14

Protocol Measurements in BitTorrent Environments

Răzvan Deaconescu
University POLITEHNICA of Bucharest
Romania

1. Introduction

Based on BitTorrent's success story (it has managed to become the number one protocol of the internet in a matter of years), the scientific community has delved heavily in analysing, understanding and improving its performance. Research focus has ranged from measurements (Pouwelse et al. (2004)) to protocol improvements (Tian et al. (2007)), from social networking (Pouwelse et al. (2008)) to moderation techniques (Pouwelse et al. (2005)), from content distribution enhancements (Vlavianos et al. (2006)) to network infrastructure impact (Das & Kangasharju (2006)).

The BitTorrent protocol currently (October, 2011[1]) accounts for one of the largest percentages of the Internet traffic. Designed to provoke peers to give more in order to get more, the BitTorrent protocol is a prime choice for large data distribution. BitTorrent's design relies on several key elements:

- the use of a particular form of *tit-for-tat* that prevents free riding[2] and stimulates peers' selflessness;
- *optimistic unchoke* lets a peer periodically allow a connection from another peer in order to exchange content; (*choking* is the operation by which a peer closes its connection to another peer);
- *rarest piece first* aims to distribute all pieces among peers and increase availability.

In this chapter we present a novel approach involving client-side information collection regarding client and protocol implementation. We have instrumented a libtorrent-rasterbar client[3] and a Tribler[4] client to provide verbose information regarding BitTorrent protocol implementation. These results are collected and subsequently processed and analysed through a rendering interface.

The aim is to measure and analyze protocol messages while in real-world environments. To achieve this aim, a virtualized infrastructure had been used for realistic environments; apart

[1] `http://www.sandvine.com/news/global_broadband_trends.asp`
[2] though this can be circumvented as shown in BitThief (Locher et al. (2006))
[3] http://www.rasterbar.com/products/libtorrent/
[4] http://www.tribler.org/trac/

from it, clients and trackers running in a real-world swarm have been used and instrumented to provide valuable protocol information and parameters. No simulators (see Naicken et al. (2007)) have been used for collecting, measuring and analyzing protocol parameters, rather a "keep it real as much as possible" approach. Information, messages and parameters are collected directly from peers and trackers that are part of a real-world Peer-to-Peer swarm.

An approach to providing a unified model for collecting information would be a standard for developing a logging implementation for various clients, such that if using a common easy to parse output format, all information about messages exchanged between the active participants can be centralized and followed by new improvements. Such an approach would ensure maximum flexibility, albeit at the cost of having to update all clients in use, which is why we have focused on the above on an approach of collecting and analysing logging information directly provided by BitTorrent clients.

The action chronology for measuring parameters had been collecting data, parsing and storing it and then subjecting protocol parameters to processing and analysis. The rest of this chapter presents the measured parameters, approaches to collecting, parsing and storing information into an "easy to be used" format and then putting it to analysis and interpretation.

2. BitTorrent messages and parameters

Analysis of BitTorrent client-centric behavior and, to some extent, swarm behavior, is based on BitTorrent protocol messages[5]. Messages are used for handshaking, closing the connection, requesting and receiving data.

The BitTorrent client will generate at startup a unique identifier of itself known as *peer id*. This is client dependent, each client encoding a peer id based on its own implementation.

2.1 Protocol messages

Each torrent is exclusively identified by a 20-byte SHA1 hash of the value of the info key from the torrent file dictionary which is defined as *info hash*. The peer id and info hash values are important in the TCP connection establishment and are typically logged by trackers.

The handshake is the first sent message. It uses the format:

```
<length><protocol><reserved><info_hash><peer_id>
```

The protocol parameter represents the protocol identifier string and the length parameter represents the protocol name length. Reserved represents eight reserved bytes whose bits can be used to modify the behavior of the protocol. Standard implementations use this as zero-filled. `info_hash` represents the identifier of the shared resource that is requested by the initiator of the connection. `peer_id` represents the initiator's unique identifier.

The receiver of the handshake must verify the `info_hash` in order to decide if it can serve it. If it is not currently serving, it will drop the connection. Otherwise, the receiver will send its own handshake message to the initiator of the connection. If the initiator receives

[5] http://www.bittorrent.org/beps/bep_0003.html

a handshake whose `peer_id` does not match with the expected one – it must keep a list with peers addresses and ports and their corresponding `peer_id`'s – then it also must drop the connection.

Remaining protocol messages use the format:

```
<length><message ID><payload>
```

The length prefix is a four byte big-endian value representing the sum of message ID and payload sizes. The message ID is a single decimal byte. The payload is message dependent.

- **keep-alive** (`<len=0000>`)
  The Keep-alive message is the only message without any message ID and payload. It is sent to maintain the connection alive if no other message has been sent for a given amount of time. The amount of time is about two minutes.
- **choke** (`<len=0001><id=0>`)
  The Choke message is sent when the client wants to choke a remote peer.
- **unchoke** (`<len=0001><id=1>`)
  The Unchoke message is sent when the client wants to unchoke a remote peer.
- **interested** (`<len=0001><id=2>`)
  The Interested message is sent when the client is interested in something that the remote peer has to offer.
- **not interested** (`<len=0001><id=2>`)
  The Not interested message is sent when the client is not interested in anything that the remote peer has to offer.
- **have** (`<len=0005><id=4><piece index>`)
  The piece index is a 4 bytes value representing the zero-based index of a piece that has just been successfully downloaded and verified via its hash value present in the torrent file.
- **bitfield** (`<len=0001+X><id=5><bitfield>`)
  The bitfield message may only be sent immediately after the handshake sequence has occurred and before any other message is sent. It is optional and need not be sent if a client has no pieces. The Bitfield payload has the length X and its bits represent the pieces that have been successfully downloaded. The high bit in the first byte corresponds to piece index 0. A set bit indicates a valid and available piece, and a cleared bit indicates a missing piece. Any spare bits are set to zero.
- **request** (`<len=0013><id=6><index><begin><length>`)
  The Request message is sent when requesting a block. Index is the zero-based index of the piece containing the requested block, begin is the block offset inside the piece and length represents the block size.
- **piece** (`<len=0009+X><id=7><index><begin><block>`)
  The Piece message is sent when delivering a block to an interested peer. Index is the zero-based index of the piece containing the delivered block, begin is the block offset inside the piece and block represents the X-sized block data.

