UPC

epsc Escola Politècnica Superior de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TU Delft

Delft University of Technology

MASTER THESIS

TITLE : Joining BitTorrent and *swift* to improve P2P transfers

DEGREE: Master of Science in Telecommunication Engineering & Management

AUTHOR: Guillem Cabrera Añón

DIRECTOR: Juan López Rubio

SUPERVISOR: Dr. Ir. Johan Pouwelse (TU Delft)

DATE: 5th July 2010

Title : Joining BitTorrent and *swift* to improve P2P transfers

Author: Guillem Cabrera Añón

Director: Juan López Rubio

Supervisor: Dr. Ir. Johan Pouwelse (TU Delft)

Date: 5th July 2010

Overview

In the last decade, peer-to-peer systems showed up as an efficient content-distribution tool through Internet. However, one of the most successful systems (called BitTorrent) lacked some key features to become the silver bullet for this purpose. Aiming to solve the detected issues in BitTorrent, *swift* was designed as a new content-centric multiparty transport protocol with swarming capabilities to efficiently disseminate different contents among a group of users.

This thesis addresses the challenge of integrating the *swift* protocol into an already existent BitTorrent client called Tribler. The design and the implementation of a novel module to use the already developed Libswift library for the BitTorrent client is presented. After that, several experiments were run to assess the operation and the bandwidth performance of the solution. From the obtained results, the future tasks needed to improve the solution and the *swift* cooperation with other protocols are highlighted.

This project was carried out in the Tribler team of the Parallel and Distributed Systems research group, part of the Software Technology department in the Faculty of Electrical Engineering, Mathematics and Computer Science of the Delft University of Technology (The Netherlands).

Títol : Joining BitTorrent and *swift* to improve P2P transfers

Autor: Guillem Cabrera Añón

Director: Juan López Rubio

Supervisor: Dr. Ir. Johan Pouwelse (TU Delft)

Data: 5 de juliol de 2010

Resum

En l'última dècada, els sistemes peer-to-peer han demostrat ser una eina molt eficient per a la distribució de continguts a través d'Internet. Tot i així, un dels sistemes més exitosos en aquest camp (anomenat BitTorrent) no s'adapta perfectament a aquest propòsit i li manquen algunes funcionalitats imprescindibles. Intentant resoldre aquestes mancances, es va dissenyar *swift* com a nou protocol de transport amb l'habilitat de distribuir continguts de manera eficient entre un grup nombrós d'usuaris.

Aquest treball s'enfrenta al repte d'integrar el protocol *swift* a un client de BitTorrent anomenat Tribler. Es presenta el disseny i la implementació d'un nou mòdul per a utilitzar la llibreria Libswift des de Tribler. Tot seguit, es mostren els experiments realitzats per a avaluar el funcionament de la solució implementada i esbrinar el seu rendiment en termes d'ample de banda utilitzat. A partir dels resultats obtinguts, es tracen les línies de treball a seguir per a millorar el rendiment de la solució i adaptar *swift* al funcionament amb altres sistemes d'intercanvi de continguts.

El projecte ha sigut desenvolupat amb l'equip de treball de Tribler, dins del grup de recerca Parallel and Distributed Systems, que és part del departament de Software Technology de la Faculty of Electrical Engineering, Mathematics and Computer Science de la Delft University of Technology (Holanda).

I would like to publicly thank Dr. Johan Pouwelse for giving me the opportunity to perform this work in the Parallel and Distributed Systems group and supervising my progress during my stay at Delft University of Technology; Dr. Victor Grishchenko and Riccardo Petrocco for their support, guidance, advise and ideas during the development of the project; Dr. Arno Bakker and Boudewijn Schoon for their technical advise; and Juan López Rubio for supervising and grading this Master Thesis.

CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

During the last decade, Internet had an exponential growth in terms of users and traffic. End-users also changed their habits, moving some of the common activities (such as reading the newspaper, watching TV or making phone calls) to Internet. This caused the traffic to significantly increase and made the content-generators to need larger infrastructures to offer their services.

While Internet traffic was growing, peer-to-peer applications increased their popularity and contributed to this rise. Most of the well-known P2P applications are meant to be used for file transfers and often unfairly related with piracy issues. However, these systems also evolved to be ready for different use cases: video-telephony (with Skype[1] as the clearest exponent), music on demand applications (Spotify[2]), live TV channels on streaming (Joost[3] or PPLive[4]), peer-assisted generic CDNs[5] (BitTorrent DNA[6] or Pando Content Delivery Cloud[7]) or even CPU time sharing for extraterrestrial intelligence searches (the SETI@home program[8]).

Regarding the classical client-server paradigm, the server capacity of a service must increase accordingly to the number of users of the server. Linked with the user and traffic growth, this schema becomes no longer valid for content mass-distribution via Internet: some services would need either extremely expensive or so large than impossible to manage infrastructures. These platforms also suffer from scalability and the single point-of-failure issues. For those cases where scalability and reliability are crucial and the number of users is reasonably large, P2P systems could become a much more than a feasible solution, though they could lack of some must-needed features.

1.1. Peer-to-Peer systems

P2P systems break the client-server paradigm, building architectures in which all users in the network (called *peers* from now on) implement the same functions regardless their different constraints. The basic idea is that peers offer their available resources to other peers, but they obtain in return their needed resources from other peers, all in a non-centralized way.

Peers in distributed systems create an overlay network over the existing physical networks (see Figure 1.1). Thanks to this overlay, they are able to directly communicate and share resources. This fact makes the available resources on the network increase with the number of peers, making them a flexible and scalable system. Also, the lack of central elements

[1] http://www.skype.com/
[2] http://www.spotify.com/
[3] http://www.joost.com/
[4] http://www.pplive.com/en/
[5] Content Delivery Network
[6] http://www.bittorrent.com/dna/
[7] http://www.pandonetworks.com/
[8] http://setiathome.berkeley.edu/

in the architecture mitigates the single point-of-failure problem.

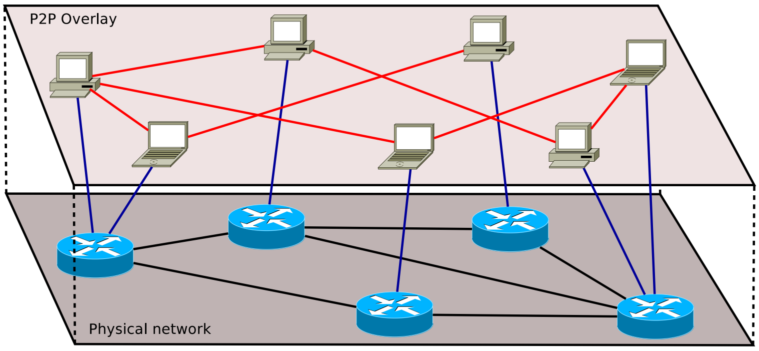


Figure 1.1: Physical networks and P2P overlay layering example. Notice the difference between physical networks and P2P overlays.

In terms of scalability, ideally more peers means more available resources, but also some overhead on communications (since a bigger overlay needs to be maintained). Thus, most of the current P2P systems either depend on some central architecture (trackers or super-peer networks) or become unstable under certain conditions. Task distribution amongst peers has also some drawbacks: since no central component exists, peer discovering and resource searches become very hard to perform actions.

As stated in [1], one of the latest and largest Internet traffic studies, P2P still produces most Internet traffic worldwide, although its proportion has declined in favor of file hosting (such as Megaupload[9] or Rapidshare[10]) and media streaming (like YouTube or Vimeo[11]) services. However, its traffic was still from the 43% of the total in North Africa up to the 70% in Eastern Europe, as Figure 1.2 shows.

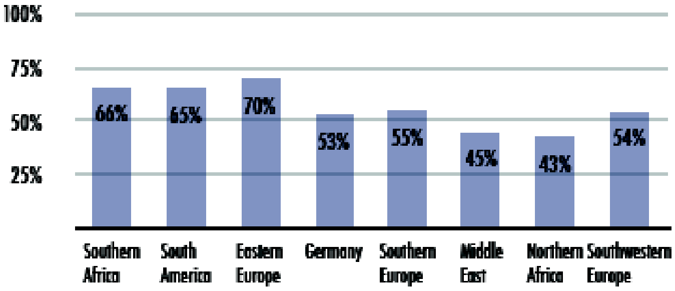


Figure 1.2: Relative P2P traffic volume in all studied regions. Source: Ipoque, Internet Study 2008/2009 [1].

Even though in the last year P2P traffic yield in front of other old-fashioned applications, the P2P systems could still be very useful to offer the new growing services. Nevertheless, this decline could become a challenge to keep working on improving P2P systems and make them better to cover file distribution and video streaming cases.

[9] `http://www.megaupload.com/`
[10] `http://www.rapidshare.com/`
[11] `http://www.vimeo.com/`

1.2. Document layout

After this introduction, the thesis report is structured as follows. Chapter 2 describes all the concepts related to the intended work of this thesis.

After that, Chapter 3 presents the innovation and research questions to solve during the development of this project.

Chapter 4 explains the adopted software solution and the techniques performed for *swift* integration into Tribler.

Chapter 5 describes the experiments run to evaluate the implemented solution and shows the obtained results.

Finally, Chapter 6 presents the result conclusions of this work, a proposal of future tasks to improve the obtained results, an environmental impact study and some final personal conclusions of the project.

CHAPTER 2. BACKGROUND

This chapter is an overview over the technologies involved and considered during the development of this Master Thesis.

First, a brief overview over the existing BitTorrent protocol is done. After that, the internal matters of the new transport protocol called *swift* is presented, pointing to its key features and describing its operation. Finally, a brief overview over the Tribler BitTorrent client and its characteristic features is included.

2.1. The BitTorrent protocol

BitTorrent is one of the most successful peer-to-peer systems, used by 160 million users worldwide [2]. The protocol [3] was designed by Bram Cohen in 2003 and its traffic covers more than half of the total P2P traffic in Internet, as shown in [1].

Although many file sharing systems were deployed in the last decade, BitTorrent showed to be one of the most stable, reliable and robust ones. Due to this fact, it has been pointed as one of the best mass dissemination tools for multimedia contents or files through Internet. Good examples are the main television or radio broadcasters looking at BitTorrent as a perfect infrastructure cost-reducing technology for content distribution and Canonical Ltd.[1] using BitTorrent to spread the Ubuntu Linux distribution.

BitTorrent splits any byte-content into fix-length pieces (of 2^N bytes long) and includes a hash of all pieces in a metadata file (usually called the .torrent file), which also includes a list of tracker URLs[2]. This file is distributed to end-users by external means of the BitTorrent protocol, such as dedicated websites (The Pirate Bay[3], Mininova[4] and isoHunt[5] are the most famous ones) or file repositories.

Once the user obtains the .torrent file describing one file (or set of files), the peer contacts one or more trackers in the contained list. Trackers maintain a list of the active peers sharing that content (also known as swarm) and the available pieces each peer has. The user then obtains a set of peers in the swarm and the pieces they have, so they could be directly contacted in order to start bartering for pieces. Figure 2.1 shows a diagram on how the BitTorrent protocol proceeds on the sharing process.

Peers in a swarm exchange content pieces using mechanisms that prevent free-riding and give incentives for sharing. Although the original BitTorrent was designed to operate a tit-for-tat [4] strategy on piece bartering [5], a late study by Levin *et al.* proposed in [6] an auction-based model which proved to be more accurate for the real BitTorrent operation.

[1] `http://www.canonical.com/`

[2] Universal Resource Location

[3] `http://thepiratebay.org/`

[4] `http://www.mininova.org/`

[5] `http://isohunt.com/`

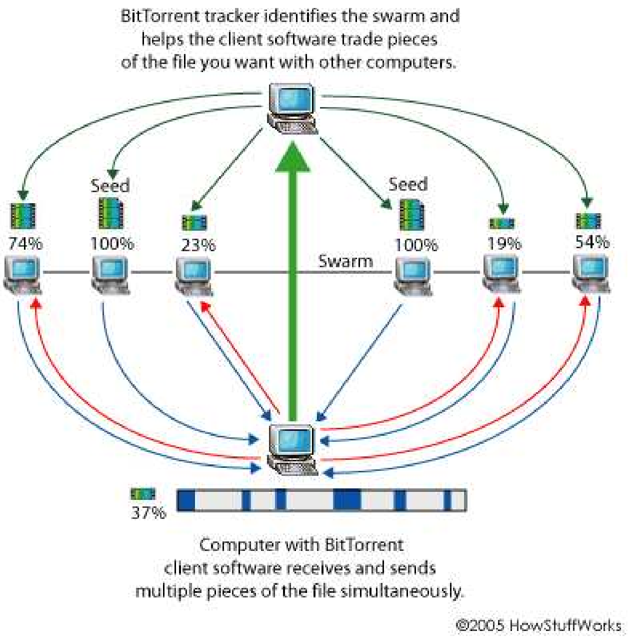

BitTorrent tracker identifies the swarm and helps the client software trade pieces of the file you want with other computers.
Seed
Seed
74%
100%
23%
100%
19%
54%
Swarm
37%
Computer with BitTorrent client software receives and sends multiple pieces of the file simultaneously.
©2005 HowStuffWorks

Figure 2.1: BitTorrent's P2P download process. Source: HowStuffWorks, How BitTorrent Works.

Once a peer joins the swarm by contacting other peers, some pieces are obtained for free, thanks to a process called optimistic unchocking. By default all peers are chocked, so no pieces are sent to them. According to the available pieces and following a rarest-first strategy, a peer unchockes other peers and start sending them pieces, expecting to receive other ones in exchange. Peers having the whole set of pieces in a swarm are known as *seeders* and peers only having part of the peers are known as *leechers*.

2.1.1. BitTorrent weak features

Even though the BitTorrent protocol is promising in terms of mass content distribution and cost-saving infrastructure, several problems made the protocol not to fit in all use cases. Some of the key issues preventing BitTorrent from becoming the silver bullet for Internet content distribution are detailed in the following paragraphs.

The first problem is BitTorrent not being a fully-distributed protocol: .torrent files need to be distributed through external channels (usually dedicated websites) and then peers need to contact a tracker to join a swarm. These two processes are usually centralized in some servers, making the infrastructure vulnerable to scalability and the single point-of-failure issues. Although several initiatives have been proposed to solve the .torrent distribution (the BuddyCast protocol [7] in Tribler) and the tracker decentralization (DHT[6] BitTorrent extension [8]), they are not widely adopted in all client implementations.

