are no centralized servers containing information about nodes using the system.

All routing algorithms share the same property that the unique ID shares the same namespace as the key. They differ in the way a node decides to what node the query will be sent. The algorithms take different issues in account and each has different properties. [5]

- Latency between two nodes, the routing table can be optimized so that the path latency will be as small as possible.
- A node is responsible for the keys closest to its unique numerical ID. The namespace is considered as a circle, where the nodes and keys are spread out. The node's set of neighbors consists of those that have the closest numerical ID to the node and some other nodes around the circle that are chosen in a bit more complex way. The node then routes a query to the node that has the longest shared prefix with the key.
- The two strategies stated above uses a one dimensional key space. Another strategy is to use a d-dimensional toroidal key space, whereas a node has a hypercubal region of this key space. The neighbors of a node are the nodes that own contiguous hypercubes and the routing is done by sending the query to the node closest to the key.

There is a lot of discussion concerning which one of the routing algorithms is the best. They all have their advantages, and plenty of research is made to pick out the different advantages into an optimal algorithm.

4.2 Multi trackers

To avoid interruption when a tracker fails, multiple trackers can be used for one torrent. To use this feature there are an extensions to the BitTorrent 1.0 metadata format. [4]

- "*announce-list*" – this is an addition to the standard "*announce*"-key. It refers to a list of list to addresses to trackers, whereas the URLs are divided into tiers. The client will ignore the standard "*announce*"-key if it is compatible with the multitracker specification.

The announce-lists tiers will be accessed sequentially and all URLs in a tier must be checked before moving on to the next tier. The URLs within a tier are shuffled first, and then accessed in order. Another feature is that if a connection with a tracker was successful it will be moved up to the top of the tier.

Examples:
['announce-list'] = [[tracker1] , [backup1] , [backup2]]
This announce-list consists of three tiers. The trackers will be chosen in the following order. First try tracker1, if no connection is made, try backup1 and backup2 respectively. The order will be the same next announce.

['announce-list'] = [[tracker1,tracker2,tracker3]]
The three trackers will be shuffled at start, let's just say that it has been shuffled. If tracker1 cannot be reached, but tracker2 can, then the list would look like this:
[[tracker2,tracker1,tracker3]]
Next time, if neither tracker2 or tracker1 can be reached, but tracker3, the list would look like this:
[[tracker3,tracker2,tracker1]]

['announce-list'] = [[tracker1,tracker2] , [backup1]]
The first tier *[tracker1,tracker2]* will be shuffled at start. So tracker1 or tracker2 will always be tried before backup1, but the order of visits can be varied.

4.3 Encryption

Due to the fact that BitTorrent traffic takes up around 30% of all Internet bandwidth, some ISPs have started taken countermeasures to the BitTorrent traffic. They block BT traffic, and all clients not supporting encryption will thereby not be able to use their BitTorrent software. Therefore, encryption of BitTorrent traffic was implemented in some clients to fool the ISPs blocking system. Of course, both encrypting and recognition of BitTorrent traffic, takes up resources. That's why some developers claim encryption is a bad choice, and respectively that ISPs don't implement this kind of blocking, also known as traffic shaping or bandwidth throttling.
[6]

5 References

[1]
Cachelogic, BitTorrent bandwidth usage
http://www.cachelogic.com/research/2005_slide06.php

[2]
The official protocol specification
http://www.bittorrent.com/protocol.html

[3]
Additional information on the BitTorrent Protocol
http://wiki.theory.org/BitTorrentSpecification

[4]
Multitracker specification
http://home.elp.rr.com/tur/multitracker-spec.txt

[5]
Routing Algorithms for DHTs: Some Open Questions
http://www.cs.rice.edu/Conferences/IPTPS02/174.pdf

[6]
BitTorrent End to End Encryption and Bandwidth Throttling - Part I
http://www.slyck.com/news.php?story=1083