- **cancel** (<len=0013><id=8><index><begin><length>)
  The Cancel message is sent when canceling a block request sent before. Index is the zero-based index of the piece containing the requested block, begin is the block offset inside the piece and length represents the block size.
- **port** (<len=0003><id=9><listen port>)
  The Port message is sent by clients that implement a DHT tracker. The listen port is the port of the client's DHT node listening on.

Swarm measured data is usually collected from trackers. While this offers a global view of the swarm it has little information about client-centric properties such as protocol implementation, neighbour set, number of connected peers, etc. A more thorough approach (Iosup et al. (2006)) uses network probes to interrogate various clients.

Our approach, while not as scalable as the above mentioned one, aims to collect client-centric data, store and analyse it in order to provide information on the impact of network topology, protocol implementation and peer characteristics. Our infrastructure provides micro-analysis, rather than macro-analysis of a given swarm. We focus on detailed peer-centric properties, rather than less-detailed global, tracker-centric information. The data provided by controlled instrumented peers in a given swarm is retrieved, parsed and stored for subsequent analysis.

We differentiate between two kinds of BitTorrent messages: *status messages*, which clients provide periodically to report the current session's download state, and *verbose messages* that contain protocol messages exchanged between peers (chokes, unchokes, peer connections, pieces transfer etc.).

Another type of messages are those provided by tracker logging. Tracker-based messages provide an overall view of the entire swarm, albeit at the cost of less-detailed information. Tracker logging typically consists of periodic messages sent by clients as announce messages. However, these messages' period is quite large (usually 30 minutes – 1800 seconds) resulting in less detailed information. Their overall swarm vision is an important addition to status and verbose client messages.

2.2 Measured data and parameters

Data and parameters measured are those particular to BitTorrent clients and swarms, that provide support for evaluation and improvements at protocol level. The measured parameters are described in the Table 1, Table 2 and Table 3, depending on their source (either status messages, verbose messages or tracker messages).

2.3 Approaches to collecting and extracting protocol parameters

Peer-to-Peer clients and applications may be instrumented to provide various internal information that is available for analysis. This information may also be provided by client logging enabled for the client. Such data features parameters describing client behavior, protocol messages, topology updates and even details on internal algorithms and decisions.

We "aggregate" this information as messages and focus on protocol messages, that is messages regarding the status of the communication (such as download speed, upload speed) and those with insight on protocol internals (requests, acknowledgements, connects, disconnects).

Parameter	Explanation
Download speed	Current peer download speed – number of bytes received
Upload speed	Current peer upload speed – number of bytes sent
ETA	How long before the complete file is received
Number of connections	Number of remote peers currently connected to this client
Download size	Bytes download so far
Upload size	Bytes uploaded so far
Remote peers ID	IP address and TCP port of remote peers
Per-remote peer download speed	Download speed of each remote connected peer
Per-peer upload speed	Upload speed of each remote connected peer

Table 1. Parameters from Status Messages

Parameter	Explanation
`CHOKE`	Disallow remote peer to request pieces
`UNCHOKE`	Allow remote peer to request pieces
`INTERESTED`	Mark interest in a certain piece
`NOT_INTERESTED`	Unmark interest in a certain piece
`HAVE`	Remote peer possesses current piece
`BITFIELD`	Bitmap of the file
`REQUEST`	Ask for a given piece
`PIECE`	Send piece
`CANCEL`	Cancel request of a piece
`DHT_PORT`	Present DHT port to DHT-enabled peers

Table 2. Parameters from Verbose Messages

Parameter	Explanation
Swarm size	The number of peers in the swarm
Client IP/port	Remote peer identification (IP address and TCP port in used)
Client type	BitTorrent implementation of each client
Per-client download size	Download size for each client
Per-client upload size	Upload size for each client

Table 3. Parameters from Tracker Messages

As such, there is a separation between periodic, status reporting messages and internal protocol messages that mostly related to non-periodic events in the way the protocol works. These have been "dubbed" *status messages* and *verbose messages*.

Status messages are periodic messages reporting session state. Messages are usually output by clients at every second with updated information regarding number of connected peers, current download speed, upload speed, estimated time of arrival, download percentage, etc. Status messages are to be used for real time analysis of peer behaviour as they are lightweight and periodically output (usually every second).

Status messages may also be used for monitoring, due to their periodic arrival. When using logging, status messages are typically provided as one line in a log file and parsed to provide

valued information. Graphical evolution and comparison of various parameters result easily from processing status messages log files.

Verbose messages or *log messages* provide a thorough inspection of a client's implementation. The output is usually of large quantity (hundreds of MB per client for a one-day session). Verbose information is usually stored in client side log files and is subsequently parsed and stored.