Next key issue is the long warm-up time for data transmissions in the protocol. Once a peer obtains the .torrent file, it has to contact a tracker, obtain a list of peers in the

[6] Distributed Hash Table

swarm and contact them. Then, the piece bartering process starts with other peers, which usually takes some time to reach high download rates. This is not really important for file downloads, but it has a very bad effect on the user experience for video or audio on demand or streaming cases, since the user has to wait until media starts playing.

For big file distributions, either .torrent files must contain a very long list of smaller piece hashes or a shorter list of larger piece hashes. While the problem in the first case is the larger metadata file needed to bootstrap the transfer, the second case has a direct effect on the piece bartering process: those new *leechers* having no pieces to trade with need longer times to obtain the first complete pieces, so they must wait for optimistic unchocking from contacted peers for longer. This causes even longer warm-up times, followed by the unpleasant effects previously explained.

In BitTorrent, data transfer on the network is actually done by the TCP[7] protocol. This protocol multi-layering leads to the situation where each layer uses its requesting and acknowledging mechanisms at the same time, so complexity and message overhead is unnecessarily added. BitTorrent retrieves and acknowledges pieces (split in smaller chunks), but TCP at the same time splits these chunks in segments that are also acknowledged. Thus, same data is split in all layers of the stack, so the sender and the receiver need to perform extra construction and deconstruction tasks. Moreover, this makes some bandwidth to be wasted by the additional headers included, therefore end-users can not take advantage of all the available bandwidth in their connections.

Finally, bandwidth management in BitTorrent is not user-friendly: usually BitTorrent traffic interferes with other user traffic, slowing down applications like web browsing or email reading. If the users have the feeling other applications are not running fine while their BitTorrent clients are working, they will close the client and will stop seeding content.

Due to BitTorrent's decentralization nature, the more users in a swarm the better the performance is (in terms of download rates), so traffic priority mechanisms must be applied to make BitTorrent traffic less aggressive in front of others and the user is happy running it while using other applications. However, traffic priority features are not common in end-user Internet connection devices, so other procedures should be studied: available-bandwidth discovering techniques could be applied on the user's up-link to make BitTorrent to use only the not used bandwidth. Nevertheless, the use of TCP as the transport protocol and its fixed congestion control algorithms limits possible actions on this way.

2.1.2. The μ Transport Protocol

In an attempt to address some of the above issues, the company behind the development of the official BitTorrent client, BitTorrent Inc.[8], redesigning the protocol and created μTP[9] [9][10][11], a protocol aiming to replace the legacy BitTorrent communications.

μTP relies on a new specific framing scheme for message transmission and a new con-

[7] Transmission Control Protocol

[8] `http://www.bittorrent.com/`

[9] μTransport Protocol or Micro Transport Protocol

gestion control algorithm. This new protocol lays over UDP[10] datagrams, avoiding the restrictions fixed by TCP on legacy BitTorrent and performing some of its features at the application layer. The main differences of this new packet scheme are the presence of timestamps in messages and selective acknowledgements. Timestamps are used by the congestion control algorithm and selective acknowledgements allows the protocol to notify out-of-order packet reception. The new congestion control algorithm, known as LEDBAT[11] (see below for more details), tries to be less aggressive on bandwidth usage [12][13].

Currently, the μTP protocol is only fully implemented by the μTorrent client and partially by Libtorrent.

2.1.2.1. Low Extra Delay Background Transport

LEDBAT is an alternative experimental congestion control algorithm [14] based on a so called congestion window to control the network data transmission. LEDBAT enables an advanced networking application to minimize extra delay it induces while saturating the link bottleneck by an end-to-end version of a bandwidth-scavenger service. LEDBAT was designed by Stanislav Shalunov (from BitTorrent Inc.) to be included as the default congestion control algorithm in BitTorrent DNA and as an experimental deployment in the μTorrent client.

Most of TCP congestion control algorithms are based on losses to detect congestion on the network and then back off. LEDBAT tries to be smarter, predicting losses by measuring the connection delay and backing off before the losses actually occur. Considering most of up-link home connections contain enough buffer to get delays up to several seconds and a drop-tail FIFO[12] queuing discipline with no AQM[13], a loss-based congestion control algorithm fills the whole buffer and causes the output delay to be maximum. The main drawback of this situation is interactive applications (such as VoIP[14] or on-line gaming) stop working properly and even web browsing turns to be very slow.

LEDBAT first design goal was to keep the latency across the congested link low even if it was saturated, so heavy-bandwidth consuming applications (such as peer-to-peer clients) do not disturb the user. This is done measuring the output queuing delay and throttling output data rate depending on this measured value, making LEDBAT connections yield quickly when competing with other traffic. The challenge is to keep the one-way delay low (usually below 100 ms), but non-zero, since it would mean no data was actually being sent. Nonetheless, measuring low delay values is a hard to achieve task, so although the best case would be to aim for 25 ms, current implementations take the maximum 100 ms value as the target delay. This value is high enough to be used with current operating system clock features ans limits the used bandwidth before the user notices a service degradation.

After defining a target delay low enough not to interfere on other traffic, one way delay measurements must be performed on the up-link: the sender adds a time-stamp value

[10] User Datagram Protocol

[11] Low Extra Delay Background Transport

[12] First In First Out

[13] Active Queue Management

[14] Voice over IP

in all outgoing packets and the receiver calculates the difference between this time and the local receiving time. The result is then attached to the acknowledgement message. The sender then saves the received value as the current delay and sets the base delay as the minimum value between the previous base delays (the algorithm should save more than 2 but less than 10 previous values, representing the minimum value for a minute of transmission) and the current delay. The queuing delay can now be computed as the difference between the current delay and the base delay and after that used to modify the congestion window size.

The receiver should execute the following algorithm for LEDBAT congestion control:

```
on incoming data package:
   remote_time-stamp = packet.time-stamp
   ack.delay = local.time-stamp() - remote_time-stamp
   ack.send()
```

The sender, who is actually the one controlling the congestion, should operate as follows:

```
on incoming acknowledgement:
   current_delay = ack.delay
   update_base_delay(ack.delay)
   update_current_delay(ack.delay)
   queuing_delay = current_delay() - base_delay()
   off_target = TARGET - queuing_delay
   cwnd += GAIN * off_target / cwnd

update_current_delay(delay)
   #Maintain last 10 delay measurements
   del last of current_delay
   first of current_delay = delay

current_delay()
   return min(current_delay)

update_base_delay(delay)
   if(round_to_minute(now) != round_to_minute(last_rollover))
      last_rollover = now
      del last of base_delays
      first of base_delays = delay
   else
      last of base_delay = min(last of base_delay, delay)

base_delay()
   return min(base_delay)
```

In the later algorithm, `TARGET` is usually set to a value from 25 ms to 100 ms, as stated earlier; and `GAIN` must be set up to ensure a maximum ramp-up of as much as TCP.

Peer clock synchronization is not a problem for LEDBAT: clock offsets between peers are canceled when using differences between the current delays and base delay; and clock skews between peers, unless the target delay is very low (which it is usually not the case for LEDBAT), are not too bad to have an interference on the measurements. For obvious security reasons, it is stated that timestamps should be authenticated to avoid fake values, although this feature is delegated to the application layer using the congestion control algorithm.

LEDBAT should start its operation in a slow-start mode, as TCP does. For simplicity, fixed and equal size packets are recommended (although some implementations increase the packet size while the delay remains low) and each one should be separately acknowledged. Once the congestion avoidance phase is started, the delay measurement process and window size throttling take control of the transmission.

2.2. The *swift* protocol

The *swift* protocol is described in [15] to be *a new content-centric multiparty transport protocol with the mission of efficiently disseminate content among a swarm of peers*. The ultimate goal of the protocol is to abstract Internet as a big data cloud from where end-users could retrieve any type of data using unique identifiers. To some extent, it could be regarded as BitTorrent at the transport layer and was designed aiming to solve its issues stated in Section 2.1.1..

As stated by Van Jacobson *et al.* in [16], Internet is no longer used as it was designed at the beginning: instead of being used to connect scientists to remote computers, it is now responsible of disseminating content amongst up to millions of users. So then, the original protocols designed at the beginning (such as TCP, UDP or even IP[15]) are no longer valid, since the concept of dialogue between two parties disappeared. However, while Van Jacobson's PARC team[16] propose a replacement of the whole TCP/IP stack with their Content Concentric Networking framework[17], *swift* focuses on becoming a new transport protocol aiming to carry almost the whole traffic in the current Internet. The protocol aims to serve three basic use cases, which are briefly described below:

File download Usual file acquiring, getting all needed pieces to compose the whole goal file. Out of order pieces and big delays are allowed, since the file is not ready until all the pieces are correctly received. Currently, this user case is covered by web servers, file repositories or other peer-to-peer protocols (such as BitTorrent).

VoD[18] Similar to file download case, but delay and in order piece acquiring become key issues to face: the whole file must be finally retrieved, although it is played while it is received. Nowadays, mainly user-close CDNs[19] are offering this service (the best example is YouTube[20]), although some distributed options also exist (like the BitTorrent-based Vuze client[21]).

Streaming Live multimedia stream reception. Out of order pieces are allowed if they are inside of a sliding window around the playback time, but delays are not acceptable: all possible pieces must be acquired before their playback time in order to maintain a good continuity rate and offer a good user experience. These days, the distribution of this kind of content is also done from user-close CDNs.

[15] Internet Protocol

[16] Palo Alto Research Center, Networking: `http://www.parc.com/work/focus-area/networking/`

[17] `http://www.ccnx.org/`

[19] Content Delivery Network

[20] `http://www.youtube.com/`

[21] `http://www.vuze.com/`

Covering these three use cases, *swift* aims to be suitable for use in general purpose software (such as web browsers or media players) and hardware (especially multimedia set-top boxes).

swift has five main design goals, derived from the requirements for the new multiparty transport protocol:

- Embeddable kernel-ready protocol, able to work as a transport protocol on any network devices, even the ones with low-performance capabilities.
- Support real-time streaming and file download to be able to transport almost all Internet traffic, covering as much situations as possible.
- Short warm-up times, avoiding unnecessary establishment round trips and improving user experience by showing the content as soon as possible (specially needed for VoD and streaming use cases).
- Transparent NAT[22] traversal to work behind most of Internet home boxes or firewalls, since most of nowadays users are hidden by them.
- Pluggable congestion control algorithms, allowing the user to choose non-intrusive congestion control algorithms not to interfere to regular traffic (such as web browser or email traffic).

From these design goals it is easy to realize that TCP-alike protocol features were not convenient for a multiparty transport protocol: high memory footprint for every connection, not NAT-friendly and fixed congestion control algorithms. Moreover, in-order reliable traffic delivery is no longer useful, since peer-to-peer transmissions naturally support unordered transmissions and packet losses might be recovered easily from other sources. Flow control can be easily assumed by congestion control algorithm, which in turn needs to be customized and is not possible with the usual tight integration of TCP stack to the operating system.

Although *swift* is meant to be a transport protocol by itself, UDP was chosen as real transport protocol. This avoided many integration problems with current operating systems, made things easier for NAT traversal and offered a simple enough transport layer, without all unneeded features of TCP but ready to run in all current platforms.

As a transport protocol, *swift* must not include any technical metadata for the transmission, such as the .torrent files for BitTorrent. However, it was still needed to identify different transmissions and check the integrity of all transferred data. To achieve this, Merkle trees (see Section 2.2.2. for more details) and SHA1 [17] hashes are used to obtain a unique root hash for every file. This root hash is the only information used for bootstrapping a transfer and check the integrity of every single piece the transfer is split in (refer again to Section 2.2.2. for more about root hash construction).

[22] Network Address Translation

2.2.1. Datagrams and channels

One of the *swift* guidelines is to break the pipe and the in-order flow concepts in TCP transmissions. So that, the protocol was designed to work by peers exchanging independent datagrams, such as UDP does. Each datagram contains a 4-byte unique transmission identifier, carries zero or more messages and ideally neither message nor message interdependencies should span over multiple datagrams.

To distinguish different file transfers between the same pair of peers, the concept of channels is introduced. A channel could be regarded as a TCP connection, but only deals with the transfer of a single file. Channel identifiers are negotiated at the beginning of a connection, each peer announcing the identifier to be used for the input datagrams for that transfer.

2.2.2. Merkle trees

Merkle trees are a data structure containing summary information about a piece of data in a binary tree. They are often used to verify the integrity of a long data set or files, taking advantage of hash functions. In peer-to-peer networks, Merkle hashes were proposed to be used in BitTorrent as an official extension of the protocol (defined in [18]) and are a key feature in *swift*, where they are used to obtain the root hash of a file from 1024-byte pieces, a 2^{63} pieces long base-layer tree and the SHA1 cryptographic function.

To build a Merkle tree, data is split into blocks of fixed size and a tree large enough to fit all data blocks in the bottom level is built. Then, a hash function (such as SHA1 or MD5) is applied to all of the blocks and the resulting value is placed in the corresponding bottom-layer leaf of the tree. In case some of the bottom level leaves are empty, they are filled with zero value. Then, all nodes further up are hashes of the children leaves union, resulting on a final 20-byte hash. Figure 2.2 shows how a Merkle tree is built from a simple set of data blocks.

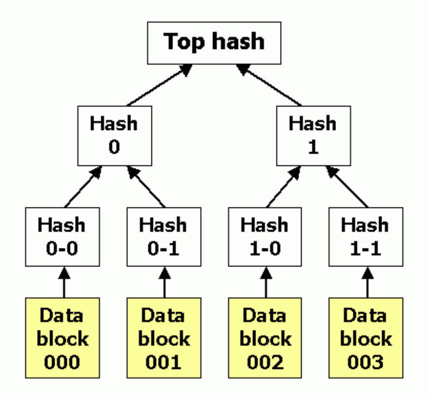


Figure 2.2: Binary Merkle hash tree construction. In this example, the value for hash0 is hash(hash0-0 + hash0-1). Source: Wikipedia, Hash Tree [19].

Hash trees can check the integrity of data blocks and therefore also the whole file or set of files. If the root hash is acquired from a trusted party, fake blocks or hashes can be easily discarded by checking the block hash against the root using the sibling and uncle hashes (shown in Figure 2.3). If a hash check fails, either the piece itself or some of the used hashes are wrong, so some actions could be taken in order to avoid retrieving new information from the malicious parties.