Verbose information may not be easily monitored due to its event-based creation. When considering the BitTorrent protocol, verbose messages are closely related to BitTorrent specification messages such as `CHOKE`, `UNCHOKE`, `REQUEST`, `HAVE` or internal events in the implementation. Verbose information may be logged through instrumentation of client implementation or activation of certain variables. It may also be determined through investigation of network traffic.

Apart from protocol information provided in status and verbose messages, one may also collect information regarding application behavior such as the piece picking algorithm, size of buffers used, overhead information. This data may be used to fulfill the image of the overall behavior and provide insight on possible enhancements and improvements.

There are various approaches to collecting information from running clients, depending on the level of intrusiveness. Some approaches may provide high detail information, while requiring access to the client source code, while others provide general information but limited intrusiveness.

The most intrusive approach requires placing hook points into the application code for providing information. This information may be sent to a monitoring service, logged, or sent to a logging library. Within the P2P-Next project[6], for example, the NextShare core provides an internal API for providing information. This information is then collected either through a logging service that collects all information or through the use of a monitoring service with an HTTP interface and MRTG graphics rendering tools.

Another approach makes use of logging information directly provided by BitTorrent clients. There are two disadvantages to this approach. The first one is that each client provides information in its own way and a dedicated message parser must be enabled for each application. The second one is related to receiving verbose messages. In order to be able to receive verbose messages, one has to turn on the verbose logging. This may be accomplished through a startup option, an environment variable or a compile option. It may be the case that non-open source applications possess none of these options and cannot provide requested information. This is the approach that we will focus on for the rest of the chapter.

Finally, a network-oriented approach requires a thorough analysis of network packets similar to deep packet inspection. It allows an in depth view of all packets crossing a given point. Its main advantage is ubiquity: it may be applied to all clients and implementation regardless of access to the source code. The disadvantage is the difficulty in parsing all packets and extracting required information (specific to the BitTorrent protocol) and, perhaps more pressing, the significant processing overhead introduced.

[6] `http://www.p2p-next.org`

Messages and information collected are concerned with client behavior. As such, the applications in place work at the edge of the P2P network on each client. No information is gathered from the core of the network, inner routers or the Internet. In order to provide an overall profile of the swarm or P2P network information collected from all peers must be aggregated and unified. While having only edge-based information means some data may be lacking it provides a good perspective of the protocol internals and client implementation. We dub this approach client-centric investigation.

Collected data may be either monitored, with values rendered in real time or it may also be archived and compressed for subsequent use. The first approach requires engaging parsers while data is being generated, while the other allows use of parsers subsequently. When using parsers with no monitoring, data is usually stored in a "database". "database" is a generic term which may refer to an actual database engine, file system entries, or even memory information. A rendering or interpretation engine are typically employed to analyze information in the database and provide it in a valuable form to the user.

3. Log collecting for BitTorrent peers

The log collection approach implying a less intrusive activity but providing a great deal of protocol parameters is the use of logging information from clients. Each client typically presents status information (the dubbed *status message*) consisting of periodic information such as download speed, upload speed, number of connection and, if enabled, a set of enhanced pieces of information (the dubbed *verbose message*). Types of messages and their content have been thoroughly described in Section 2. All or most of BitTorrent clients provide status messages but some sort of activation or instrumentation is required to provide verbose messages.

3.1 Using and updating BitTorrent clients for logging

Throughout experiments we have used multiple open-source clients. All of them provided basic status information, while some were updated or altered to provide verbose information as well. Transmission, Aria2, Vuze, Tribler, libtorrent-rasterbar and the mainline client had been used to provide status parameters, while Tribler and libtorrent-rasterbar had also been instrumented to provide verbose parameters.

As building the setup, deploying peers and collecting information and subjecting it to dissemination is a lengthy process, this has to be automated Deaconescu et al. (2009).

Several approaches had been put to use to collect status information, depending on the client implementation:

- The main issue with **Azureus** was the lack of a proper CLI that would enable automation. Though limited, a "Console UI" module enabled automating the tasks of running Azureus and gathering download status and logging information.
- Although a GUI oriented client, **Tribler** does offer a command line interface for automation.

- **Transmission** has a fully featured CLI and was one of the clients that were very easy to automate. Detailed debugging information regarding connections and chunk transfers can be enabled by setting the `TR_DEBUG_FD` evironment variable.
- **aria2** natively provides a CLI and it was easy to automate. Logging is also enabled through CLI arguments.
- **hrktorrent** is a lightweight implementation over **libtorrent-rasterbar** and provides the necessary interface for automating a BitTorrent transfer, albeit some minor modifications have been necessary.
- **BitTorrent Mainline** provides a CLI and logging can be enabled through minor modifications of the source code.

In order to examine BitTorrent transfer parameters at a protocol implementation level, we propose a system for storing and analysing logging data output by BitTorrent clients. It currently offers support for hrktorrent/libtorrent[7] and Tribler[8].

Our study of logging data takes into consideration two open-source BitTorrent applications: Tribler and hrktorrent[9] (based on libtorrent-rasterbar). While the latter needed minimal changes in order to provide the necessary verbose and status data, Tribler had to be modified significantly.

The process of configuring Tribler for logging output is completely automated using shell scripts and may be reversed. The source code alterations are focused on providing both status and verbose messages as client output information.

Status message information provided by Tribler includes transfer completion percentage, download and upload rates. In the modified version, it also outputs current date and time, transfer size, estimated time of arrival (ETA), number of peers, and the name and path of the transferred file.