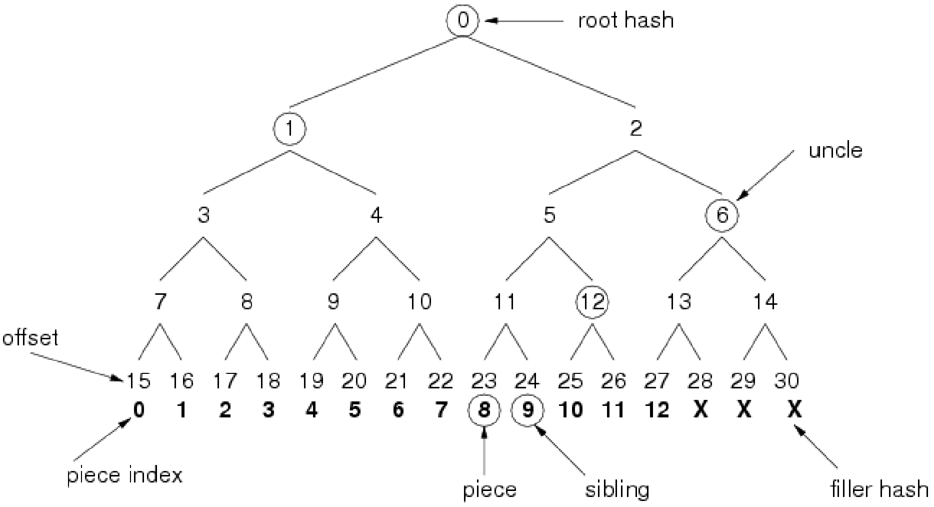


Figure 2.3: Merkle hash tree elements and integrity checking. Considering piece 8 as the piece whose integrity needs to be checked against the root hash, hashes 24 (sibling), 12 and 6 (uncles) would be required. Source: Tribler, Merkle Hashes [20].

The integrity check process is done by applying the hash function to the received piece and then building the whole tree up to the root hash (as explained in Figure 2.2). Sibling and uncle hashes are used to build this tree, as shown in Figure 2.3. Once the root hash of this tree is obtained, it is compared to the original root hash for the whole file. If both hashes are equal, the piece was correctly received.

2.2.3. Data blocks and acknowledgements

As stated earlier, data blocks in *swift* are 1024 bytes long. The choice of 1 KByte data length was motivated by the 1500 bytes limited MTU[23] and the lack of Jumbo frames support in most access networks. Considering datagrams usually carry several messages and 1024 being a 2^N number, it seems to be the a convenient choice.

All pieces from a file are set in a Merkle tree to obtain the root hash. However, this Merkle tree is also useful for piece acknowledgement purposes. Consider the binary tree and the leaf numbering shown in Figure 2.4, where only the layer-0 pieces are real data pieces and all the odd-numbered pieces represent partial hashes used in the Merkle tree.

One of the weaknesses detected in BitTorrent was the multiple acknowledgement levels used in the BitTorrent/TCP layered stack, which lead to message overheads, wasted bandwidth and unnecessary complexity. *swift* simplifies this by using binary intervals (called easily bins), whose binary tree is simple and fits perfectly with the machine integer representation and Merkle trees (see Figure 2.4 for graphical representation). Thanks to this

[23] Maximum Transmission Unit

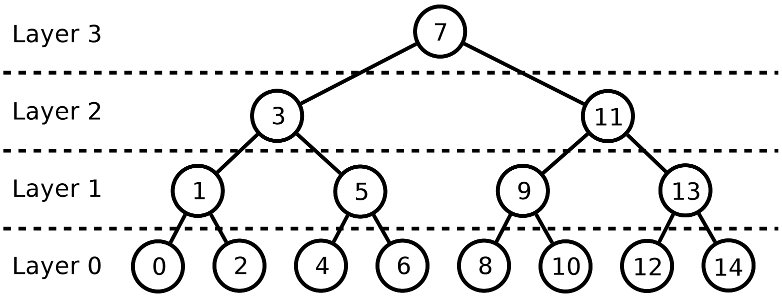


Figure 2.4: Binary interval numbering, done in the order of interval's center ascending, for an eight-piece data set. Using this numbering schema, piece 4 could be recognized by acknowledging bin 4 itself, bin 5 (which acknowledge bin 6 too), bin 3 (which also recognizes bin 1) and bin 7 (which stands for all pieces in this case). Alternatively, bins can also be numbered by the layer number and the offset position from the left of the layer. For instance, in this diagram bin 3 could also be numbered as (2,0) and bin 6 as (0,3).

easy correlation, an exponential piece acknowledgement schema was designed and a natural way to refer to leaves in the Merkle tree came up. For the acknowledgement mechanism, each piece can be acknowledged explicitly with its number or implicitly with one of the upper-layer identifiers and for the leaf referring mechanism, bin identifiers are used to label partial hashes or data block ranges in some messages of the protocol.

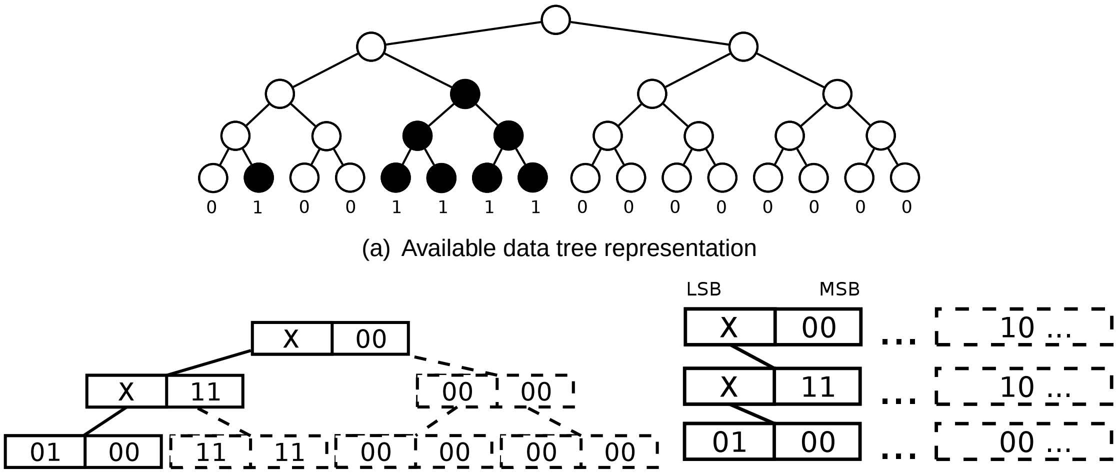
2.2.3.1. Binmaps

Once the data ordering strategy was decided, peers needed to notify others about their local data block tree status, so they can track the state of the transmission. Three classical approaches appeared to describe this state: plain bitmaps, extent lists and extent binary trees.

On one hand, bitmaps are vectors where each bit stays for a unit of data taking values 0 or 1 depending on the absence or presence of the corresponding index block within the set. Their scalability is really poor, since they have no aggregation strategies and they represent all data units, though they are very simple. On the other hand, extent lists aggregate fine but are not easy searchable. Finally, extent binary trees offer good aggregation and search features, but they become extremely large in terms of used memory.

Since both aggregation and searching features plus low memory footprint and graceful degradation were required, something else was needed for *swift*. A new data structured called binmap was designed [21]. A binmap is a hybrid of a bitmap and a binary tree, with limited length (it supports aggregation), good searching features and natural fit into the machine 32-bit integer system.

A binmap is a binary tree of cells consisting on two 16-bit halves. Each half then can be either a real bitmap standing for data or an index pointing to a lower layer cell and data is aggregated to the highest layer possible. A binmap is naturally hosted in a vector of integers. Every 16th cell hosts flag bits for the previous 30 halves (or 15 cells, so having a capacity of 2^{15} cells), which are used to distinct halves standing for real bitmaps or for deep halves (pointing to a lower layer cell). Figure 2.5 is an example of a 4-bit cell binmap representation, showing the concept of the structure operation.



(a) Available data tree representation

(b) Binmap tree representation

(c) Real binmap structure

Figure 2.5: Binmap representation. Assuming 2-bit halves and 4-bit cells, the example tree in (a) would be contained in a structure like the one shown in (b), where dashed lines and cells represent aggregated, so non-existing, parts of the tree. Finally, the real integer array layout for this case would be the one shown in (c). In the latter dashed cells denote for the last cell of each integer vector, which indicates the half stands either for real data representation (with a 0 value) or for a pointer to a lower layer cell (with a 1 value).

Finally, longer layout might be represented by extending the maximum size of a binmap: by reserving one bit of an offset-carrying half as a flag, it is possible to make halves to point to a child cell or to another vector. Thus, the capacity is extended to 2^{30} cells (from $2^{15} \times 2^{15}$).

2.2.4. Messages

The operation of *swift* is based on several messages exchanged by peers. Each type of message has a fixed length depending on the type of message and they send on the network inside of datagrams and all messages are noted with hex-like two char per byte.

The basic set of messages building the *swift* protocol is listed and described in the following lines:

HANDSHAKE This message starts a transfer of a file between two peers and announces which identifier must be used for that transmission. It has the format `00 CHANNEL_ID`, where the channel identifier is a four bytes integer which uniquely identifies a transmission. This identifier is chosen by the receiver and announced to the sender, who should use the given label for all datagrams in that transmission. Channel identifiers should be random enough to prevent easy guessing and avoid any security issue.

DATA This message carries real data from a transmission. It has the format `01 BIN_ID DATA`, where the bin identifier is a four bytes label of the carried data block and data is the actual data. Usually 1024 bytes are carried, but in some cases (mostly for the tail message of a transmission) less data can be present. Notice this is the only message that can have non-fixed length.

ACK This message is an acknowledgment for data received in the current channel. It has the format `02 BIN_ID TIMESTAMP`, where the bin identifier is a four bytes identifier of the acknowledged data and the time-stamp is a four bytes value of the local milliseconds counter of the receiver. This message must be sent to the source peer as soon as an incoming DATA message is verified and accepted.

HAVE This message is an information message announcing the peer acquired and correctly verified the given bin from another source. It has the format `03 BIN_ID`, where the bin identifier is a four bytes integer of the acknowledged bin.

HASH This message inform about hashes of any leaf of the hash tree. It could be used by the receiver to inform the sender about which root hash is interested in; or by the sender to transfer the hashes needed by the receiver to check data integrity. It has the format `04 BIN_ID HASH`, where the bin identifier is a four bytes integer and the hash is a 20 bytes string corresponding to the result of applying the hash function to that block.

PEX+ This message notifies about a new peer that might be contacted. It has the format `05 IP_ADDRESS PORT`, where the IP address and the port are the endpoint to contact that peer.

PEX- This message notifies a peer quit the swarm, so it is no longer available. If the receiver is in contact with that peer, the connection should be interrupted. It has the format `06 IP_ADDRESS PORT`, where the IP address and the port were the endpoint used to contact that peer. This message should be sent to all peers the leaving peer was informed about earlier.

SIGNED HASH This message inform about hashes of bins, but they are signed by the sender. It has the format `07 BIN_ID HASH SIGNATURE`, where the bin identifier is the label of the tree leaf, the hash is the actual hash value for that leaf and the signature is the result value of applying a digital signature algorithm over the hash value. This message is used in the streaming use case, as explained later in Section 2.2.6..

HINT This message is a request from the receiver asking to the sender to transfer certain data. It has the format `08 BIN_ID`, where the bin identifier is the retrieved bin label.

2.2.5. File transfer operation

After defining all the internal structures and the messages of the protocol, this section explains how it operates for the file transfer and the VoD use cases. The description includes the connection initialization, peer exchange, data transfers, acknowledgements and integrity checks. Besides all the information included in this section, a practical example of the protocol operation is presented in Appendix A.

2.2.5.1. *Channel initialization*

A channel is established with a three-way HANDSHAKE message exchange. Known the contact address and port for a remote peer, the initiator may send a datagram with a HANDHAKE message announcing which identifier must be used for that connection; and a HASH message identifying the root hash of the file to be transferred (the root hash is always identified as 7FFFFFFF). This datagram is to be sent to channel 0, as shown in the following example:

```
00000000 00 0200305
04 7FFFFFFF 56c7265192c857963f305cd5edaef5af536d3c92
```

The remote peer then must send a datagram response to the previously announced channel, containing a HANDSHAKE message with its channel identifier. It may also attach some minor payload (mainly HAVE messages reporting about its available data). Finally, the initiator must send another paket, even if it is a zero-message datagram, to keep the connection alive and complete the three-way handshake initiation.

2.2.5.2. *Peer exchange and smooth NAT traversal*

Many peer-to-peer protocols include some specific messages to exchange peer lists between themselves in a gossip fashion. This process is named peer exchange and relieves trackers of some work. *swift* defines messages PEX+ and PEX- for this purposes but they are not only used by means of peer gossiping but also for tracker-to-peer notifications. Namely, when a peer contacts a tracker in order to join a swarm, it receives one or more PEX+ messages with the information to contact active peers in the swarm.

According to the design goals for *swift*, smooth NAT traversal was a key feature for a multiparty transport protocol. To achieve so, peer exchange and NAT hole punching functionalities were joined, making this process transparent. Figure 2.6 shows the operation of this technique, where the introducing peer (A) acts actually as a STUN server [22] for introduced peers (B and C).

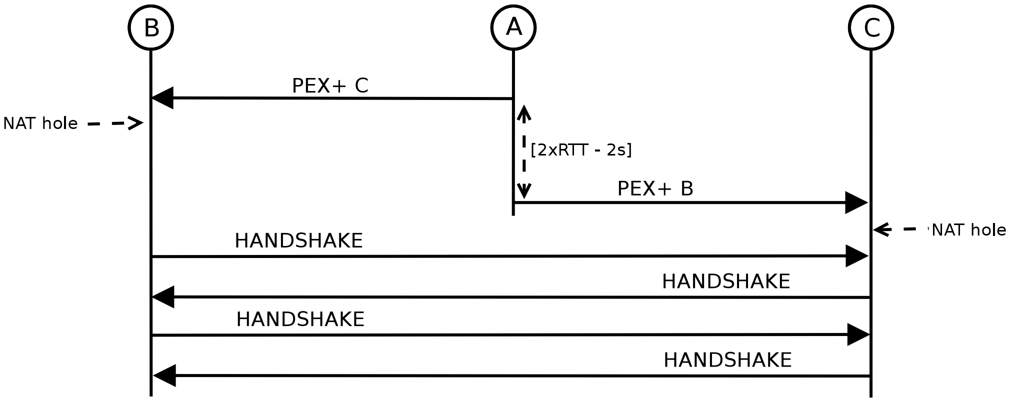


Figure 2.6: NAT hole punching operation. Once peer A notifies peer B to peer C, it should also introduce C to B with a time delay from twice the typical RTT to 2 seconds. Peers B and C then are supposed to initiate the handshake dialog to each other.

Peers may ignore PEX+ messages if they are not interested in contacting new peers, therefore the new channel initiation process would fail and the communication would not be feasible.

2.2.5.3. Data retrieving, transfer and acknowledging

Keeping in mind the idea of making *swift* ready for as much use cases as possible, the protocol was designed to work either with explicit data requests or without them. Data requests in *swift* are done using the HINT message, but the reception of one of these messages can be ignored. However, a peer should send out piece requests, by HINT messages, to coordinate with other peers in the swarm and avoid unnecessary data transmission.

Real data is transferred in terms of individual bins of 1 KByte by DATA messages, what enables the use of a logarithmic acknowledging system. Within the same channel, individual bins must be acknowledged immediately by ACK messages, which also carry a time-stamp required by the LEDBAT congestion control algorithm. To update the status information to other remote peers, HAVE messages are sent to the rest of the established channels acknowledging either individual bins or partial tree branches by reporting upper-layer bins. By this way, data can be reported to be acquired a logarithmic number of times, compensating for the datagram unreliability and avoiding data retransmissions on ACK message losses.

All peers maintain a binary tree with the status of all remote peers contacted in the swarm, created from all the incoming ACK or HAVE messages. These trees help the peers to create an overview of the swarm status and enable them to always know which bins are available from all remote peers.

2.2.5.4. Data integrity check

Data integrity is checked for every bin of data received by the Merkle tree checking procedure explained earlier in Section 2.2.2.. Data pieces can be either permanently accepted and acknowledged if the hash check is correct or dropped if not. In order to perform the integrity check, some of the hash leaves of the tree might be needed.

All senders must keep track of which parts of the hash tree the neighbor peers already know. Thus, when data is sent to these peers the needed hashes to check its integrity (known as chain of uncle hashes) could be appended in the same datagram. This latter fact is important, since all hashes needed to verify the DATA message contained in a datagram should be included in the same datagram. Only in case the resulting datagram would result to be larger than the usual MTU, HASH messages can be carried in different datagrams from the data they are intended to check. Although ideally only needed hashes should be appended, some random redundant hashes can be added in all communications.

2.2.5.5. *File length discovery and Peak Hashes*

HASH messages and the Merkle tree structure are also useful to determine the length of a file from the root hash. First time a peer joins a swarm (therefore it announces no HAVE messages identifying already acquired pieces), the sender should bootstrap the newcomer with all needed hashes to determine the length of real data of the file. These hashes are also known as peak hashes and are defined to be hashes over data-filled bins and whose parent hashes are defined over incomplete bins. Figure 2.7 shows an example on how to determine peak hashes over a file.

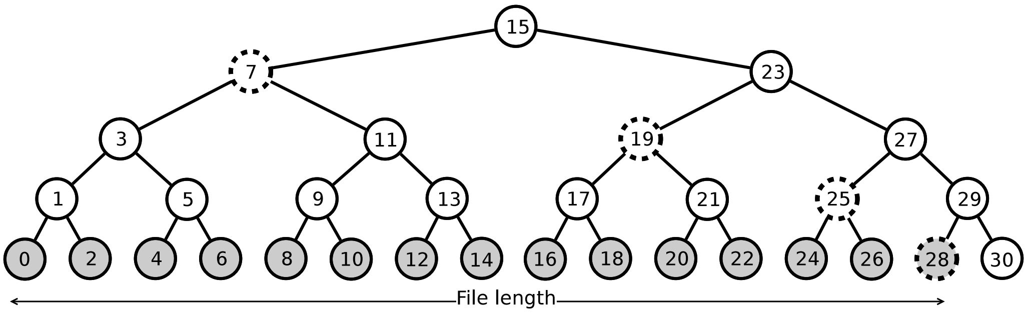


Figure 2.7: Peak hashes. Assuming a file to be 15 KBytes large (filled bins), peak hashes for that tree would be bins 7, 19, 25 and 28 (dotted line bins).

2.2.6. Streaming operation

In the case of live streaming, the size of the transferred information is not known in advance, since the content is a byte stream. Therefore, it is not possible to obtain the root hash for that content nor to use the same techniques for size discovering in file download or VoD.

To solve the lack of a content identifier, the usual root hash is replaced by a public key, which is then used to identify and bootstrap the stream and its transfer.

Size discovery loses all sense in this case, since a stream is meant to be constantly growing on size. The solution for this is to dynamically enlarge the data tree at the stream rate: the sender send new peak hashes as soon as the stream covers new areas of the virtual tree. These peak hashes are sent by SIGNED HASH messages, which are signed by the sender using the root public key to authenticate them as real peak hashes for the content.

All other procedures of this use case work as defined in Section 2.2.5. for the other covered use cases.

2.2.7. Custom congestion control

The *swift* protocol was designed to employ pluggable congestion control algorithms, so the most suitable algorithm could be used for each case. This allows servers to use TCP-like congestion control schemes while end-users are expected to use weaker-than-TCP algorithms. These latter algorithms are intended to avoid interferences of downloads or uploads on the regular traffic, based on free-bandwidth discovering algorithms and leading to incentiveless seeding.

For those peers acting as servers, so offering the tracker service, congestion control algorithms used in TCP are still valid, due to the little amount of data transferred. Classic AIMD[24] (Reno, Tahoe) or advanced algorithms implemented in TCP (Vegas, BIC[25], CUBIC) could be used.

For end-user peers or *seeders*, the goal not to interfere on regular traffic is reached using new bandwidth-scavenger congestion control algorithms. The main exponent of this new family of algorithms is the Low Extra Delay Background Transport (presented earlier in Section 2.1.2.1.), designed to yield against other connections in the same link.

2.2.8. Current implementation

Nowadays, the *swift* protocol has only one implementation, named Libswift [26] and consisting on a C++-written library which implements the protocol. At the time of this writing, the library was still under development, though it currently runs for the file download use case in the most common operating systems.

The file transfer use case is ready (and so the VoD one), meaning two instances of the library can correctly transfer a file (or a set of files) as explained in Section 2.2.5.. For these transfers, LEDBAT is used as the congestion control algorithm. The streaming case features are not implemented yet in Libswift. Currently, the bin retrieval is done in a sequential way (including some minor scramble) and the library offers a method to limit the scope of the bin retrieval process.

The Libswift library is licensed under the LGPL[27] license and runs on Mac OS X, Windows and Unix operating systems. It offers a well-known API[28] to build applications using the protocol. The last available performance tests showed a maximum transfer throughput around 400 Mbps.

2.3. Tribler

Tribler[29] is a BitTorrent client which includes some improvements over the basic protocol operation. From its website, the official definition of Tribler is *an application that enables its users to find, enjoy and share videos, audios, pictures and any kind of data*. Tribler focuses on creating a social ecosystem around data transfers, so users could interact and improve these transfers. In terms of content consumption, Tribler offers a multimedia-ready

[24] Additive Increase Multiplicative Decrease
[25] Binary Increase Congestion control
[26] `http://www.libswift.org/`
[27] Lesser General Public License
[28] Application Programming Interface
[29] `http://www.tribler.org/`

interface, so newly acquired contents can be directly played within the same P2P software client.

The first improvement added by Tribler is BuddyCast [7], a protocol which allows end users to perform content search directly from the BitTorrent client. This avoid the need of external mechanisms to obtain the .torrent files, since now they are exchanged directly between peers. Basically, this is done by creating a new swarm with all Tribler users, letting them to exchange their content preferences and the .torrent files to obtain these contents. This feature evolved in a recommendation system, in which after some transfers Tribler learns about user tastes and recommends some related contents.

Another key feature in Tribler is BarterCast [23], a fully distributed system for reputation management integrated into BitTorrent. The protocol exchanges altruism levels of peers and creates incentives for cooperation between them, trying to avoid free-riding by taking into account reputation data when unchocking peers in BitTorrent.

Related to reputation systems, collaborative downloading in BitTorrent was first introduced in Tribler, by means of 2Fast [24]: a newly built protocol which let users help other peers with their idle bandwidth. The client offers the possibility to select some other users as friends. These friends can use the user's free bandwidth to retrieve some data pieces to other peers and then transfer them to themselves. This procedures is proved to be efficient speeding up the download rates.

In terms of implementation, Tribler is written in Python, on the top of the ABC Bittorrent client[30], based at the same time on BitTornado[31] and the original BitTorrent Core System from Bram Cohen. As BitTornado does, Tribler can obtain data from BitTorrent swarms and HTTP servers, combining both protocols to obtain better results.

[30] http://pingpong-abc.sourceforge.net/

[31] http://www.bittornado.com/

CHAPTER 3. PROBLEM DESCRIPTION

After describing the operation of BitTorrent, the design and operation of the new *swift* protocol and the Tribler challenges, this chapter introduces which problems are faced in the scope of this thesis work.

The first goal of this work is to build a multi-protocol P2P client, working with BitTorrent, HTTP and *swift* to obtain data pieces and barter with them. This way, Tribler could become more efficient and improve its transfer rates, offering the end-users better experience for content consumption. Moreover, the use of LEDBAT for outgoing connections would let the user to fill up its up-link connection, but not to interfere other applications traffic.

3.1. Research questions

Besides building the multi-protocol P2P client itself, this project looked forward on how this new solution worked in real environments.

Specifically, this thesis wants to prove if the bandwidth usage is improved when using several protocols at the same time. Also, determine which is the best piece picking technique when combining BitTorrent and *swift* to maximize the used bandwidth is a challenge to face.

Bandwidth usage

Thanks to the use of the new congestion control algorithm, LEDBAT (described earlier in Section 2.1.2.1.), Tribler should be able to use all the upload bandwidth without disturbing the user. Due to its design, LEDBAT always yields to other connections, so it is not possible to prioritize its transfers in front of other TCP transfers (which controversially would go against its main design principle).

The interest here is to discover how LEDBAT behaves when sharing the link with legacy BitTorrent TCP transfers (see Figure 3.1 to understand the expected behavior).

Theoretically, UDP transfers would back off in front of others (BitTorrent TCP transfers in this case) if the latests use all the bandwidth, but it would be interesting to discover if LEDBAT efficiently uses the remaining bandwidth and fills up the up-link bottleneck when some bandwidth is available.

Thus, *swift* would merely work as a transfer boost, more overall bandwidth would be available in the swarm and so the transfer performance would be improved in all the transfers in the swarm.

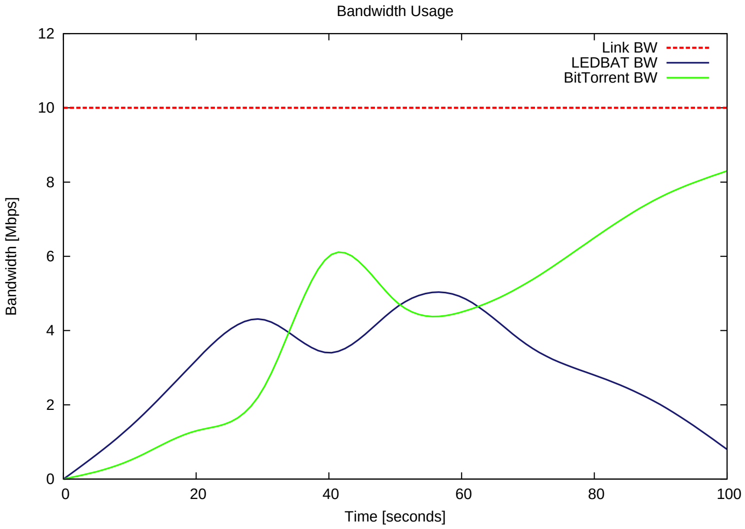


Figure 3.1: Theoretical bandwidth usage evolution. LEDBAT is able to use all non-used bandwidth by BitTorrent, so overall rates should be always close to the limit when *swift* is used as a transfer boost.

Piece translation

Difference in piece sizes between BitTorrent transfers and *swift* bin system is also an issue to solve when joining both protocols in a single client. Remember here that *swift* data bins are 1 KByte long, while BitTorrent uses variable size pieces (in the protocol specification it is stated that they must be 2^N bytes long) and splits them in 16 KBytes chunks.

Thus, some piece retrieval coordination mechanism must be designed in order to correctly coordinate the BitTorrent and *swift* operation, so the same client can obtain and share all parts of the content from and to both swarms at the same time.

Multi-protocol piece retrieval techniques

Considering BitTorrent and *swift* were independently designed, each protocol has its own features and its operation mode. The idea for this first integration is to keep BitTorrent as the main transfer protocol, so *swift* needs to adapt to the BitTorrent operation.

Taking into account the constraints of Libswift about bin requesting from external tools, several piece retrieval mechanisms could be used when running Tribler and Libswift:

- **Opportunistic piece retrieval:** request only one piece from the *swift* swarm at a time, as if the whole swarm was only one BitTorrent peer. This is the simpler approach but probably the obtained performance would be very low.
- **Bulk data retrieval:** request several pieces to the *swift* swarm, so Libswift can work on a longer set of bins to acquire. This approach requires higher control over the pending pieces, but would lead to much better performance. The operation of this mode could be definitely limited by the Libswift external requesting mechanism.

CHAPTER 4. DESIGN AND IMPLEMENTATION

This chapter describes the design of the integration of *swift* as a supported protocol in the BitTorrent Tribler client, so the latest becomes the first *swift*-capable P2P client.

To achieve this goal, several challenges need to be solved: how to integrate Tribler Python and C++ Libswift codes into a single piece of software and how to fit the new *swift* protocol into the current BitTorrent operation in Tribler.

4.1. Design mainlines

The current Tribler client is a BitTorrent-based P2P client, with the added feature of using web servers as data source. The goal of this work is to add *swift* as another supported protocol, so data could be obtained from and seeded to *swift* swarms (see Figure 4.1 for very high level sketch).

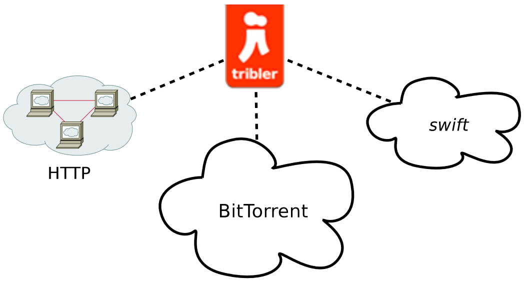


Figure 4.1: Tribler as a multi-protocol P2P client, supporting BitTorrent, HTTP and *swift*.