In order to enable *verbose message* output, we took advantage of the fact that Tribler uses flags that can trigger printing to standard output for various implementation details, among which are the actions related to receiving and sending BitTorrent messages. The files we identified to be responsible for protocol data are changed using scripts in order to print the necessary information and to associate it to a timestamp and date. Since most of the protocol exchange data was passed through several levels in Tribler's class hierarchy, attention had to be paid to avoid duplicate output and to reduce file size. In contrast to libtorrent-rasterbar, which, at each transfer, creates a separate session log file for each peer, Tribler stores verbose messages in a single file. This file is passed to the verbose parser, which extracts relevant parts of the messages and writes them into the database.

Unlike Tribler, hrktorrent's instrumentation did not imply modifying its source code but defining `TORRENT_LOGGING` and `TORRENT_VERBOSE_LOGGING` macros before building (recompiling) libtorrent-rasterbar. Minor updates had to be delivered to the compile options of hrktorrent in order to enable logging output.

[7] http://www.rasterbar.com/products/libtorrent/
[8] http://www.tribler.org/trac
[9] http://50hz.ws/hrktorrent/

Although our system processes and stores all protocol message types, the most important messages for our swarm analysis are those related to changing a peer's state (choke/unchoke) and requesting/receiving data. Correlations between these messages are the heart of provisioning information about the peers' behaviour and BitTorrent clients' performance.

3.2 Storage

Logging information is typically stored in log files. In libtorrent-rasterbar's case, logging is using a whole folder and logging information for each remote peer is using a single file in that folder. Usually information is redirected from standard output and error towards the output file.

As in a given experiment logging information occupies a large portion of disk space, especially verbose messages, files and folders are compressed in archive files. There would generally be a log archive for each client session. When information is to be processed, logging archives are going to be provided to the data processing component. A log archive contains both status messages and verbose messages.

Logging information may be stored in archive files for subsequent use or it may be processed live – that is parsing and interpreting parameters as log files are being generated. When running a live/real-time processing component, compressing logging information may not be required. However, in order to still preserve the original files, some experimenters may choose to retain access to the log archives.

The usefulness of a live processing component is based primarily on relieving the burden of space consumption, in case archiving is disabled. Most of the logged information is not useful, due to the fact that some peers may not be connected to other peers and status information, though provided, consists of parameters that are equal to zero – no connections means `0 KB/s` download speed, `0 KB/s` upload speed and others. On a certain occasion, a log file that had been used for more than 3 weeks, occupied more than 1GB of data but resulted in just 27KB valuable information.

Either when using live parsers or subsequent analysis, parameters are parsed for rapid use. The post-parsing storage is typically a relational database. The advantage of such a storage facility is its rapid access for post processing. When inquiring about given swarm parameters, the user would query the database and rapidly obtain necessary information. If that wouldn't be enabled, each inquiry would require a new parsing activity, resulting in large overhead and CPU consumption. Database storage is the final step of the logging and parsing stage. Parameter analysis, interpretation and advising activities would not be concerned of logging information, but only query the database.

3.3 Experiments

In order to collect information specific to a swarm, one must have access to all clients and logging information from those clients. As such, either all clients are accessible to the experimenter, or users would subsequently provide logging information to the experimenter.

Some remote information may be replaced by that provided by a tracker log file. A tracker logs information regarding the overall swarm view, albeit its periodicity is quite large (typically 30 minutes – 1800 seconds).

An intermediate approach to collecting logging information is a form of aggregation of information on the client side. This information may be either sent to a logging service or stored to be subsequently provided to the user. The former approach it taken by the Logging Service withing the P2P-Next project.

Typical experiments are those that allow full control to the user and provide all information rendered by clients. Deployment, log activation, log collection/archiving and even parsing are accomplished in full automation. One would create a configuration file and run experiments. Log archive files typically result from the experiment and, after gathering all, may be subjected to analysis.

The inclusion of tracker information had been enabled in the use of UPB Linux Distro Experiments[10] as described by Bardac et al. (2009). Tracker log files are parsed live and provide overall swarm parameters. Various information from tracker log files are provided as graphic images that show the evolution of swarm parameters.

Tracker logs have also been enabled in certain experiments. These experiments rely on extensive logging information (verbose messages) provided by seeders and tracker information. The lack of complete access to the all clients in the swarm is balanced out by the usage of verbose logging on the seeders' side. However, remote peers' intercommunication is not logged in anyway. Such that a form of aggregation and collection of remote peers' intercommunication messages is still required.

3.4 Monitoring and post-processing

Log processing, as described in Section 4 refers to parsing and interpreting BitTorrent protocol parameters. Data is parsed into an easy to be accessed database that is provided to the user.

As described above, one may choose to store logging information and then enable analysis. We dub this approach *post-processing*. The other approach is for live analysis of the provided parameters, resulting in client and swarm monitoring. The two approaches may, of course, be combined: while doing parsing of information, it is also stored in a database while various parameters are also monitored.

An overview of a typical architecture for data processing is presented in Figure 4. Separate parsers are used for live parsing and classical parsing. Classical parsing results in a database "output", while live parsing results in both a database "output" and the possibility of deploying live client and swarm monitoring.

4. Protocol data processing engine

As client instrumentation provides in-depth information on client implementation, it generates extensive input for data analysis. Coupled with carefully crafted experiments and message filtering, this will allow the detection of weak spots and of improvement

[10] http://torrent.cs.pub.ro/

possibilities in current implementations. Thus it will provide feedback to client and protocol implementations and swarm "tuning" suggestions, which in turn will enable high performance swarms and rapid content delivery in peer-to-peer systems.