Instead of writing a new full implementation of the *swift* protocol in Python and use it in Tribler, integrating the current Libswift implementation seemed to be a much more feasible choice: reuse a solid and already-working software with better performance because of the C++ compiled code. How this integration is achieved is explained later in this chapter (refer to Section 4.2.).

Even though Libswift is a working implementation of the *swift* protocol, it is designed to be integrated in upper-level applications as a transport layer. In this particular case, Libswift is planned to work in combination with BitTorrent as part of a P2P file-sharing client. Thus, a part from integrating the library, a control module was needed to adapt *swift* operation to BitTorrent's, making the second to control the data flow from the first. Section 4.3. in this document describes how this module was designed, the operations it performs and how it works within the Tribler client.

4.2. Python and C/C++ integration

The first open issue to face when integrating Libswift into the Tribler client was how to join its Python implementation with C++ Libswift's implementation. The most suitable procedure here was to wrap the C++ *swift* implementation and make it accessible for the Python Tribler's code, as if it was another simple Python module.

Python is an interpreted language that relies on another software, called interpreter, to execute its code. Although many Python implementations exist, the most common and widely used one is CPython[1], which is written in C and provides an implementation for all Python instructions and methods in this language.

Building Python bindings directly using the CPython API is a complex and tedious process, since small changes in the implementation could mean rewriting the whole binding. Moreover, debugging errors is not a trivial job in this case. However, several initiatives exist to make this work lighter: they provide tools to generate the needed bindings and modules from a new small piece of code. The considered tools and their little descriptions are listed and described below:

- **SWIG[2]:** this is a software development tool that connects C or C++ written programs with several high-level programming languages (such as Python, Perl, Java, Ruby, PHP and many others). For Python, SWIG automatically creates a C/C++ binding and a Python module for that binding, so C/C++ methods and attributes can be transparently accessed from Python code. It is a lightweight tool, its results are platform-independent and is one of the most used tools for C/C++ wrapping.

- **Python-SIP[3]:** this is a an extension module generator specifically designed for creating only Python modules binding C/C++ libraries. This tool derived from the PyQt[4] project and initially intended to help on the GUI development for complex software pieces. Its operation is similar to SWIG, although it is based on a different interface format: automatically generate a binding and its Python module. It is also light and platform-independent.

- **Boost.Python[5]:** this is a C++ library which enables very smooth interoperability between C++ and Python. It is part of the Boost library set and provides full integration with the tools offered by these libraries. Anyway, it is said to be one of the best C++ to Python integration. All the Boost libraries are ready to work on almost all modern operating systems.

- **Pyrex[6]:** a new programming language for writing Python extension modules that mix Python and C data types. This code is later compiled into a C extension which can be accessed. Available for many of the current platforms.

[1] `http://wiki.python.org/moin/CPython`

[2] `http://www.swig.org/`

[3] `http://www.riverbankcomputing.com/software/sip/`

[4] `http://www.riverbankcomputing.com/software/pyqt/`

[5] `http://www.boost.org/doc/libs/1_42_0/libs/python/doc/index.html`

[6] `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`

- **ctypes[7]:** this is a foreign function interface package for Python, which allows to call functions offered by library APIs. It works under almost any existing platform, but it offers no support for C++ code wrapping.

After a rough study over the C-to-Python wrapping tools, SWIG and Boost.Python seemed to be the most convenient options for Libswift. Though its promised better C++ interfacing, its focus on providing access to all other Boost libraries (which were not used in Libswift implementation) dismissed Boost.Python as the chosen tool for this case. SWIG became the natural choice, thanks to its simplicity but its good enough C++ integration and performance.

Data type management in SWIG is quite simple as long as basic types are used. Since *swift* implementation declared some complex data types in the API, an extra wrapping over this original API was needed. This extra layer declares the same methods the original Libswift's API, but using standard or well-known C data types, and adapts the information to perfectly fit into the original methods.

The basic data type methods are then exposed to SWIG, using ANSI C/C++ declarations and special SWIG directives to describe the target methods and the desired result module. From this file, SWIG generates a C/C++ file containing all the wrapper code and Python file describing the new module itself. The C/C++ wrapper file must be then compiled with all other needed code files and linked as a shared library for Unix operating systems or a DLL[8] for Windows environment.

The full procedure and the used instructions to create the Libswift Python module using SWIG are described in detail in Appendix B.

4.3. *swift* integration into Tribler

In order to able Tribler to use the new multiparty transport protocol, some integration with the present `BitTorrent Download Manager` module was needed. Currently, Tribler instances a `BitTorrent Download Manager` for every active torrent, which at the same time starts a `PiecePicker` algorithm and several BitTorrent data transfer controllers. Optionally, an `HTTP Downloader` can be started to complement the regular BitTorrent transfers. This manager retrieves pieces from web servers by HTTP requests and considers each server as an active *seeder* in the swarm.

The complete solution here would be to design and implement a new *swift* download manager which was aware of all *swift* internal matters (such as peers, their status or transfer times) and worked in coordination with the current `BitTorrent Download Manager`. However, the immaturity of Libswift implementation and specially the idea of abstracting the network as a single data cloud (hiding peer management, piece retrieval and transfer management to the application layer) discouraged this choice.

[7] `http://docs.python.org/library/ctypes.html`

[8] Dynamic-link Library

Alternatively and as a first approach of *swift* integration in Tribler, the `HTTP Downloader` could be taken as a model. In this case, a new *swift* `Downloader` must be designed, making the *swift* swarm to be considered a single BitTorrent peer acting as a *leecher* for the Tribler `PiecePicker`. Thus, the novel module should interact with the regular BitTorrent's piece picker operation and translate the BitTorrent piece requests to *swift* operation. Figure 4.2 shows a high-level module schema, presenting the intended interaction between all of the involved ones.

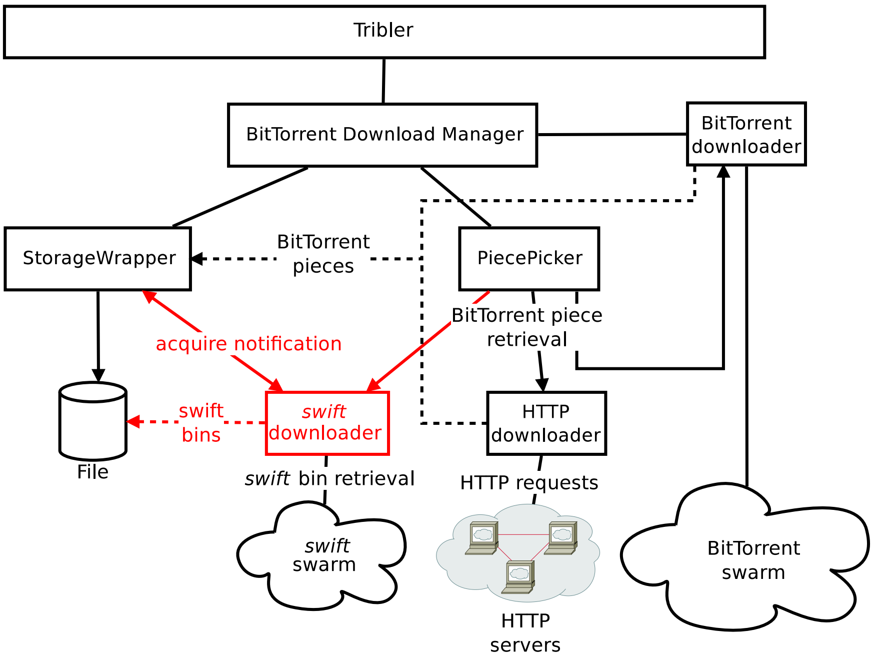


Figure 4.2: Tribler module overview. Red color indicates the new components designed to integrate *swift* as a transport protocol for Tribler.

Tribler proceeds as following when downloading a regular .torrent file: the `BitTorrent Download Manager` loads the .torrent file ans reads its information. Then, it starts a `StorageWrapper` instance to create a file on the hard drive and a `PiecePicker` to decide which pieces to retrieve at any time of the transfer. Then, it contacts the tracker to obtain the list of active peers in the swarm. Immediately, one `BitTorrent Downloader` is initiated for each peer in that list. These modules are responsible for transfer and receive data to and from remote peers. All downloaders then request the `PiecePicker` which piece to retrieve, notifying which pieces are available from the remote peer. The `PiecePicker` checks on the `StorageWrapper` which pieces are not yet retrieved or present on the file, runs an algorithm and answers to the downloader the index of the piece to acquire. The `BitTorrent Downloader` then obtains all the chunks forming the indicated piece and deliver the data to the `StorageWrapper`. Finally, the `BitTorrent Downloader` asks the `PiecePicker` again for another piece, repeating the same process until the whole file is obtained.

4.3.1. *swift* metadata in .torrent files

The first issue to solve is how to integrate the *swift* bootstrap metadata (consisting only on a root hash and a tracker URL) in the legacy BitTorrent .torrent files. The cleanest and simpler solution was to add this information inside of the .torrent file without breaking its integrity, so a Python script was written to modify existing .torrent files.

The choice was to create a new dictionary inside of the pre-existing .torrent file, containing a text string parameter for the *swift* root hash and a list of string parameters for tracker URLs. Additionally, another string list could be attached to include a set of well-known peers in the target swarm.

Once the .torrent contains the *swift* information, it is ready to be used from Tribler. The `BitTorrent Download Manager` would read all needed information to bootstrap the BitTorrent transfer and if the *swift* metadata is present, it would create an instance of the new *swift* download manager to join the corresponding *swift* swarm.

4.3.2. The *swift* Downloader module

The *swift* `Downloader` module would use the Libswift library through the Python wrapping module described earlier in Section 4.2. to interact with other *swift* peers in the swarm and transfer data. However, in this case, Libswift should not acquire all data bins linearly (as it does nowadays in the standalone operation) but only acquire the bins corresponding to the BitTorrent pieces ordered by the Tribler `PiecePicker`.

4.3.2.1. Piece management

First consideration is to state that BitTorrent piece length will be always regarded as a power of 2. This way, BitTorrent pieces can be easily converted to *swift* bins, as one of the pieces corresponds to a branch of the tree identified by a bin from layer N (shown in Figure 4.3). Then, using one of the public methods of the library, it is possible to limit the scope of Libswift retrieval to only the bin range the Tribler `PiecePicker` pointed.

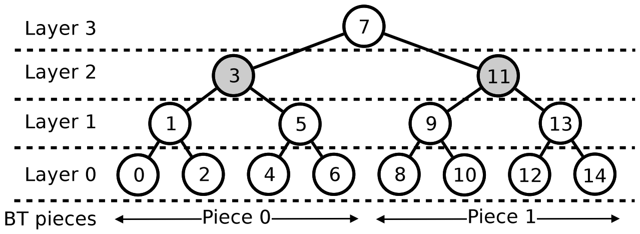


Figure 4.3: BitTorrent pieces to *swift* bins. Assuming 4 KBytes BitTorrent pieces, each layer 2 bin would stand for a whole BitTorrent piece: in the diagram, bin 3 would stand for BitTorrent piece 0 and bin 11 for piece 1.

Second consideration is that only .torrents describing a single file would be managed. As a first implementation, this would keep things simpler and would make easier the piece management.

The novel module should then determine which bins should be acquired from the *swift* swarm to fulfill the `PiecePicker` requests. The easiest procedure here seems to determine which layer of the tree stands for full BitTorrent pieces and then use the piece index as the offset using the layer-and-offset bin notation.

At this point, the *swift* `Downloader` should notify Libswift which pieces the client is interested to obtain from the *swift* swarm. This is done using a method offered by the library which limits the scope of bins that should be retrieved from the swarm. Limiting this range to the bins corresponding to the requested BitTorrent piece is the only way to perform this action at the time of developing the module.

4.3.2.2. Piece storage

All the obtained pieces should be written to and read from the file described in the .torrent and stored on the disk. While Tribler uses an abstraction module called `StorageWrapper` to perform all disk-related tasks, Libswift writes and read all the obtained bins directly to or from the file.

In the case of Libswift use through a Python wrapper, the library runs inside the same process as the Python interpreter. Thus, Libswift and Tribler can access to the same file as long as they work in separate areas of the file.

To make the `StorageWrapper` notice about the parts of the file already written by Libswift, a new method was written. This method only sets up the *swift* written parts of the file to be real data, hash check and other BitTorrent specific operations are omitted.

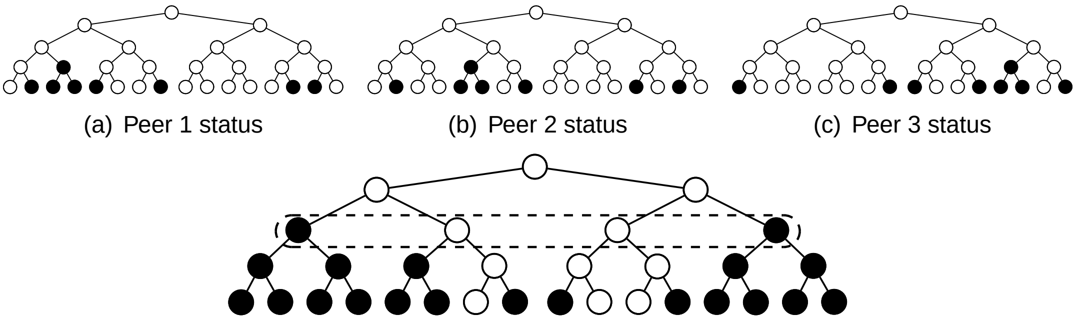
4.3.2.3. Piece notification

Since two requesting and acknowledging systems (BitTorrent and *swift*) will be used, this module should coordinate the operation of both protocols by notifying each other about the progresses done on the other swarm.

First, this module notifies the Tribler `PiecePicker` which pieces are available to be obtained from the *swift* swarm. With this information, the `PiecePicker` can better determine which pieces to retrieve first (usually following a rarest-first picking algorithm). This information is maintained by Libswift in a binary tree for each remote peer contacted, so some method was needed to obtain it in a way the `PiecePicker` understood it.