Due to various types of modules employed (such as parser implementations, storage types, rendering engines) a data processing framework may provide different architectures. A sample view of the infrastructure consists of the following modules:

- **Parsers** – receive log files provided by BitTorrent clients during file transfers. Due to differences between log file formats, there are separate pairs of parsers for each client. Each pair analyses status and verbose messages.
- **Database Access** – a thin layer between the database system and other modules. Provides support for storing messages, updating and reading them.
- **SQLite Database** – contains a database schema with tables designed for storing protocol messages content and peer information.
- **Rendering Engine** – consists of a GUI application that processes the information stored in the database and renders it using plots and other graphical tools.

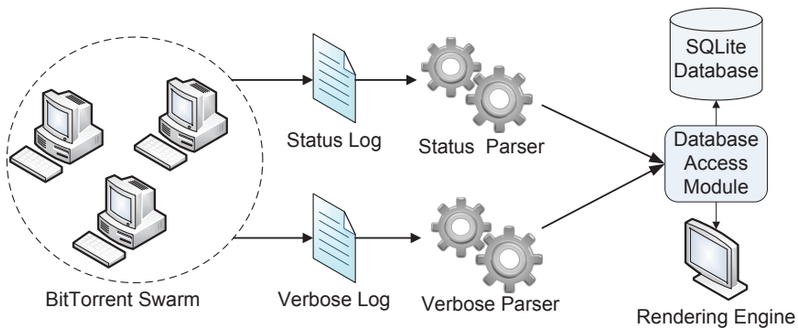


Fig. 1. Logging System Overview

As shown in Figure 1, using parsers specific to each type of logging file, messages are sent as input to the *Database Access* module that stores them into an SQLite database. In order to analyse peer behaviour, the *Rendering Engine* reads stored logging data using the *Database Access* module and outputs it to a graphical user interface.

Once all logging and verbose data from a given experiment is collected, the next step is the analysis phase. The testing infrastructure provides a GUI (*Graphical User Interface*) statistics engine for inspecting peer behaviour.

The GUI is implemented in Python using two libraries: *matplotlib* – for generating graphs and *TraitsUi* – for handling widgets. It offers several important plotting options for describing peer behaviour and peer interaction during the experiment:

- *download/upload speed* – displays the evolution of download/upload speed for the peer;
- *acceleration* – shows how fast the download/upload speed of the peer increases/decreases;
- *statistics* – displays the types and amount of verbose messages the peer exchanged with other peers.

The last two options are important as they provide valuable information about the performance of the BitTorrent client and how this performance is influenced by protocol messages exchanged by the client.

A sample GUI screenshot may be observed in Figure 2 and Figure 3:

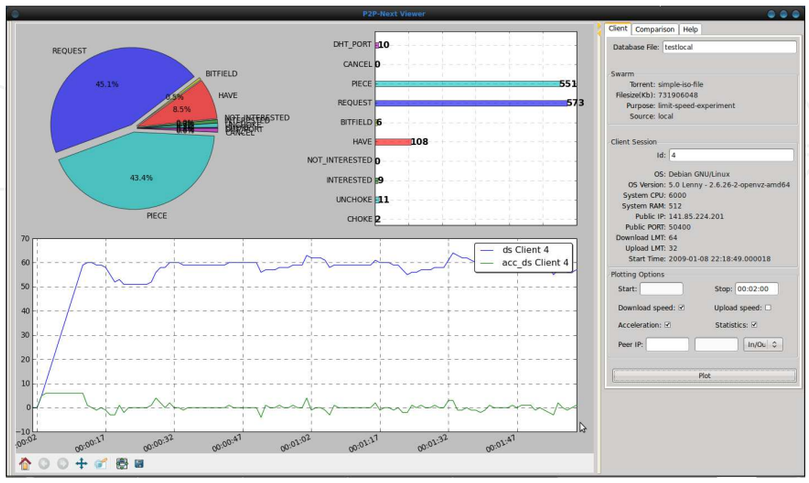


Fig. 2. Rendering Engine for BitTorrent Parameters: Client Analysis

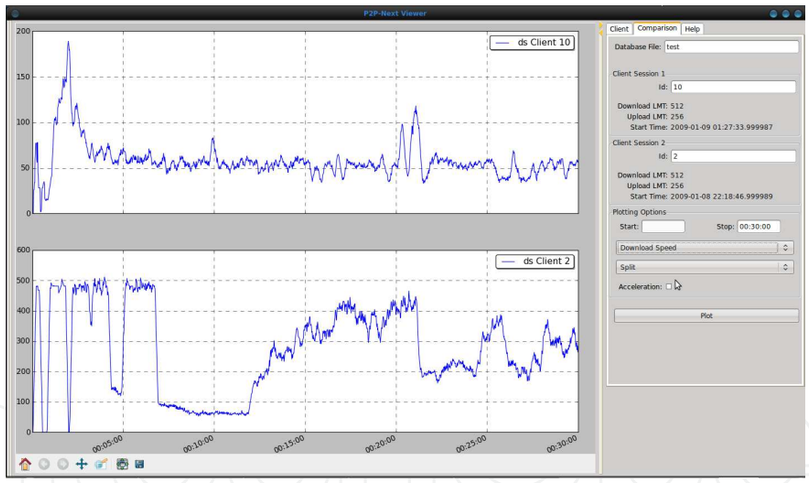


Fig. 3. Rendering Engine for BitTorrent Parameters: Client Comparison

The *acceleration* option measures how fast a BitTorrent client is able to download data. High acceleration forms a basic requirement in live streaming, as it means starting playback of a torrent file with little delay.

The *statistics* option displays the flow of protocol messages. We are interested in the choke/unchoke messages.

The GUI also offers two modes of operation: *Single Client Mode*, in which the user can follow the behaviour of a single peer during a given experiment, and *Client Comparison Mode*, allowing for comparisons between two peers.

4.1 Post processing framework for real-time log analysis

The dubbed post processing framework is used for storing logging information provided by various BitTorrent clients into a storage area (commonly a database). An architectural view of the framework is described in Figure 4.

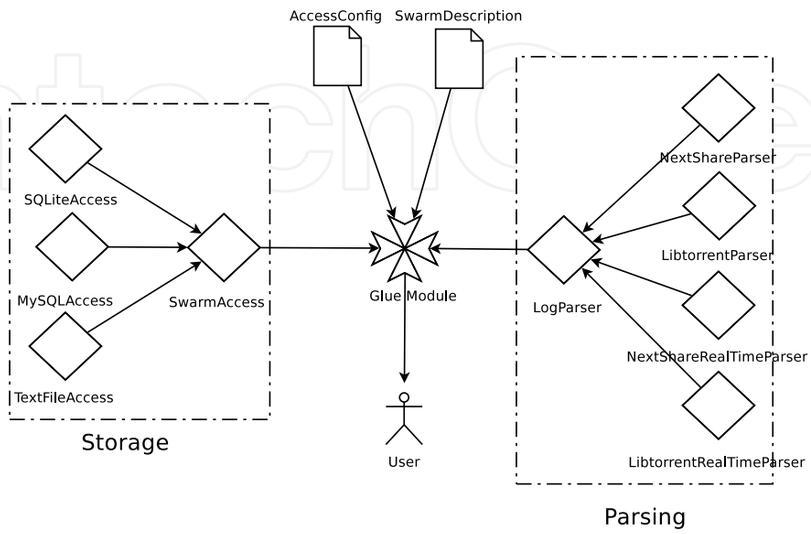


Fig. 4. Post-Processing Framework Architecture

The two main components of the framework are the parser and the storage components. Parsers process log information and extract measured protocol parameters to be subject to analysis; storers provide an interface for database or file storing – both for writing and reading. Storers thus provide an easy to access, rapid to retrieve and extensible interface to parameters. Storers are invoked when parsing messages – for storing parameters, and when analyzing parameters – for retrieving/reading/accessing parameters.

Within the parser component, a `LogParser` module provides the interface to actual parser implementations. There are two kinds of parsers: log parsers and real time log parsers. The former are used for data already collected and subsequently provided by the experimenter. Another approach involves using parsers at the same time as the client generates logging information. This real time parsing approach possesses three important advantages: monitoring may be enabled for status messages, less space is wasted as messages are parsed in real time, and processing time is reduced due to the overlapping of the parsing time and the storing time. The disadvantage of a real time parser is a more complex implementation as it has to consider the current position in the log file and continue from that point when data is available. At the same time, all clients must be able to access the same database, probably located on a single remote system.

The storage component is interfaced by the `SwarmAccess` module. This module is backed by database-specific implementations. This may be RDMBS systems such as MySQL or SQLite or file-based storage. Parameters are stored according to the schema described in Figure 5.

Configuration of the log files and clients to be parsed is found in the *SwarmDescription* file. All data regarding the current running swarm is stored in this file. Client types in the description

file also determine the parser to be used. Selection of the storage module is based on the configuration directives in the *AccessConfig* file. For an SQLite storage, this contains the path to the database file; for an MySQL file, it contains the username, password, database name specific to database connection inquiries.

The user/developer is interfaced by a *Glue Module* that provides all methods deemed necessary. The user would call methods in the Glue Module for such actions as parsing a session archive, a swarm archive, for updating a configuration file, for retrieving data that fills a given role.

4.1.1 Parameter storage database

The database schema as shown in Figure 5 is used for relational database engines such as MySQL or SQLite.

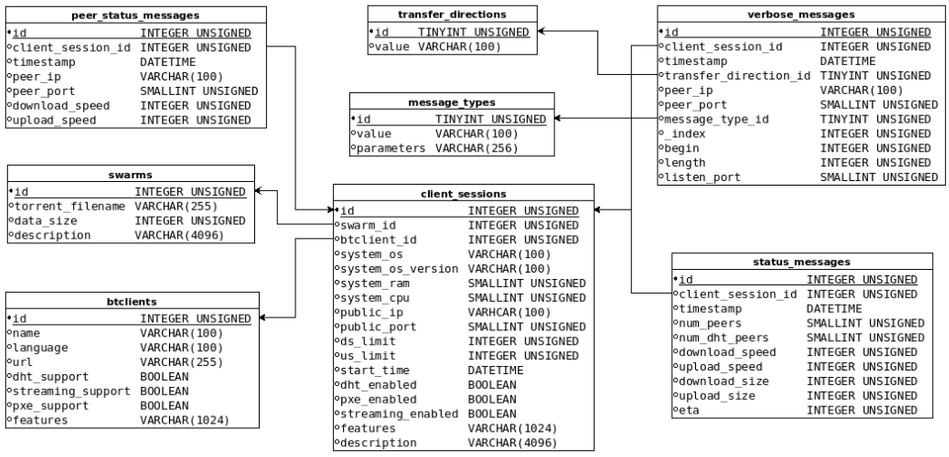


Fig. 5. Database Schema

The database schema provides the means to efficiently store and rapidly retrieve BitTorrent protocol parameters from log messages. The database is designed to store parameters about multiple swarms in the `swarms` table; each swarm is identified by the `.torrent` file its clients are using.

Information about peers/clients that are part of the swarm are stored in the `client_sessions` table. Each client is identified by its IP address and port number. Multiple pieces of information such as BitTorrent clients in use, enabled features and hardware specifics are also stored.