Libswift internally had methods to combine already existing binary trees and collapse them into bit strings. To obtain the full BitTorrent pieces available on the swarm, all the remote peer status trees should be combined by an `OR` operation, obtaining a single tree with all the available bins in the swarm. Later, this tree can be flattened to the layer standing for full BT pieces to obtain the desired bit string representing the available pieces, ash shown in Figure 4.4(d).

The module also notifies Libswift about which pieces were already acquired by external


(a) Peer 1 status

(b) Peer 2 status

(c) Peer 3 status

(d) Resulting OR binary tree and coarse bitmap

Figure 4.4: Binmap combination and collapse. Assuming binary trees representing the status of remote peers to be the ones shown in (a), (b) and (c), the result of applying an OR operation to all trees would be the one in (d). If BitTorrent pieces were 4 KBytes, reading the tree to layer 3 would result in the bit string 1001.

means (in this case either BitTorrent or HTTP transfers). Therefore, Libswift should notify remote peers the corresponding bins were already acquired (using HAVE messages) and can transfer them. Again, the layer-and-offset notation (using the layer standing for full BitTorrent pieces and the BitTorrent piece number as the offset) can be used.

A little modification was done on the Tribler StorageWrapper module to execute this action every time the BitTorrent Downloader or the HTTP Downloader acquire a piece. Same translation procedures used to determine which bins to acquire from the PiecePicker requests (explained earlier in Section 4.3.2.1.) can be used here.

In both cases, hash checking and data corruption should not be an issue because each piece of software perform independent data integrity checks (even though different techniques are used).

4.3.2.4. Threading integration

Although Tribler is a multi-threaded application, most of the BitTorrent related tasks are executed by a single thread called *Network Thread*, including network socket management, disk interaction and piece picking algorithms.

The Libswift library has as an asynchronous single-threaded design and needs the external application using it to constantly call a method to run the library. This method runs for the specified amount of time, polling over a UDP socket waiting for incoming datagrams and sending the needed messages to retrieve and share data. To integrate the Libswift module into the Tribler operation, a new thread is needed to execute all instructions interacting with the C++ code. From now on in this document, this thread will be regarded as the Libswift Thread.

Also, Python threading limitations must be regarded. The Pyhton interpreter executes only one thread at a time, even when running on multi-core systems, with the goal to simplify thread interaction. When using Python threads, the interpreter automatically switch between threads after a certain and fixed number of instructions, working as a pseudo multi threading environment.

However, this is no longer valid when running C/C++ code from Python modules: the interpreter does not switch threads until the C/C++ instructions are not fully completed. The reason behind this fact is C/C++ usage was only meant to perform high-performance (so extremely fast) operations. So that, in this particular case, the `Libswift Thread` would monopolize the execution time, making stop all the BitTorrent operation.

To solve this issue, a time-based operation mode was designed to switch between active threads. A breaking condition was included on the `Libswift Thread` so every time Libswift runs for a certain amount of time, the thread sleeps for the same time to let Tribler perform all other needed actions. Although this is not an optimum solution, it is valid enough as first integration approach.

Finally, to assure all Libswift related tasks were run in the `Libswift Thread`, a task scheduler mechanism was implemented. All other threads can add a task to execute by the `Libswift Thread` and they are stored on a queue. The queued tasks are executed every time the thread is run. Together with a similar feature of the *Network Thread*, this becomes the communication mechanism between both threads.

Figure 4.5 shows the thread interaction and execution times when running Tribler with *swift* capabilities.

4.3.2.5. Piece retrieval techniques

Tribler includes several piece picker algorithms for BitTorrent, intended to be used in several cases, such as Video on Demand, streaming or regular files. These algorithms work basically following the rarest-first principle, introducing some weight to certain pieces when running on delay or order sensitive cases. All of them were designed considering each `BitTorrent worker` would be only acquiring one piece at a time.

The first approach was to make the *swift* `Downloader` work as a `BitTorrent Downloader`, requesting the `PiecePicker` for a single piece index and setting the corresponding bin range in the Libswift limiter as the only active one. However, the constraints of the time-based Libswift execution created an issue: if the limited range corresponding to a full BitTorrent piece is small and can be acquired in little time, neither Libswift nor BitTorrent will download more data until the `Network Thread` runs again and so the `BitTorrent Downloaders`.

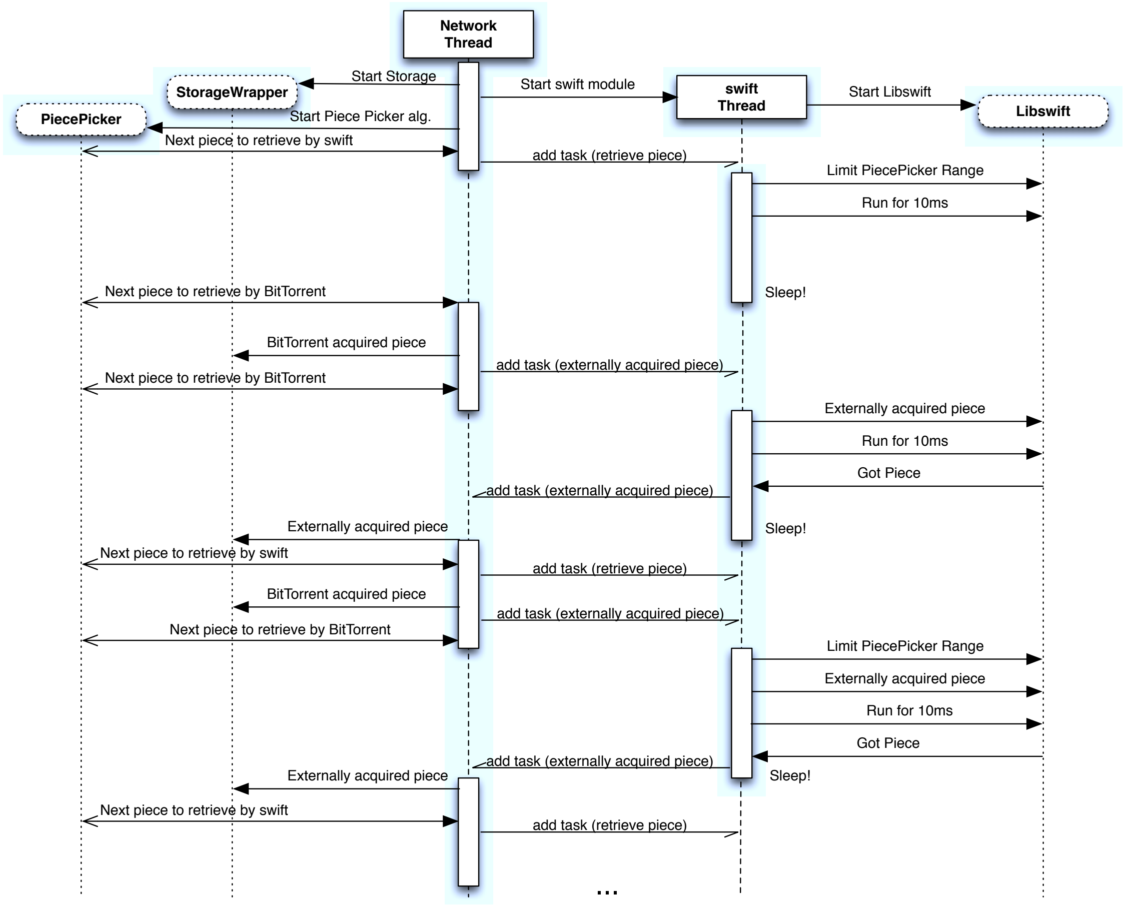

Network Thread
StorageWrapper
PiecePicker
swift Thread
Libswift
Start Storage
Start swift module
Start Libswift
Start Piece Picker alg.
Next piece to retrieve by swift
add task (retrieve piece)
Limit PiecePicker Range
Run for 10ms
Next piece to retrieve by BitTorrent
Sleep!
BitTorrent acquired piece
add task (externally acquired piece)
Next piece to retrieve by BitTorrent
Externally acquired piece
Run for 10ms
Got Piece
add task (externally acquired piece)
Externally acquired piece
Sleep!
Next piece to retrieve by swift
add task (retrieve piece)
BitTorrent acquired piece
add task (externally acquired piece)
Next piece to retrieve by BitTorrent
Limit PiecePicker Range
Externally acquired piece
Run for 10ms
Got Piece
add task (externally acquired piece)
Externally acquired piece
Sleep!
Next piece to retrieve by swift
add task (retrieve piece)
...

Figure 4.5: Thread interaction mechanism. This diagram shows the thread interaction mechanism designed when integrating the new *swift* capabilities to Tribler. Also, the module interaction is shown.

CHAPTER 5. EXPERIMENTS AND EVALUATION

This chapter describes the experiments done with the implemented Libswift to Tribler integration.

The goal of these tests was to evaluate the operation of the new solution and the performance obtained when combining *swift* and BitTorrent under different conditions.

5.1. Testing tools and procedures

In order to run all the experiments, several tools were used in all the scenarios. In all cases described in this chapter, the same 928670754 bytes long file was the data source of the swarms.

Simple BitTorrent tracker and seeder

Starting a BitTorrent tracker and seeder with Tribler is quite simple using the already developed tool *dirtrackerseeder*.

This script reads a directory looking for .torrent files, which then are offered in a BitTorrent tracker and seeded as a regular BitTorrent seeder. Data files must be also present in the same directory.

Also, the *dirtrackerseeder* tool can limit the upload rate for the BitTorrent seeder, a very interesting feature to test the operation under different traffic situations.

Logging features

The measured variable in the experiments will be the instant download rate reached by each protocol. The easiest way to measure this is to save a timestamp and the number of incoming bytes each time a full BitTorrent is acquired.

Pyhton logging tools[1] were used to add to the *swift* `Downloader` module (described earlier in Section 4.3.2.) the feature of saving a file with the timestamp and the piece size for both protocols. From that information, it is easy to calculate the instant bandwidth and the accumulate mean bandwidth, the information shown in the following experiment results.

[1] Logging facility for Python: `http://docs.python.org/library/logging.html`

Bandwidth estimation

The bandwidth used by each protocol can be easily estimated from the logged timestamps and the BitTorrent piece size in bytes.

On one hand, the instant bandwidth is obtained dividing the amount of obtained data since the last timestamp by the difference between the current and the previous timestamp (see Equation 5.1).

$$BW_{instant_N} = \frac{\#bytes}{t_N - t_{N-1}} \tag{5.1}$$

On the other hand, the mean is obtained by dividing all the obtained data by the time since the beginning of the module operation (see Equation 5.2).

$$BW_{mean_N} = \frac{\sum_{0}^{N} \#bytes}{t_N - t_0} \tag{5.2}$$

Adapting Libswift

BitTorrent is naturally a request-reply protocol, where peers explicitly request for the information they are interested in. Tribler `PiecePicker` algorithms are only prepared to work when data is requested before it is acquired. In order to adapt Libswift to this behaviour and preserve the nature of the experiments, the `PUSH` operation of the protocol was deactivated in the seeder. Thus, the protocol worked in a request-reply manner as BitTorrent does.

After the preliminary tests of the implementation, a Libswift-related issue appeared: when using the method to notify some parts of the data tree were acquired by external means (mainly BitTorrent in this case), the hashes of the new bins are not received. Since these hashes were needed later to perform the hash-check for other parts of the tree, this caused *swift* to drop the bins which integrity was not possible to check and lots of data retransmissions (see Figure 5.1 for graphical explanation). At that time, Libswift was limited to acquire a range of bins that could not be checked, so it was blocked in a loop receiving and dropping always the same bins.

This issue was unexpected and was out of the initial scope of this thesis work. So that, in order to quick solve the problem and be able to run the tests, an easy shortcut was taken. After starting the *swift* seeder, Libswift creates a binary file containing the hashes of the hash tree needed for that transfer in the same directory where the seeded file is. The receiver also creates this file, but hashes are written into it as soon as they are received from the swarm (as `HASH` messages). The solution then was to overwrite the file created by the receiver (which at the beginning of the transmission is empty) with the one created by sender (which contains the whole hash tree).

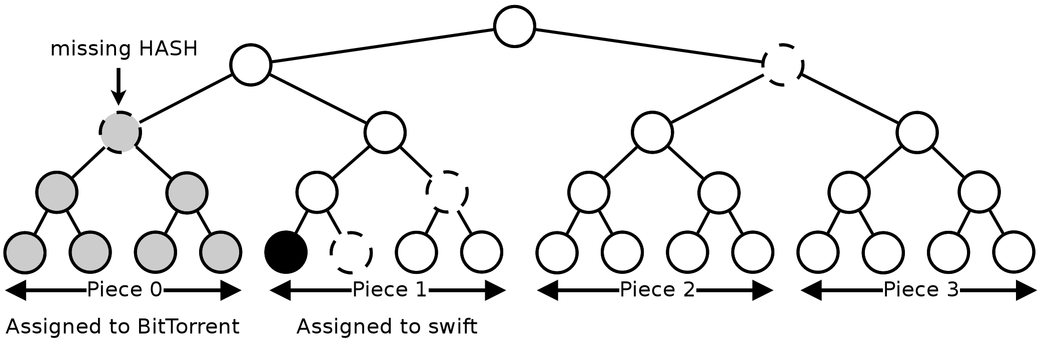


Figure 5.1: Hash check fail when missing bins. Assuming bin (2,0) (corresponding to BitTorrent piece 0) was marked as being acquired by BitTorrent, all the hashes within that bin would be missing. If Libswift is assigned to retrieve the BitTorrent piece 1 (and so the bin (2,1) in *swift* notation), when receiving bin (0,4), hashes (0,5), (1,3), (2,0) and (3,1) (all the dotted line bins) should be prepended to the DATA message. However, hash (2,0) is neither received from the swarm because the peer announced to have it (sending a HAVE or an ACK message) nor present there because it was acquired from BitTorrent and that branch of the hash tree was not calculated. Then, the hash check against the root hash fails and data must be dropped.

5.2. Operation assessment

The first experiment performed had the intention to assess the correct operation of the implemented integration, making sure the piece retrieval and the protocol coordination worked as expected. The scenario used to run this test was the one shown in Figure 5.2. The expected result was BitTorrent downloading data at a fixed rate and Libswift obtaining also some data.

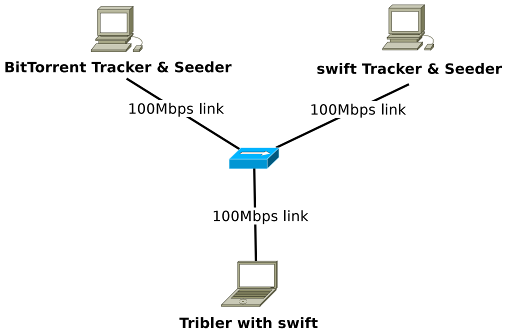


Figure 5.2: Operation assessment scenario. Both seeders had the complete file so the new Tribler client with *swift* capabilities should download data from both swarms (BitTorrent and *swift*). The BitTorrent seeder upload capacity was limited to 8 Mbps so the test could run for longer time. Note that all the computers involved in this experiment were not performing any other CPU or network consuming task that could interfere in the results.

After running the experiment, the obtained results in terms of used bandwidth were the ones shown in Figure 5.3.

With the obtained results and after checking the integrity of the obtained file, it can be

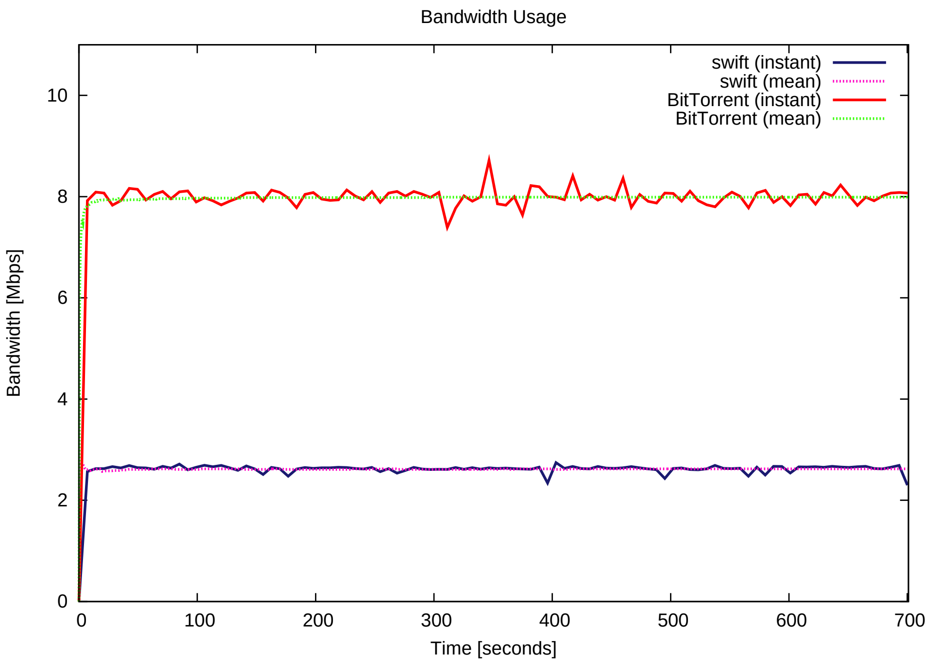


Figure 5.3: Assessment results. As expected, BitTorrent only downloaded at the limited rate of 8 Mbps, but *swift* bandwidth (around 2.5 Mbps) indicates some pieces were obtained from the *swift* swarm.

stated than the implemented integration of Libswift into the Tribler code works in terms of piece retrieval and acquiring coordination.

5.3. Performance measurements

After making sure the protocol coordination worked fine, the performance of the implementation needed to be assessed. To do so, the same scenario described in Figure 5.2 was used.

The procedure here was to test the solution under different BitTorrent rate conditions. Therefore, the same test was run in three more situations: running Tribler with the BitTorrent rate unlimited (see results in Figure 5.4); the rate limited to 40 Mbps (see results in Figure 5.5 and to 2 Mbps (see results in Figure 5.6).

After running all the experiments and considering also the results obtained in the operation assessment (presented earlier in Section 5.2.), it can be clearly stated that the maximum download rate obtained from the *swift* swarm when using the built software was around 2.5 Mbps, independently of the traffic present in the network link.

Although the obtained results were not promising, this work pretended to be a first approach to protocol integration. The reasons most likely to be behind of this low performance are the time based operation used when integrating the newly created thread with the Tribler `Network Thread` and the single-piece requesting mechanism used to interact with Libswift .

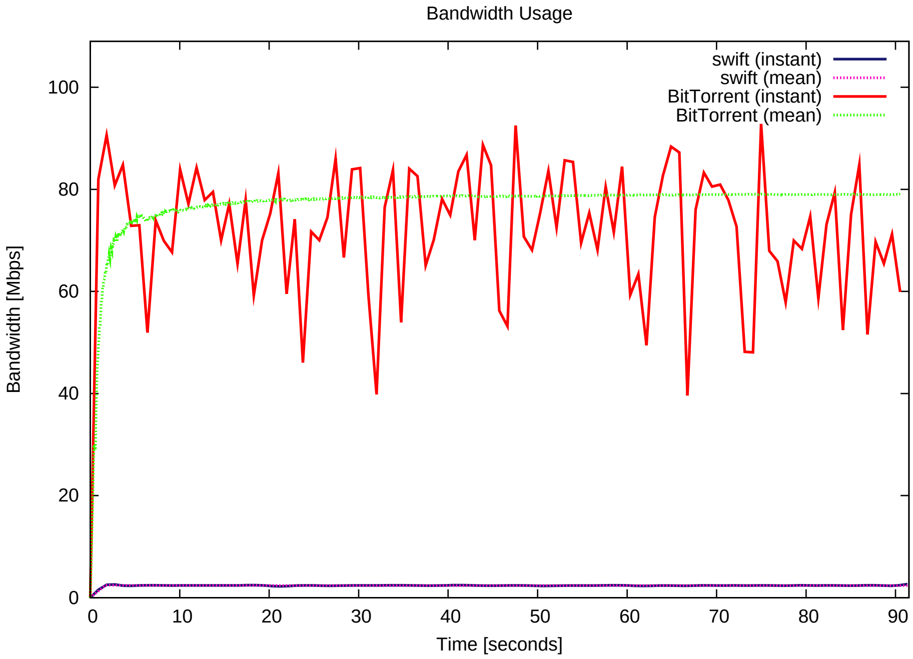


Figure 5.4: Bandwidth evolution when BitTorrent not limited. As shown, BitTorrent works close the the link limit and *swift* uses only around 2.5 Mbps.

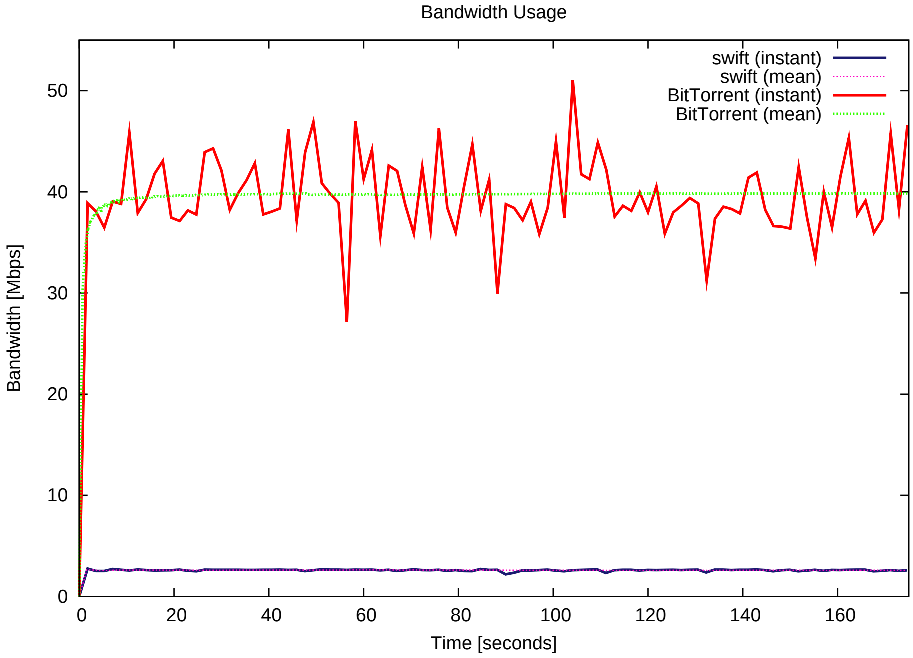


Figure 5.5: Bandwidth evolution when BitTorrent limited to 40 Mbps. BitTorrent works only at around 40 Mbps but *swift* keeps using only around 2.5 Mbps.

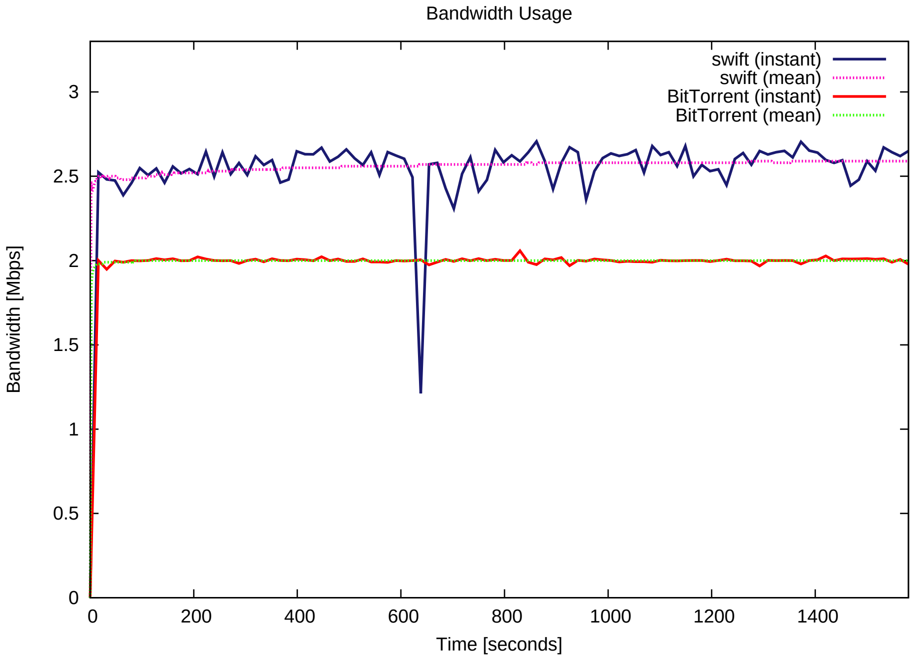


Figure 5.6: Bandwidth evolution when BitTorrent limited to 2 Mbps. Again, while BitTorrent uses only 2 Mbps (so the link is almost empty), *swift* again only uses around 2.5 Mbps.

CHAPTER 6. CONCLUSIONS

This chapter presents in Section 6.1. the final conclusions made after all the performed tasks and experiments. After that, the next steps to be done if this work was to be continued and future research ideas are listed in Section 6.2..

Then, a brief environmental impact analysis of the involved technologies and the output of this project is included in Section 6.3.. Finally, some personal conclusions are presented in Section 6.4. to show my motivation and the progress of the project.

6.1. Achieved objectives

With the obtained results and after implementing the first prototype for *swift* usage in Tribler, this section summarizes the contributions of this project.

The first one was the implementation of the first *swift*-to-Tribler integration prototype itself and the first appliation using the new transport protocol. The developed module is a key Tribler feature for the deployment of the new *swift*-based next-generation P2P engine. Although the obtained performance is not as good as expected, this first approach put some light over some issues to solve when joining *swift* with other legacy protocols.

Even though the poor download rates obtained from *swift* swarms using the new piece of software, the devised, designed and implemented inter-protocol cordination mechanism for Tribler proved to work good when combining BitTorrent and *swift* . The piece translation technique used to cordinate BitTorrent pieces and *swift* bins showed to smoothly convert from one domain to the other and the multi-protocol piece retrieval was only tested for the opportunistic retrieval case due to the limitations of the external piece requesting mechanism in Libswift. Nevertheless, this low performance hampered the LEDBAT behavior study when working in parallel with BitTorrent in a resticted network link.

Moreover, this first attempt and the run tests raised few issues in the current *swift* operation:

- The external piece requesting mechanism offered by Libswift is not powerful enough to work with the current Tribler `PiecePicker` operation.
- The external piece acquiring notification system offered by Libswift may lead to a defective operation of the protocol due to the absence of some needed hashes, so it is not valid as it is now.

The inclusion in the protocol definition and in the Libswift implementation of new features to cover the discovered lacks should be considered.

From the implementation itself, also some facts were discovered and could be stated in this section:

- The designed threading interaction schema is not valid for a solid and effective protocol cordination. The time-divided execution and the thread interaction proved to have a devastating effect on Libswift download rates.
- Using C/C++ implementations from Python code is quite useful. Although manually building the bindings is a tedious and hard task, several tools were built to ease this process and each one is intended to be used in different cases.
- The Python threading system is supposed to emulate a multi-threading execution behaviour by switching threads after several instructions, since only one thread can run at a time within the same Python interpreter instance. However, when running C/C++ code this statement is no longer valid and other strategies have to be considered.

Finally, the first goal of building a *swift* integration into the Tribler client was achieved during the development of this thesis.

6.2. Future Work

The first integration approach of the novel *swift* protocol into the Tribler BitTorrent client left some open issues to solve.

The following list describes the pending tasks or the open issues that need to be solved to try to increase the download rates from *swift* swarms:

Enable non-requested pieces in Tribler As briefly mentioned earlier, Tribler algorithms are not ready to receive data pieces without sending requests. Nevertheless, *swift* was meant to work with and without explicit requests. So, it would be very interesting to adapt these algorithms to this behavior and discover if activating the `PUSH` operation in Libswift increases the transfer rates.

Signal missing hashes in *swift* As discovered while running the first tests with the new piece of software, when *swift* works together with any external data source (such as BitTorrent), some problems appear when checking data integrity (as explained earlier in Section 5.1. and Figure 5.1). To obtain the first performance results, a shortcut was taken, but a solid solution is still missing. Two new features could be added to the *swift* protocol to solve this issue: add an explicit hash chain request message, so peers can request other peers for specific hashes; or add a data acknowledgement message to indicate data was correctly acquired but hashes are missing.

Improve the piece retrieval mechanism Due to the limitations of the existing requesting mechanism for Libswift, the current implementation can just work in one BitTorrent piece at a time. Considering that the new *swift* `Downloader` interacts with a full swarm (the `BitTorrent Downloaders` interacts with only one BitTorrent peer), working on several pieces is a must for future improvements. One first step on this way that would not require a new requesting schema could be to assign Libswift a 2^N number of consecutive BitTorrent pieces, higher level bins in the tree (and so larger data ranges) could be signalled.

New threading schema The current time-division based execution of Libswift proved to be not efficient enough. The current Tribler thread design was not ready to support an external library like Libswift. Moreover, the Python thread running limitations made things worse. Therefore, how the Libswift library is executed in Tribler is one of the most important topics for future improvements. The first should be to make the `Network Thread` to execute Libswift itself, facing all the synchronization issues. Further, combination of Tribler and Libswift network management systems should be considered.

6.3. Environmental analysis

Nowadays, the environmental impact of any technological implementation must be considered before its deployment. National and local regulations or laws must be studied if the new solution is to be deployed as a service.

When talking about computer networks, the most critical issue in terms of environmental impact is the energy consumption. Currently on Internet, the server-client paradigm is still the most common to offer many services to final users. Therefore, servers are always working to offer the service to potential consumers (even if no clients are using the service) and so wasting energy. Moreover, ventilation and security systems (also wasting energy) and UPS[1] systems (which are basically batteries containing some contaminant materials) also have to be considered as an environment threat.

As explained earlier in this report, peer-to-peer systems can be regarded as a new paradigm to offer some of the services currently provided by servers. Thanks to this, the number of servers needed to offer the same service can be reduced and so on the wasted energy. Although peers also would need some energy, they would be running anyway when consuming the service, so the idle times on servers described above are avoided.

From the social point of view, peer-to-peer systems are usually regarded as a threat for economy and a great opportunity for culture spreading. On one hand and due to its use for end-user file sharing, these systems have been and are a perfect tool for piracy. So, they cause less contents (such as movies, music, magazines or books) to be sold and the business is really affected. On the other hand, peer-to-peer systems are a great tool to spread cultural contents amongst thousands of users at very little or no cost.

Considering the output of this project, the deployment of *swift* as a transport protocol could help to reduce the amount of needed servers to provide some services, such as Video on Demand or video streaming. If data could be obtained from *swift* swarms instead of central HTTP servers, the needed platforms to offer these services (mainly large datacenters) could be smaller and so less energy would be consumed.

[1] Uninterruptible Power Supply

6.4. Personal conclusions

After working in this project for five months, I would like to point out the topics helped me to learn and grow as an engineer while working on it.

Thanks to the first theoretical study done over the involved technologies, I learnt how BitTorrent, the new μTP and *swift* protocols worked. The study over the latter gave me the background needed to face the design of the presented solution.

While designing the novel module to integrate *swift* into Tribler, I learnt how to integrate my work in a much bigger software project. Moreover, I used for first time Python and discovered its integration with C and C++ code.

Finally, when I devised and ran the experiments to assess the correct operation of my implementation, I had to develop a methodology to obtain, process and show the obtained data in a clear way, which was a real challenge for me.

BIBLIOGRAPHY

[1] iPoque. iPoque Internet Study 2008/2009, 2009.

[2] Bittorrent Inc. What is BitTorrent?

[3] B. Cohen. The BitTorrent Protocol Specification. BitTorrent Extension Protocol 3, January 2008.

[4] Tit for Tat. Wikipedia.

[5] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6. Citeseer, 2003.

[6] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. Bittorrent is an auction: analyzing and improving bittorrent's incentives. *ACM SIGCOMM Computer Communication Review*, 38(4):243–254, 2008.

[7] J.A. Pouwelse, J. Yang, M. Meulpolder, D.H.J. Epema, and H.J. Sips. BuddyCast: An Operational Peer-to-Peer Epidemic Protocol Stack. Technical report, Delft University of Technology, May 2008.

[8] A. Loewenstern. DHT Protocol. BitTorrent Extension Protocol 5, January 2008.

[9] Micro Transport Protocol. Wikipedia.

[10] A. Norberg. uTorrent transport protocol. BitTorrent Extension Protocol 29, June 2009.

[11] S. Morris. Changing the game with uTP. BitTorrent Blog, October 2009.

[12] S. Morris. Testing uTP - is uTP actually faster than regular BitTorrent? BitTorrent Blog, November 2009.

[13] S. Morris. Visualizing uTP. BitTorrent Blog, November 2009.

[14] S. Shalunov. Low Extra Delay Background Transport. Internet-draft, Internet Engineering Task Force, March 2010. LEDBAT WG.

[15] V. Grishchenko. The Generic Multiparty Transport Protocol (swift). Internet-draft, Internet Engineering Task Force, April 2010. PPSP WG.

[16] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, and R.L. Braynar. Networking Named Content. In *CoNEXT '09: Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12, New York, NY, USA, 2009. ACM.

[17] SHA-1. Wikipedia.

[18] A. Bakker. Merkle hash torrent extension. BitTorrent Extension Protocol 30, August 2009.

[19] Hash Tree. Wikipedia.

[20] Merkle Hashes. Tribler.

[21] V. Grishchenko and J.A. Pouwelse. Binmaps: hybridizing bitmaps and binary trees. 2009.

[22] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference*, pages 179–192, 2005.

[23] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. BarterCast: Fully Distributed Sharing-Ratio Enforcement in BitTorrent. Technical report, Delft University of Technology, 2008.

[24] P. Garbacki, A. Iosup, D. H. J. Epema, and M. van Steen. 2Fast: Collaborative Downloads in P2P Networks. In *6th IEEE International Conference on Peer-to-Peer Computing (P2P2006)*, Cambridge, UK, September 2006.

APPENDIXES

APPENDIX A. *SWIFT* DIALOG EXAMPLE

Figure A.1 shows an example of a dialog from a file transfer using the novel *swift* protocol. In this example, the file is 8 KBytes long, which means it completely fills a 4-layer Merkle tree. Assume *Peer 1* and *Peer 3* already have the complete file, *Peer 2* has bins 5 and 13 (meaning it already acquired bins 4, 6, 12 and 14) and Peer 4 has no pieces yet.

At the beginning, *Peer 1* contacts *Peer 3* and establish channels to communicate. They both announce they already have the whole file (announcing bin 7, which is the root hash bin), so they no need to transfer anything.

Then, *Peer 2* contacts *Peer 3* to obtain the file. After establishing the channels to communicate to each other, they notify each other its available pieces. *Peer 2* then notices *Peer 3* has interesting pieces, so retrieves for bin 0. Peer 3 then sends a datagram with: a *PEX+* message introducing *Peer 1* (who also had all pieces *Peer 2* does not), two *HASH* message needed to check the data integrity and the data itself in a *DATA* message.

Immediately, *Peer 3* sends another *PEX+* message to *Peer 1*, introducing *Peer 2*; *Peer 2* sends an *ACK* message to notify the correct reception of bin 0; and *Peer 2* initiates a channel to communicate with `Peer 1`, notifying one each other about the already acquired pieces (as done before between *Peer 2* and *Peer 3*).

Peer 2 acquires the same way piece 8 from *Peer 1* and piece 10 from *Peer 3*. Notice *Peer 2* sends *HAVE* messages to remote peers when a piece is acquired from another channel.

In the meanwhile, *Peer 4* contacts *Peer 7*: channels are established and *Peer 4* requests for bin 0. *Peer 3* then notice *Peer 4* does not know about the file size, so peak hashes (marked with a * symbol in Figure A.1) are attached to the first datagram, also carrying a `DATA` message for bin 0 and the needed `HASH` messages to check the piece integrity and a `PEX+` message introducing *Peer 2*. The same message sequence for peer introduction detailed earlier is repeated here. *Peer 4* then acquires pieces 0 and 2 from *Peer 3* and piece 4 from *Peer 2*.

Finally, *Peer 4* closes the connection first with *Peer 3* and then *Peer 2*, sending a `HANDSHAKE` message with channel identifier to 0. When *Peer 3* receives the closing message from *Peer 4*, a `PEX-` message is sent to *Peer 2*, who was previously introduced to *Peer 4*.

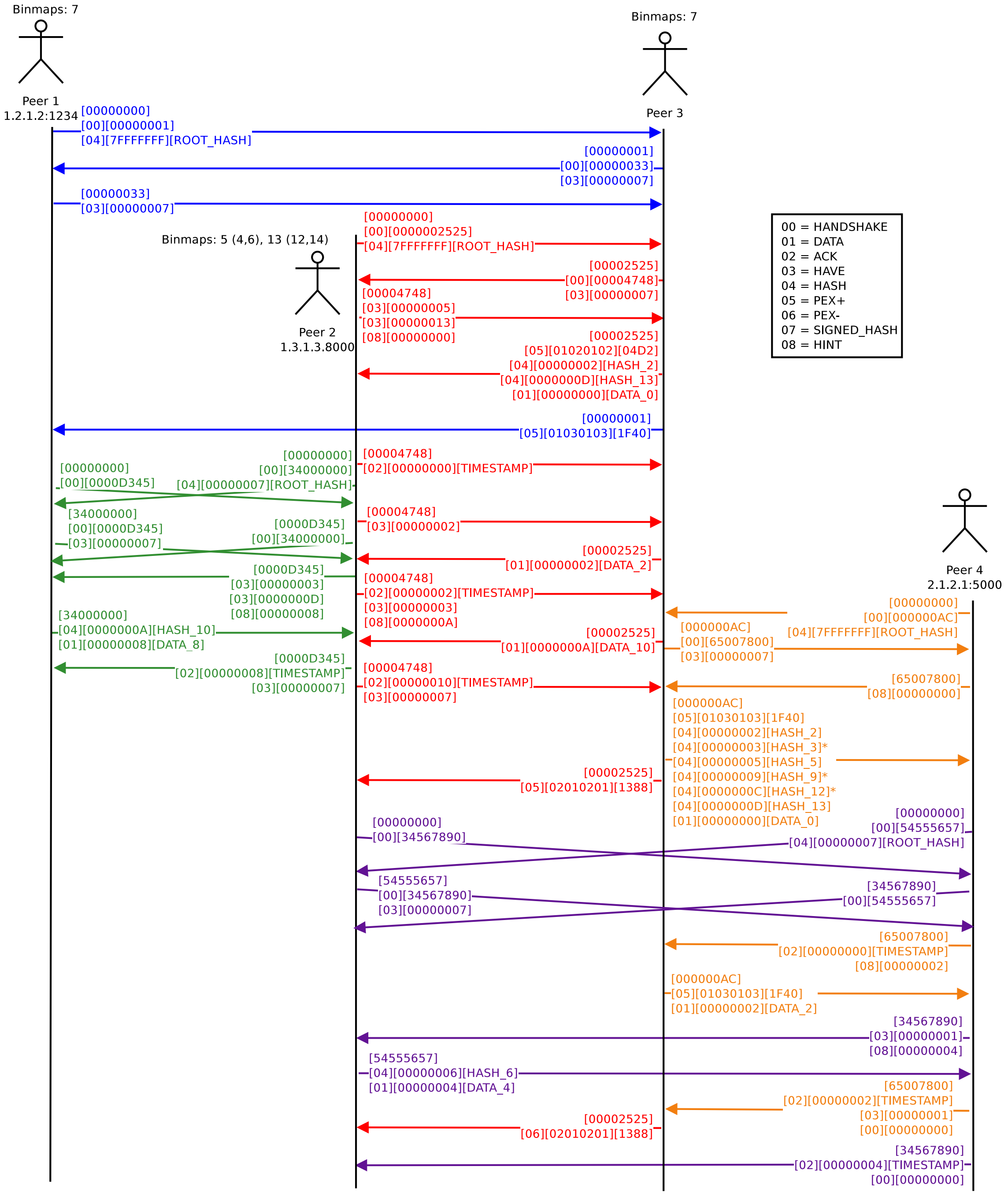


Figure A.1: *swift* dialog example

APPENDIX B. USING SWIG TO BUILD THE LIBSWIFT PYTHON MODULE

This appendix explains how to generate the Python wrapper for Libswift and compile the library to be used from Python code.

The first step is to generate the actual C++ wrapper and the Python module to call the C++ methods in Libswift . Considering the following `swift.i` file:

```
%module swift
%{
#include"pswift.h"
%}

%apply int { int8_t }
%apply unsigned int { uint8_t }
%apply int { int16_t }
%apply unsigned int { uint16_t }
%apply int { int32_t }
%apply unsigned int { uint32_t }
%apply long long { int64_t }
%apply unsigned long long { uint64_t }

%include"pswift.h"

%{
#include "swift.h"

PyObject* python_callback = 0;

void api_callback(int transfer, bin64_t bin) {
PyObject *arglist;
PyObject *result;
PyGILState_STATE gstate;

if (!PyCallable_Check(python_callback)) {
PyErr_SetString(PyExc_TypeError, "Python callback must be callable");
return;
}
arglist = Py_BuildValue("(il)", transfer, bin.v);

gstate = PyGILState_Ensure();

result = PyEval_CallObject(python_callback, arglist);

Py_XDECREF(arglist);
Py_XDECREF(result);

PyGILState_Release(gstate);
}

void AddProgressCallback(int transfer, uint8_t agg, PyObject *pyfunc) {
python_callback = pyfunc;
swift::AddProgressCallback(transfer,api_callback, agg);
Py_INCREF(pyfunc);
}

void RemoveProgressCallback(int transfer,PyObject *pyfunc) {
python_callback = pyfunc;
swift::RemoveProgressCallback(transfer,api_callback);
Py_INCREF(pyfunc);
}
%}

void AddProgressCallback(int transfer,uint8_t agg, PyObject*);
void RemoveProgressCallback (int transfer,PyObject*);
```

The command to run would be:

```
swig -i pswift.i
```

Then, the wrapper should be compiled with all the Libswift code to generate a compiled shared library for the current environment. Best way to do so is using the `setup` tools in Python, which easily compiles the library and the generated wrapper. Consider the following `setup.py` file:

```
from distutils.core import setup, Extension
import os
os.environ['CC'] = 'g++'
setup(name="swift",
     version='0.1',
     description='Python wrapper for libswift',
     author='gca',
     author_email='guillemcabrera@gmail.com',
     url='http://www.libswift.org',
     py_modules=['swift'],
     ext_modules=[Extension("_swift",
          ["swift.i","pswift.cpp","swift.cpp","sha1.cpp",
          "compat.cpp","sendrecv.cpp","send_control.cpp",
          "hashtree.cpp","bin64.cpp","bins.cpp","channel.cpp",
          "datagram.cpp","transfer.cpp","httpgw.cpp"],
          swig_opts=['-c++'],
     )],
)
```

Finally, running the following line to actually compile the library ready to be called from the *swift* Python module:

```
python setup.py
```

A working C/C++ compiler and a Python interpreter must be present in the environment where running this command.