Three classes of messages result in three tables: `status_messages`, `peer_status_messages` and `verbose_messages`. The `peer_status_messages` table stores parameters particular to remote peers connected to the current client, while the `status_messages` stores parameters specific to the current client (such as download speed, upload speed and others). Each line in the `*_messages` tables points to an entry in the `client_sessions` table, identifying the peer it belongs to – the one that generated the log message.

4.1.2 Logfile-ID mapping

When parsing log files, one has to know the ID of the client session that has generated the log file. In order to automate the process, there needs to be a mapping between the log file (or log archive) and the client session ID.

At the same time, the client session ID needs to exist in the `client_sessions` table in the database, together with information such as BitTorrent client type, download speed limitation, operating system, hardware specification etc. This information needs to be supplied by the experimenter in a form that is both easy to create (by the experimenter) and parse.

A swarm description file is to be supplied by the experimenter. This file consists of all required swarm and peer information including the name/location of the log file/archive.

As we consider the INI format to be best suited for this, as it is fairly easy to create, edit and update, it was chosen to populate initial information. The experimenter may easily create an INI swarm description file and provide it to the parser together with the (compressed) log files.

The swarm description file is to be parsed by the experimenter and SQL queries will populate the database. One entry would go into the `swarms` table and a number of entries equal to the number of peers in the swarm description file would go into the `client_sessions` table. As a result of these queries, swarm IDs and client sessions IDs are going to be created when running SQL insert queries (due to the `AUTO_INCREMENT` options). This IDs are essential for the message parsing process and are going to be written down in the Logfile-ID-Mapping-File.

The swarm description file parser is going to parse that file and also generate a logfile-id mapping file. The parser is responsible for three actions:

- parsing the swarm description file
- creating and running SQL insert queries in the `swarms` and `client_sessions` tables
- create a logfile-id mapping file consisting of mappings between client session IDs and log/file

A logfile-id mapping file is to be generated by the swarm description parser and will subsequently be used by the message parser (be it status messages or verbose messages). The mapping file simply maps a client session ID to a log file or a compressed set of log files. A sample file is stored in the repository. The message parser doesn't need to know client session information; it would just use the mapping file and populate entries in the `*_messages` tables.

The message parser is going to use the logfile-id mapping file and the log file (or compressed set of log files) to populate the `*_messages` tables in the database (`status_messages`, `peer_status_messages`, `verbose_messages`.

The workflow of the entire process is highlighted in Figure 6.

There is a separation between the *experimenter* – the one running trials and collecting information and the *parser* – the one interpreting the information.

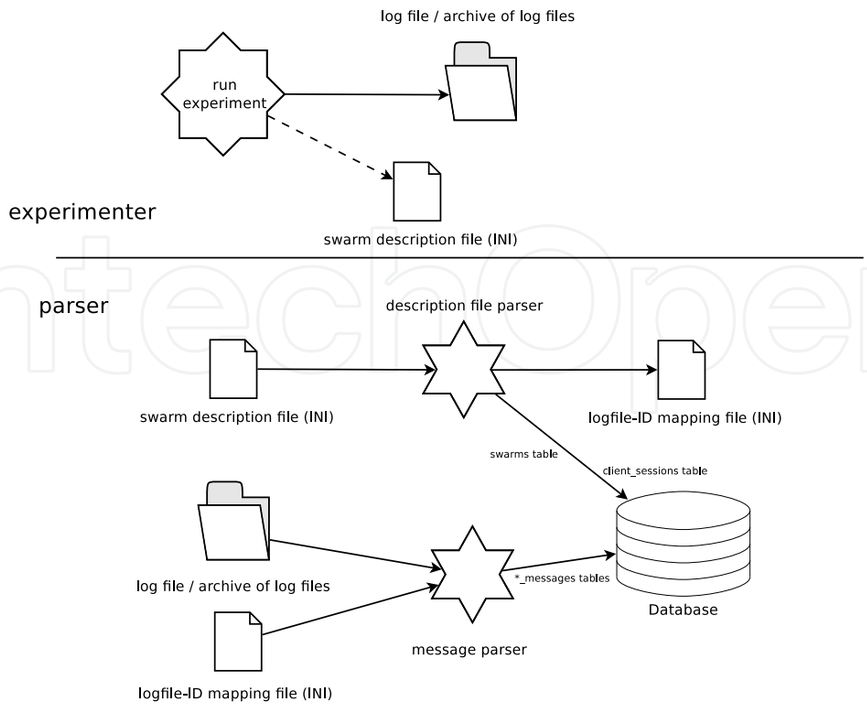


Fig. 6. Workflow of Log Parsing Considering ID Mapping

Trials are run and the experimenter provides a log file or set of log files or archive of log files (the data) and a swarm description file (INI format) consisting of characteristics of clients in the swarm, the file used and the swarm itself (the metadata).

The swarm description file is used to provide an intermediary logfile-id mapping file, as described above. This file may be provided as a file system entry (typically INI), as an in-memory information or it may augment the existing swarm description file (only the client session ID needs to be added).

The logfile-ID mapping, the swarm description file and the log file(s) are then used by the message parser and the description parser to provide actual BitTorrent parameters to be stored in the database. The parsers would instantiate a specific storage class as required by the users and store the information there.

5. Conclusion

In order to provide thorough analysis of Peer-to-Peer protocols and applications, realistic trials and careful measurements were presented to be required. Clients and applications provide necessary parameters (such as download speed, upload speed, number of connections, protocol message types) that give an insight to the inner workings of clients and swarms.

Protocol analysis, centered around BitTorrent protocol, relies on collecting protocol parameters such as download speed, upload speed, number of connections, number of messages of a certain type, timestamp, remote peer speed, client types, remote peer IDs.

We consider two kinds of messages, dubbed *status messages* and *verbose messages* that may be extracted from clients and parsed, resulting in the required parameters.

Various approaches to collecting messages are presented, with differences in the method intrusiveness and quantity and quality of data: certain methods may require important updates to existing clients and, as such, access to the source code, while others may only need access to information provided as log files.

Collection, parsing, storage and analysis of logging information is the primary approach employed for protocol parameter measurements. A processing framework has been designed, implemented and deployed to collect and process status and verbose messages. Multiple parsers and multiple storage solutions are employed. Two types of processing may be used: post-processing, taking into account a previous collection of logging information into a log archive, and real-time processing when data may be monitored as it is parsed in real time.

Protocol parameters are presented to the user through the use of a rendering engine that provides graphical representation of parameter evolution (such as the evolution of download speed or upload speed). The rendering engine makes use of the database results from the processing framework and provides a user friendly interface to the experimenter.

6. References

Bardac, M., Milescu, G. & Deaconescu, R. (2009). Monitoring a BitTorrent Tracker for Peer-to-Peer System Analysis, *Intelligent Distributed Computing* pp. 203–208.
URL: *http://www.springerlink.com/index/r528521241850jnl.pdf*

Das, S. & Kangasharju, J. (2006). Evaluation of Network Impact of Content Distribution Mechanisms, *Proceedings of the 1st International Conference on Scalable Information Systems* pp. 35–es.

Deaconescu, R., Milescu, G., Aurelian, B., Rughinis, R. & Tapus, N. (2009). A Virtualized Infrastructure for Automated BitTorrent Performance Testing and Evaluation, *International Journal on Advances in Systems and Measurements* 2(2&3): 236–247.

Iosup, A., Garbacki, P., Pouwelse, J. & Epema, D. (2006). Correlating Topology and Path Characteristics of Overlay Networks and the Internet, *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, IEEE Computer Society, Washington, DC, USA, pp. 10–.
URL: *http://portal.acm.org/citation.cfm?id=1134822.1134925*

Locher, T., Moor, P., Schmid, S. & Wattenhofer, R. (2006). Free Riding in BitTorrent is Cheap, *Fifth Workshop on Hot Topics in Networks (HotNets-V)* .
URL: *http://www.sigcomm.org/HotNets-V/program.html*

Naicken, S., Livingston, B., Basu, A., Rodhetbhai, S., Wakeman, I. & Chalmers, D. (2007). The state of peer-to-peer simulators and simulations, *SIGCOMM Comput. Commun. Rev.* 37(2): 95–98.

Pouwelse, J. A., Garbacki, P., Epema, D. H. J. & Sips, H. J. (2005). The Bittorrent P2P File-Sharing System: Measurements And Analysis, *4th International Workshop on Peer-to-Peer Systems (IPTPS)*.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.3191*

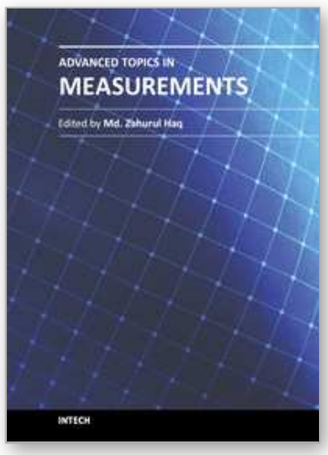
Pouwelse, J. A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D. H. J., Reinders, M., van Steen, M. R. & Sips, H. J. (2008). TRIBLER: A Social-based Peer-to-Peer

System: Research Articles, *Concurr. Comput. : Pract. Exper.* 20: 127–138.
URL: *http://portal.acm.org/citation.cfm?id=1331115.1331119*

Pouwelse, J., Garbacki, P., Epema, D. & Sips, H. (2004). A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.4761*

Tian, Y., Wu, D. & Ng, K.-W. (2007). Performance Analysis and Improvement for BitTorrent-like File Sharing Systems, *Concurrency: Practice and Experience* 19: 1811–1835.

Vlavianos, A., Iliofotou, M. & Faloutsos, M. (2006). BiToS: Enhancing BitTorrent for Supporting Streaming Applications, *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings* pp. 1–6.
URL: *http://dx.doi.org/10.1109/INFOCOM.2006.43*



Advanced Topics in Measurements

Edited by Prof. Zahurul Haq

ISBN 978-953-51-0128-4
Hard cover, 400 pages
Publisher InTech
Published online 07, March, 2012
Published in print edition March, 2012

Measurement is a multidisciplinary experimental science. Measurement systems synergistically blend science, engineering and statistical methods to provide fundamental data for research, design and development, control of processes and operations, and facilitate safe and economic performance of systems. In recent years, measuring techniques have expanded rapidly and gained maturity, through extensive research activities and hardware advancements. With individual chapters authored by eminent professionals in their respective topics, Advanced Topics in Measurements attempts to provide a comprehensive presentation and in-depth guidance on some of the key applied and advanced topics in measurements for scientists, engineers and educators.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Razvan Deaconescu (2012). Protocol Measurements in BitTorrent Environments, Advanced Topics in Measurements, Prof. Zahurul Haq (Ed.), ISBN: 978-953-51-0128-4, InTech, Available from: http://www.intechopen.com/books/advanced-topics-in-measurements/protocol-measurements-in-bittorrent-environments

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen