Rewarding Good Behavior in Peer-to-Peer Networks

Vlad Dumitrescu

Delft University of Technology

Rewarding Good Behavior in Peer-to-peer Networks

Master's Thesis in Computer Engineering

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Vlad Dumitrescu

19th September 2013

Author
Vlad Dumitrescu

Title
Rewarding Good Behavior in Peer-to-Peer Networks

MSc presentation
September 24th, 2013

Graduation Committee

Abstract

Large-scale human cooperation is vital to society. Designers and community facilitators of peer-to-peer networks are constantly investigating free-riding prevention measures for their platforms. Such schemes of rewarding good behavior remain a challenging problem in fully decentralized P2P networks.

This work is a first attempt to integrate cutting-edge technologies developed by the Tribler team into a functional reputation-driven P2P network. We start by evaluating the possibilities of our system in supporting generous donations, and we continue with the issue of enforcing preferential treatment by implementing a reciprocity mechanism in Libswift, the reference implementation of the upcoming IETF Peer-to-Peer Streaming Internet Standard.

Additionally, we use and evaluate solutions developed during this project in two experiments, followed by a third one whose goal is to demonstrate the concept of indirect reciprocity using the fully integrated system. Aiming to open the road for further research, we conclude with initial insights into the behavior of the involved components, alongside with a number of potential weaknesses.

Preface

This is the story of my confrontation with Science, which started almost 11 months ago. It seems we are almost there, at the end. Only one more step is needed, but my mission has ended. It is your turn, the Mighty Reader. For me, this project started with ambitious ideas and then it soon turned into a quite challenging task. I really enjoyed working on it, but I must say it was also frustrating at times.

Peer-to-peer networks were researched throughly during the past years, but production level solutions for the most daunting issues are still to surface. The Tribler team does an awesome job at balancing real-world development with their cutting-edge research. I hope this proof-of-concept study will be the basis for the following development efforts. I would be glad to see it culminating in a fully-fledged reputation-based Tribler version as soon as possible.

I would like to thank Johan Pouwelse, my supervisor, who was always available for crunching ideas and discussing issues in pretty long meetings. His enthusiasm and dedication towards Tribler motivated me a lot during this period. Without Elric, Riccardo, Boudewijn, Niels, Dimitra, and Arno, who always made themselves available to answer my questions, I would still be fighting the Tribler's code trenches against fearless warriors: Python, C++, and Bash. On the other side, Cor-Paul Bezemer offered his help with reviewing the history of this confrontation, which you will start reading in a few moments. Thank you all a lot, and good luck with your research endeavors!

I met a lot of smart and passionate people during my 2 years in Delft, and I was lucky to tie friendships with some of them. Thank you for being great companions! Also, I would like to thank my old friends from Romania who sometimes supported me virtually.

Finally, this learning experience in Delft would not have been possible without support from my parents. I would like to dedicate this work to them for their unconditional trust, approval, help, and love.

Vlad Dumitrescu
Delft, The Netherlands
19th September 2013

Contents

Chapter 1

Introduction

Over the last 15 years, peer-to-peer (P2P) technologies secured their place as one of the most important developments in the consumer Internet landscape. Their popularity explosion started in 1999, when the Napster [1] file-sharing system was launched. Examples of other popular networks that followed include: Gnutella [24], Direct-Connect, eDonkey, and FastTrack [15] (better known under the name of KaZaA [13] – a popular client). Currently, most of these are closed down, but BitTorrent [2] took the lead as the most used network with over 150 million monthly active users as of January 2012 [3]. P2P networks are being used for file-sharing, distribution of software updates, video & audio streaming [10, 14], communications (e.g., Skype), data caching (e.g., Content Distribution Networks), server-less website hosting, decentralized search, or digital currency.

The Internet Study 2008/2009 [11] conducted by ipoque reported P2P file-sharing traffic[1] accounting between 43% and 70%, depending on the globe's region, of the total Internet traffic. These results mark the beginning of a decreasing trend in traffic, which peaked in 2007. Despite this, P2P networks remain highly popular, and we believe they will constitute a major technology in the "Video Internet". For the 2012-2017 period, Cisco predicts a stabilized contribution of file sharing[2] in the total consumer Internet traffic [5]. Alongside these numbers, we should also note that classifying P2P traffic is becoming increasingly difficult (e.g., by encrypting the protocols), countering the efforts made by Internet Service Providers towards detecting, throttling, or even blocking (see Hart v. Comcast[3]), this kind of traffic.

While some might foresee the death of P2P networks, the latter-day attempts to legislate, control, and monitor the Internet's traffic might turn our eyes back to old pledges of this technology, such as privacy and decentralized reputation. These are still quite far from a production level quality and are not being used in popular im-

[1]The study measured the following networks: Ares, BitTorrent, eDonkey, and Gnutella.

[2]Note: this also includes file sharing traffic from client-server services (e.g., Megaupload-like sites).

[3]`http://www.wired.com/threatlevel/2007/11/comcast-sued-ov/`

plementations. Additionally, while most of the other today's Internet technologies (e.g., TCP/IP) have benefited from a through analysis and standardization process, the P2P realm still hosts a lot if different solutions and implementations tailored to each specific application.

Different from the traditional client-server model, in P2P networks all participants have equal responsibilities – they act both as a client and as a server. Ideally, this creates a decentralized network, which leads to advantages when it comes to scalability and low-cost barrier for smaller content distributors. Unfortunately, disadvantages also arise due to coordination issues (e.g., searching becomes difficult in a decentralized system), and low upload speeds available to normal Internet users. Also, the health of the network is completely in the hands of its users, which have to store and re-share the information they consume, attempting to avoid the *tragedy of the commons* [9]. Implementations usually alleviate these challenges by trading off some aspects – the use a central server to keep a search index, and track the reputation of each user in order to incentivize sharing.

Briefly, this work contributes to the attempt of the Tribler team[4] to better understand and find solutions for the long-standing issue of free-riding. The goal is to build a first demonstration of rewarding good behavior in Tribler by integrating the latest decentralized reputation tracking mechanism, BarterCast3, and implementing a preferential treatment (reciprocity) mechanism in Libswift. Additionally, we also focus on enabling generous donations.

Before detailing our research questions and goals, in Chapter 2, this chapter will continue with an overview of BitTorrent, as an example of a P2P network, and background information on different technologies that play a role in our project. Following the problem description, Chapter 3 covers the contributions made towards enabling high donations, which we consider one of the foundations for the envisioned fully decentralized reputation-driven system. Chapter 4 details our approach towards rewarding goodness, while Chapter 5 presents and discusses a set of initial experiments that were able to build. Finally, Chapter 6 draws some conclusions and brainstorms on a few directions of future work.

1.1 A P2P Network: BitTorrent

In a BitTorrent (BT) network all participants, called *peers*, collectively support the distribution of files by downloading and uploading content to and from others, with no central server being involved. The term *swarm* is used to describe the group of peers that download or upload parts of a specific file or a logical set of files. The peers that provide content (upload) are called *seeders*, while the ones that consume the content (download) are called *leechers*. A swarm's content is split into small individual *pieces* which are transmitted from seeders to leechers. When a leecher completely downloads a piece it can further seed it to other swarm

[4]http://www.tribler.org/

peers. In general, the network of peers and connections to their neighbors that forms as a result of these interactions is called a *P2P overlay*.

To initiate the process, also referred to as *joining a swarm*, a new leecher or seeder needs some information about the content that he is going to download or, respectively, upload. This information is published by the initial content seeder in a small *torrent* file which includes: content file names, folder structure, total size, piece size, a list of cryptographic hashes for each piece, and, optionally, a list of trackers. The torrent file can also contain other information specific to BitTorrent extensions. The cryptographic hashes are used to verify the actual content that will be received via the P2P network, thus protecting leeching peers from obtaining maliciously modified, or damaged data. Note that the original torrent file, containing the hashes, still needs to be obtained from a trusted source. Once the torrent file is obtained, the leecher can query the listed *trackers* for addresses of other peers in the swarm. Since the tracker is still a centralized component, an alternative is to use a Distributed Hash Table [20] for this operation.

A main drawback of the BT protocol is torrent file distribution, which is still being done in the traditional client-server way, by using a centralized website. Since shutting down torrent websites can effectively cripple the BT communities formed around them, efforts are being made to also distribute the torrent files in a decentralized manner. However, this becomes a circular chicken-or-the-egg problem which cannot be definitively solved.

1.2 Background: The Tribler Project

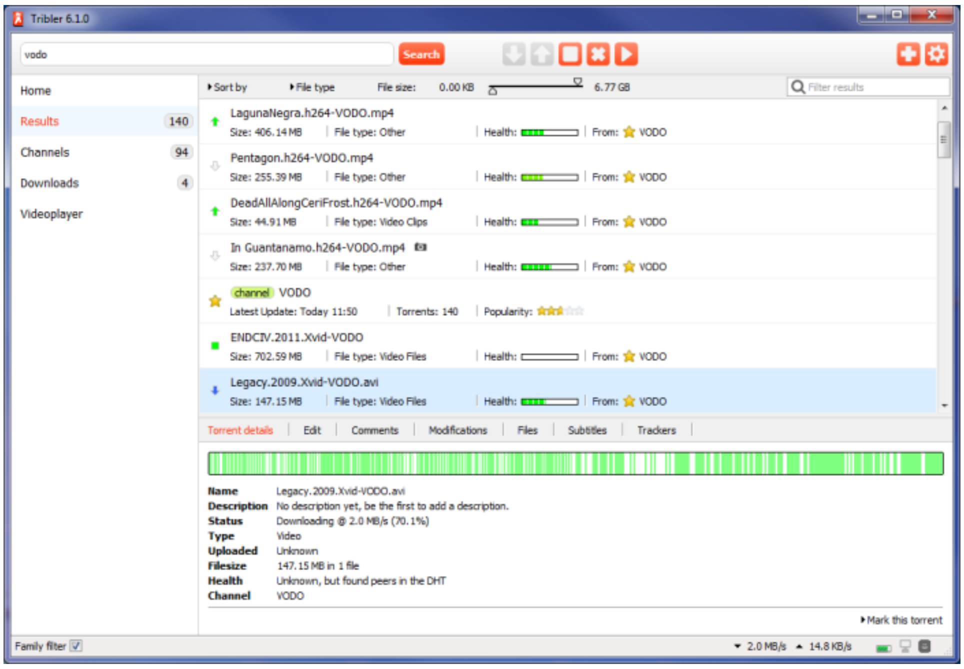
The Tribler project is a BitTorrent-compatible P2P client used within this research. We start with a short overview of the whole Tribler application and continue with the relevant sub-projects: Libswift, Dispersy, BarterCast3 Community, and Gumby. Interested parties can further refer to the documentation and development efforts published as open-source projects on GitHub[5].

1.2.1 Tribler

Tribler is an open-source P2P client started in 2005 at Delft University of Technology, which builds upon the original BitTorrent with novel features such as video streaming, decentralized searching, channels, and social-like features (e.g., users with similar tastes, voting system). Detailed information about all features are given by Zeilemaker et al. [29]. Recent statistics[6] counted approximately 1.47 million unique users since 2006. The Tribler network also acts as a real-world testbench for cutting-edge P2P research projects and experiments run by TU Delft and its various partners.

[5]https://github.com/tribler

[6]http://statistics.tribler.org/



LagunaNegra.h264-VODO.mp4
Size: 406.14 MB | File type: Other | Health: | From: VODO

Pentagon.h264-VODO.mp4
Size: 255.39 MB | File type: Other | Health: | From: VODO

DeadAllAlongCeriFrost.h264-VODO.mp4
Size: 44.91 MB | File type: Video Clips | Health: | From: VODO

In Guantanamo.h264-VODO.mp4
Size: 237.70 MB | File type: Other | Health: | From: VODO

channel VODO
Latest Update: Today 11:50 | Torrents: 140 | Popularity:

ENDCIV.2011.Xvid-VODO
Size: 702.59 MB | File type: Video Files | Health: | From: VODO

Legacy.2009.Xvid-VODO.avi
Size: 147.15 MB | File type: Video Files | Health: | From: VODO

Name	Legacy.2009.Xvid-VODO.avi
Description	No description yet, be the first to add a description.
Status	Downloading @ 2.0 MB/s (70.1%)
Type	Video
Uploaded	Unknown
Filesize	147.15 MB in 1 file
Health	Unknown, but found peers in the DHT
Channel	VODO

Figure 1.1: Interface of a recent Tribler version showing results of its decentralized search.

Cryptographic public-keys, identifying each peer, form the foundation of Tribler's secure framework, on top of which most extensions are built. This public-key-based framework, which is not present in the original BT, offers the possibility to build complementary P2P overlays for metadata dissemination. Initially, the BuddyCast [23] overlay handled peer discovery and preference data exchange, while BarterCast was built on top of it and disseminated reputation information. Users could publish their favorite torrent files in so-called *channels*, and the ChannelCast overlay was responsible for discovering and sorting these channels. Interesting torrent files were also prefetched from users with similar tastes discovered by BuddyCast, enabling efficient remote search (Figure 1.1). These features are still present in today's Tribler, but they are now based on Dispersy – a generalized data dissemination solution which is further described in Section 1.2.3. As another major extension, Tribler implements piece picking policies, alternative to the BT's rarest-first, suited for video-on-demand and streaming.

The client is also ported to the NextShare TV [28] set-top box produced by Pioneer, and as a browser plug-in called SwarmPlayer[7] which can be used to embed P2P-backed videos in web pages.

[7]`http://swarmplayer.p2p-next.org/`

1.2.2 Libswift & the Peer-to-Peer Streaming Protocol

Besides BitTorrent, Tribler also supports the new Libswift [21] download engine, though, at this moment, it is mainly used for collecting torrent files to support the decentralized search system. It is also possible to re-seed BT content using Libswift, thus real-content usage will increase as more data is going to be injected.

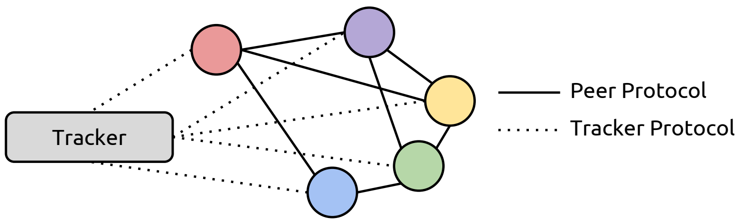


Figure 1.2: PPSP architecture showing the 2 sub-protocols for peer-to-peer and peer-to-tracker communication. Note that the tracker is a logical component that can also function decentralized.

Libswift is the reference implementation of the IETF proposed Internet Standard, the Peer-to-Peer Streaming Protocol[8] (PPSP), a standardization effort to which the Tribler team is also contributing. The scope, requirements and use cases of PPSP are detailed in the Informational RFC 6972 [31]. Figure 1.2 shows the general architecture of the system with the 2 sub-protocols: a peer-protocol (PPSPP) and a tracker-protocol (PPSP-TP). Briefly, the proposal attempts to unify, in the dawn of the ”Video Internet”, the multitude of proprietary P2P protocols that have been developed during the past years. The main tasks and design issues addressed by the standard are:

- signaling standardization for both live and video-on-demand streaming
- support both centralized and distributed trackers
- loosely coupled peer-to-tracker and peer-to-peer protocols
- tracker protocol: peer & content identification, peer properties discovery (e.g., NAT, IPv4/IPv6), light-weight design, peer list optimization (i.e., by tracking content & peers status)
- peer protocol: global content identification & verification, light-weight chunk availability representation, protocol efficiency (i.e., metadata overhead)

Research efforts focusing on video streaming use cases are guided by forecasts indicating a 3-fold increase in consumer IP video traffic [5] by 2017, but we should also note that the traditional file-sharing applications are also supported by the

[8]These IETF protocols are still in Internet-Draft stage. The updated version can be found at `http://datatracker.ietf.org/wg/ppsp/`. This work refers to revision 07 of PPSPP.

standard. A recent example of such an application, which is also using Libswift, is Teamshare[9].

1.2.3 Dispersy

Dispersy [30] is a data dissemination middleware designed to be used in challenged network environments (i.e., erratic connectivity, high latency, or unavailable end-to-end paths). It provides a node discovery mechanism that also embed a NAT-puncturing solution. The originating peer sends an *introduction request* to an already connected neighbor, that, in turn, will send a *puncture request* to a new *candidate* peer and a *introduction reply* back to the source, thus enabling the NAT-puncture and the connection between the two. This system can be bootstrapped using always connected *tracker* peers, a few of which are maintained by the Tribler team. To avoid *eclipse attacks*[10], the candidate peers are assigned to disjoint categories with predefined selection probabilities; one category is composed of only trusted peers and has a selection probability of 1%.

Dispersy splits data in *bundles* (or messages) and synchronizes these between peers by first exchanging Bloom filters[11] followed by the actual missing data. All transfers are implemented over UDP, which simplifies the NAT-puncturing mechanism, but limits the size of the Bloom filters as fragmentation has to be avoided. This requires Dispersy to only synchronize a subset of the available bundles, which is established heuristically.

Applications can use Dispersy by defining a *community* which specifies structure and encoding for different messages, authentication requirements, dissemination policy (e.g., full dissemination), permissions for different peers, and candidate peer list management. Communities are identified by unique cryptographic public keys, which are generated at *creation* time and need to be known by the *joining* peers. The communities used in Tribler are:

- Channel: Users can start their own channels advertising favorite torrent files.
- AllChannel: Since Channel communities can be created by anyone, their existence and meta-information (i.e., public key) needs to be disseminated via this community. It also handles voting and channel discovery.
- Search: This community collects torrent files and provides decentralized search.
- PrivateSearch: Provides anonymous searching (under-development).
- BarterCast3: Disseminates reputation information.

[9] `https://github.com/open-software-solutions/Teamshare`

[10] The victim of a eclipse attack will only receive information from the attacker. For example, it will only get introduced to other malicious peers.

[11] Bloom filters are used to efficiently test set-membership of an element.

1.2.4 BarterCast

BarterCast is a protocol, used by Tribler to disseminate information about the users' activity in order to combat free-riding. The first version of the BarterCast [18] worked by gossiping upload and download statistics of each peer into the network. Using these, a peer can build a partial view of the network in the form of a directed graph where the edges are labeled with traffic volumes between 2 peers. Since peers can report false information, BarterCast used a maxflow-based subjective reputation as described in [7]. Because this algorithm is very expensive, only paths with a length of 2 were considered. To incentivize seeding, the BarterCast's reputation values were used in combination with the original tit-for-tat reciprocity mechanism defined by the BitTorrent protocol. This way, peers with long-term contributions will get higher priorities when downloading.

Version 3 of the protocol is currently under active development and it is implemented as a Dispersy community. Moreover, it also adds *user interaction strength estimation* metrics into the equation, as introduced by Jia et al. [12], and other improvements. We will discuss more about reputation, reciprocity, and improvements to BarterCast in the following chapters.

1.2.5 Gumby

Gumby is a testing framework, designed for Tribler and Dispersy, under early development at the time of this writing. Its ultimate goal is to unify various existing experimentation frameworks and other components that independently developed in the Tribler team. The required features include:

- automatic environment configuration with dependencies management
- convenient scenario specification using a Domain Specific Language
- local, remote, and DAS-4[12] deployment
- support for experiments with peers located in different clusters (i.e., on DAS-4, local, and other remote machines)
- runtime peer configuration server and Dispersy tracker
- support for SystemTap[13] instrumentation and other resource usage measurement solutions
- support for running on Jenkins[14]
- log collection, post-processing, and graph generation

The integration experiment described at the end of Chapter 5 served as one of the first case studies for this framework.

[12]`http://www.cs.vu.nl/das4/`
[13]`http://sourceware.org/systemtap/`
[14]`http://jenkins-ci.org/`

Chapter 2

Problem Description

Cooperation in a peer-to-peer networks is essential, and free-riding cannot be tolerated. Free-riders are participants that refuse to contribute back to the community after they obtained their benefit. Unfortunately, the selfish nature of humans do not allow simple solutions for this problem. Protocol designers and community facilitators are obliged to come up with schemes that provide incentives for participants to act in the interest of their group. The need for such schemes imposes limitations on the level of decentralization the networks can have. Current solutions use central entities to track activity levels of the participants and assign reputations.

The ultimate goal in fighting free-riding is to create a fully decentralized, self-organizing, and reputation-governed P2P network. Achieving this has proved highly challenging, with no substantial progress towards a production-level solution. This motivates our desire to build tools and infrastructure that can be used to empirically test, analyze, and evaluate different approaches in an environment as close as possible to real-world. While different components of such a system have been studied thoroughly in the past, a fully functional implementation is yet to be proven.

2.1 Research Problem

The term *goodness* quantifies the level of cooperation that peers exhibit towards sustaining the network. Specifically, this can be measured in different ways such as download/upload ratio, or on-line time. Once the system is able to provide reliable values, we can impose penalties for the free-riding peers, while offering preferential treatment for (or *rewarding*) those that are willing to share their resources with the community (i.e., by storing and uploading content).

The broader problem of rewarding good behavior poses a number of challenges. The first one is *expressing goodness*, or what kind of information should we track for each user. Solutions could be based on the traditional upload/download ratio, or on-line time (which emphasizes willingness to cooperate rather than actual activity). The next question is how much *accounting accuracy* do we require considering we will probably use heuristic algorithms to compute the reputation since they are, most likely, expensive. This is also connected to the problem if *information dissemination*. While a central tracker enables maximum accuracy and full information availability, in a decentralized algorithm we need to consider propagation speed, coverage, overhead, and how these affect scalability. Finally, each peer should rank its barter partners and enforce *preferential treatment* based on their goodness.

If all these problems are solved and successfully integrated in Tribler, be believe it will incentivize our users to donate their resources and benefit from their improving status. Because of this, we need to guarantee that the system will support *generous donation* of resources. For example, similar improvements and tests have been conducted by *libtorrent* developers [16].

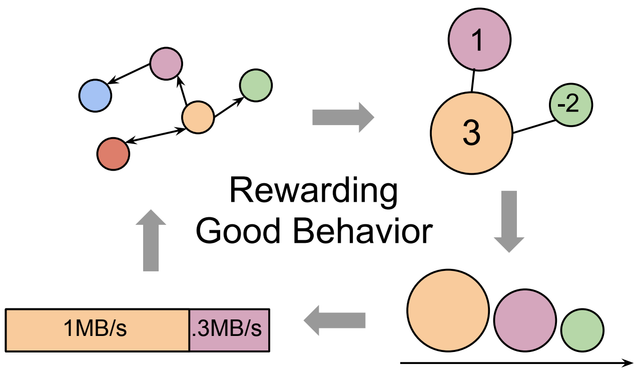


Figure 2.1: The approach taken by Tribler can be seen as a 4-steps loop: activity tracking and dissemination, local reputation computation, ranking service requesters, and prioritizing resources.

Figure 2.1 shows the actual steps that build up the whole process in Tribler. Goodness metrics are tracked and disseminated into the network, followed by each participant running a local reputation algorithm which results in a ranking of its barter partners. This ends with applying a *reciprocity* policy that will divide available upload resources, which, in turn, will affect its peers' activity levels, and the steps are reiterated. The implementations of these steps can be interchanged and tuned in order to study the sub-problems mentioned above.

A number of components and algorithms have been developed and studied independently [17, 12, 8, 22], but we require a reliable platform for integrated evaluation, as close as possible to real-world conditions. The goal of this work is to study the possibilities of such integrated experiments using Tribler. Firstly, we need to build the tools that will enable thorough testing and evaluation of Libswift capabilities to handle large amounts of data. Secondly, we seek to build, demonstrate and analyze solutions for rewarding good behavior. And thirdly, we need to prepare experimental setups and scenarios targeting each involved components, culminating with a unified experiment.

2.2 Research Questions

How can we evaluate Libswift under heavy load, in a controlled testing environment? We need to test Libswift with both very large files. Previous attempts [26] are limited by the start-up overhead imposed by other variables that we are not interested to test (e.g., storage layer speed and space). The tools that will enable this should pose minimal impact on performance of Libswift and should not interfere with the implementation.

How can we enforce preferential treatment and how can we study different reciprocity policies? There is no existing analysis and implementation of preferential treatment solutions for Libswift. To complete the integration efforts we also need to design and implement this feature, and make an initial assessment of possible policies.

How can we evaluate fully integrated version of Tribler with all components required for rewarding good behavior? The final goal is to provide a base for integration experiments which can be run in clustered environments, with, possibly, large number of real Tribler instances.

Chapter 3

Enabling Generous Donations

An important aspect of our envisioned reputation-based fully-decentralized P2P system is the ability to upload and download extremely high amounts of data. The ultimate goal is to allow users to *donate* their resources – processor cycles, storage, and bandwidth – such that they can maintain or increase their reputation, which will further increase their service level. This challenging problem has also been addressed by other P2P projects [16]. To support this goal, Tribler needs tools that can be used to asses the performance of its new algorithms and implementations.

Previous attempts to measure the performance of Libswift [26] were limited by the constraints imposed by the normal filesystems. Specifically, it generates files at the beginning of the test, and copies them to each node in case checks for transfer correctness are required. This also means that hashes needed for content integrity check are computed each time a file is generated – a time consuming operation. To work around those issues in our testing environment, we need to create very large mocked files, in the order of hundreds of GB, and large collections of files. These will be seeded or leeched by Libswift during experimental scenarios. The main requirement of such a tool is minimal delay for setting up the test files, such that they can be used with our continuous integration[1] system. Additionally, the mocking capability should have minimal impact on Libswift's code base.

The approach chosen consists of a virtual, in-memory, filesystem built using the Filesystem in Userspace (FUSE)[2] library. This will enables us to generate predefined content on-the-fly and skip the hashing part each time a test is run by maintaining precomputed hashes for the generated content.

The next section will give a short overview of the FUSE library, after which we will continue with design and implementation details of out virtual filesystem, followed by some usage examples, and a performance benchmark. The virtual filesystem is used in the first experiment described in Chapter 5.

[1]Besides running unit tests, the Tribler continuous integration setup also runs experiments. These are required because, in this case, the performance of algorithms used is equally important to having bug-free code.

[2]`http://fuse.sourceforge.net/`

3.1 Filesystem in Userspace

The FUSE mechanism facilitates the development of new filesystems for Unix-like operating systems that can be mounted directly by users, without requiring root privileges. The usual approach (i.e., not using FUSE) is to insert the code of the new filesystem into the kernel, as a module – this operation requires administrator privileges. While this approach may be feasible if you control the testing machines, the FUSE framework also simplifies development a lot, while not imposing overheads, as we will show in Section 3.5.

FUSE is composed of a kernel module, and a library (both depicted with green in Figure 3.1). The kernel module needs to be inserted once into the kernel by the administrator, and it will be used by all FUSE filesystems launched on the system. These are implemented as normal binaries that link against *libfuse*, and that can be run in userspace, resulting in a mounted filesystem. An usage example specific to our virtual filesystem is given in Section 3.4.

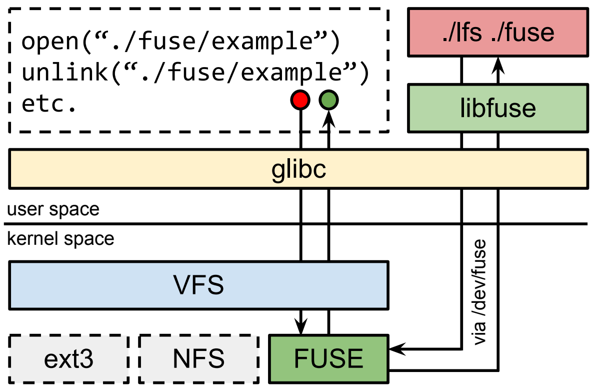


Figure 3.1: FUSE components and syscall path.[3]

As Figure 3.1 shows, a file operation goes through the common path, via the C Standard Library, that makes a syscall handled by the Virtual Filesystem Switch (VFS) layer in the kernel, and reaches the FUSE module. Next, instead of accessing a physical device, the kernel module will direct the request to the corresponding FUSE userspace process. The connection between the userspace process (via libfuse) and the kernel module goes through a special device file, `/dev/fuse`. Section 3.3 provides details on how a filesystem can be implemented, focusing on our case.

FUSE was merged in the mainline Linux kernel since version 2.6.14 (2005) and it is also available in other Unix-like operating systems: OS X, FreeBSD, NetBSD, OpenSolaris, Minix 3, and Android.

[3]Figure adapted from the FUSE documentation.

3.2 Design

As previously indicated, our virtual, in-memory, filesystem, named Lazy File System (LFS), will use the FUSE library and must meet the following key requirements:

- minimal setup time of test files
- minimal influence on the Libswift code base
- support extremely large files
- reads should return predefined content, writes can be ignored
- support precomputed Libswift meta files (i.e.,`.mbinmap` & `.mhash`)

The last two requirements are motivated by the content integrity mechanism used in Libswift, which computes a tree of hashes starting from the each content chunk. During normal operation (i.e., exchanging chunks in the P2P network), this CPU-intensive task does not represent an issue since the network bandwidth is the bottleneck anyway. It might be a issue when adding a new file, but this is a on-time operation. Unfortunately, it is unacceptable to run the hashing every time we start our high-donations tests. For example, we measured the required time for this computation, using a 128GB file using 8K chunks (around 33.5 million SHA-1 hashes), to around 30-40 minutes on an Intel Core 2 Duo CPU running at 2.10GHz. In conclusion, we need to store pre-computed hashes which, in turn, means our filesystem should return predefined content.

The files in LFS are virtual, there is no actual storage. Each file has a corresponding 4-bytes *pattern*, stored in memory, which is cycled in a round-robin fashion each time a byte is read. To conveniently set this pattern, when a file is created, its name needs to start with the 4-byte pattern as a lowercase hexadecimal string (e.g., `deadb3af_testfile`). The file can be later renamed to anything. Writes to any file do not store any information, but they do increase the size (if applicable). Deviating from these rules, operations on Libswift meta files (i.e., that end with ".mhash" or ".mbinmap") are forwarded to a directory on the normal filesystem, specified by the user at mount time. File size is set via the truncate operation and is stored as a 8-byte unsigned int.

Since this acts as a normal filesystem, Libswift does not need any further modification. The metadata files can be persistently stored on the normal filesystem, and used each time LFS is mounted. For example, the metadata for the 128GB file previously mentioned takes up around 640MB, and around 5.5MB if archived.

3.3 Implementation

To implement a new FUSE filesystem, developers are required to define functions for the common file operations that are going to be supported (e.g., `open()`, read(), write() etc.). The pointers to these functions should be stored in a struct fuse_operations which is then passed to the *libfuse* initialization methods. After the initialization phase, the FUSE filesystem can be started by calling fuse_loop() which enters a dispatcher loop and sends the process to background. The FUSE system will redirect all file syscalls to be handled by these functions. Files are identified by their path, relative to the FUSE mount point, which is passed as the first parameter to all functions. The FUSE documentation contains specific details about the API.

The main operations implemented by LFS and the required data structures used are described below. Note that LFS does not follow the complete specification of a proper file system. We only implemented functionalities required for our testing scenarios (e.g., we do not focus on respecting flags, file modes, access control, permissions etc.). While not particularly mentioned below, all these functions handle Libswift's metadata files differently by forwarding the operation to the backing real filesystem – an example can be seen in Listing 2 which shows the read() operation.

create() Allocates a new struct l_file (see Listing 1), which represents the new file in LFS, and adds it to a hash table[4] indexed by the file path string. This operations also parses the first 8 characters of the filename, as a 4-byte number in hexadecimal form, and sets l_file.pattern accordingly.

```
struct l_file {
    char path[MAXPATHLEN];
    off_t size;
    int fd;
    int realfd; /* if stored on real fs */
    char pattern[4];
    UT_hash_handle hh;
};
```

Listing 1: The main LFS data structure representing a file.

open() Assigns a unique file descriptor number to l_file.fd, which logically signifies that the file is opened in LFS. In case the file is a Libswift metadata one, the actual file descriptor returned by the operating system is stored in l_file.realfd. If the O_TRUNC flags is set, the size of the file is zeroed.

truncate() Sets l_file.size, which represents the size of the virtual file.

readdir() Iterates the hash table and calls *libfuse* filler() function for each entry. Note that LFS does not support directories, but users can mount it multiple times, in different directories to achieve the same effect.

[4]LFS uses uthash: http://troydhanson.github.io/uthash/

`read()` For non-metadata files, the read operation cyclically copies bytes from the 4-byte pattern, which was previously set in the `create()` function, to the FUSE output buffer. The full implementation is given in Listing 2 as an example.

`write()` Makes the file act as `/dev/null`, but it does increase the size if `l_file.size` is not already large enough.

```
int l_read(const char *path, char *buf, size_t size,
           off_t offset, struct fuse_file_info *fi)
{
    struct l_file *file;

    /* search the hash table */
    HASH_FIND_STR(l_data.files, path, file);
    if (file == NULL)
        return -ENOENT;

    if (is_meta_file(path)) {
        /* delegate to real fs */
        return pread(file->realfd, buf, size, offset);
    } else {
        int i, j;
        for (i = 0; i < size; i++) {
            if (offset + i >= file->size) {
                /* fill remaining buffer with 0s */
                for (j = i; j < size; j++)
                    buf[j] = 0x00;

                /* return bytes read so far */
                return i;
            } else {
                buf[i] = file->pattern[(offset + i) % 4];
            }
        }
    }
    return size;
}
```

Listing 2: Implementation of the LFS `read()` operation. Other functions follow the same steps: search for the file in the hash table, handle metadata files differently, the actual operation.

`rename()` Deletes a hash table entry and re-adds it with a changed path string. All other `l_file` fields of the file are preserved.

`getattr()` Returns predefined values for the attributes. This is not needed for our use cases (i.e., reading and writing the file), but other Unix tools (e.g., `ls`) will query these attributes.

`unlink()` Removes the corresponding `struct l_file` from the hash table, and frees the memory.

`release()` For metadata files, this forwards the `close()` syscall. There is nothing that needs to be done when a virtual file is closed.

Interested readers can further refer to the LFS's GitHub page and repository[5] for the complete Lazy File System's source code.

3.4 Usage

This section explains the features of LFS and how can they be used in Libswift experiment by going through some basic examples. We start by mounting the files system.

```
$ mkdir -p /path/to/mountpoint
$ lfs -o realstore=/path/to/real/fs /path/to/mountpoint
Libswift metadir: /path/to/real/fs
```

The `realstore` option is specific to LFS and should point to a directory on a real, hard-drive backed, filesystem. That path will be used to store the Libswift metadata files. Another option specific to LFS is `logfile`, which can optionally point to a debugging log file, also on a real filesystem. There are also a lot more options provided by the FUSE library – run `lfs --help` to list them.

Next, we will create a 1 TB file in our new virtual filesystem, mounted in `/path/to/mountpoint`.

```
$ truncate -s 1TiB /path/to/mountpoint/deadb3af_example
```

All operations on files ending with `.mhash` or `.mbinmap` are forwarded to the real filesystem.

```
$ echo "test" > /path/to/mountpoint/deadb3af_example.mbinmap
$ echo "test" > /path/to/mountpoint/deadb3af_example.mhash
```

Lets check has happened so far. We will see our 3 files on the virtual filesystem, and the 2 metadata files on the real filesystem as well.

```
$ ls -AGgh /path/to/mountpoint/
total 1.1T
-rwxrwxrwx 1 1.0T Sep  9 09:59 deadb3af_example
-rw-rw-r-- 1    2 Sep  9 09:58 deadb3af_example.mbinmap
-rw-rw-r-- 1    2 Sep  9 09:55 deadb3af_example.mhash
$ ls -AGgh /path/to/real/fs
-rw-rw-r-- 1    2 Sep  9 09:58 deadb3af_example.mbinmap
-rw-rw-r-- 1    2 Sep  9 09:55 deadb3af_example.mhash
```

[5] `http://github.com/vladum/lfs-libswift/`

Lets also see the predefined pattern being returned when the virtual file is being read.

```
$ hexdump -e '/1 "%02X "' -n 16 deadb3af_example
DE AD B3 AF DE AD B3 AF DE AD B3 AF DE AD B3 AF
```

Finally, we can also rename a LFS virtual file while preserving the previously set pattern. This is useful for preparing files that will be used by Libswift in zerostate[6] mode. When this is used, Libswift will search files named as `<roothash>`, `<roothash>.mhash`, and `<roothash>.mbinmap`.

```
$ mv deadb3af_example acd177d4
$ hexdump -e '/1 "%02X "' -n 16 acd177d4
DE AD B3 AF DE AD B3 AF DE AD B3 AF DE AD B3 AF
```

3.5 Performance

To asses the performance impact this might have on Libswift, or on the testing machine, we need to run a few simple benchmarks. This section describes and discusses the performance of the read and write operations, as well as memory usage of the LFS process when faced with a large number of files.

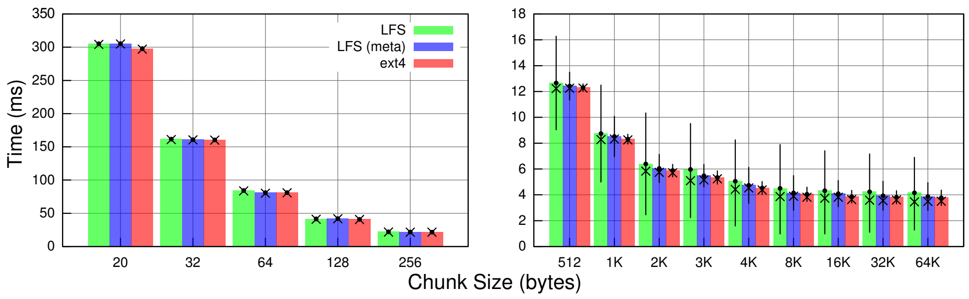


Figure 3.2: LFS `read()` performance.

To test the performance of our virtual filesystem's `read()` operation, we used a trivial C application that reads a 10MB sequentially, in a loop, one chunk at a time. We measure the time needed to completely read the file in memory, but note that the actual resulting value is marginally important – what we seek is to compare 3 types of files. The first file is a normal LFS one, generated on-the-fly with predefined content. The second one is a Libswift metadata file for which the operation goes through LFS, but is actually delegated to the underlying real filesystem. The third one is a file stored on a ext4 filesystem. We variated the chunk size used for each

[6]In this mode, Libswift keeps minimal state information in memory. When metadata is needed, it is read directly from storage. We need this available for testing large files due to the long time required to load metadata in normal mode.

individual call to `read()`, and run each test 1000 times. The plot in Figure 3.2 shows average time (bars), median value (cross), and standard deviation (as error bars).

While not initially expected, we obtained highly comparable performance for all 3 cases even if LFS does not actually read anything from the physical storage device. This could be explained by the fact that we used the same file on ext4 as generated by LFS. This means that the ext4 driver probably cached the 4-bytes repeating content after the first read.

However, we can see large standard deviations when reading LFS virtual files. We ran the experiment multiple times, but the standard deviations are inconsistent. This might be the result of LFS process running in userspace with the same priority as other processes. Since creating the output buffer when `read()` is called is a purely CPU task, the wall time can vary depending on the system load. Implementing a trivial output buffer caching might alleviate this variation.

The most important conclusion is that LFS and the underlying FUSE does not introduce overheads compared with the real filesystem, even when we retrieve predefined content and the caches are probably hit all the time.

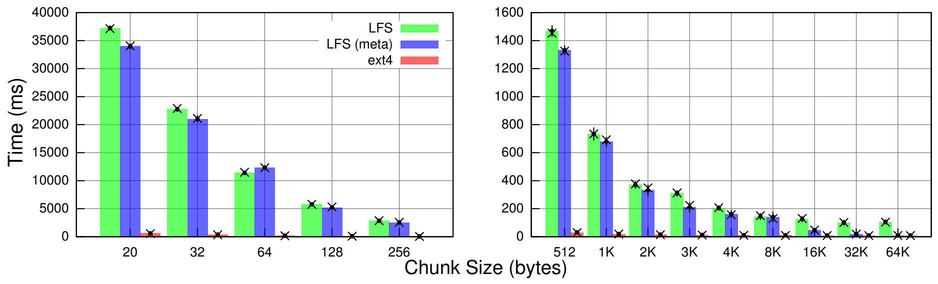


Figure 3.3: LFS `write()` performance.

For the second benchmark, we used the same experimental setup as before, but this time we are issuing 10 `write()` calls for each test on a 10MiB file. Despite the fact that the write operation does not do anything, or just forwards the write to the real filesystem, Figure 3.3 shows huge performance impact. There seems to be overhead from FUSE, but we did not explored this further since it does not impact or tests. For example, with 8K chunks we can write 10MiB in approximately 170ms, which gives us a rate of 60MiB/s, which is above the network speed, and similar to the hashing speed (see Section 5.1).

Finally, since LFS is keeping state information for each file, we also looked at peak memory usage. For 1 million files the LFS process consumes approximately 350MB of memory. Note that using large files does not change anything.

We conclude by noticing the LFS does not have higher impact than a normal filesystem will do, and the approach is completely separated from Libswift. The initial requirements are met, and LFS can be used for these particular testing needs.

Chapter 4

Rewarding Goodness

This chapter builds upon the short overview of Libswift given in Section 1.2.2 with more in-depth details about the protocols inner workings, and finally introduces the changes needed for rewarding goodness. This discussion and contributions only affect the peer protocol part of PPSP. The goal of this part is to develop a mechanism allowing concurrent upload stream, inside Libswift, with different priorities. As described by Figure 4.1, these priorities are based on the reputation values that BarterCast3 computes for each peer. Besides this mechanism, this part also regards the API needed to inject these values into Libswift.

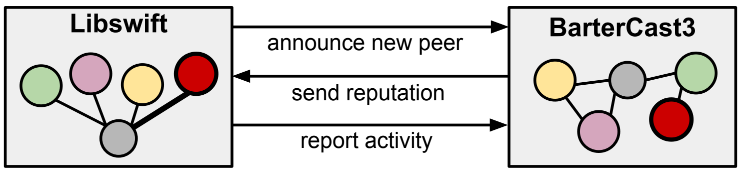


Figure 4.1: Libswift/BarterCast3 feedback loop and the high-level API between these 2 systems.

When a new peer connects to the local Libswift instance his presence will be announced to BarterCast3 which will attempt to compute a reputation value using its own algorithm and knowledge (i.e., local network view). This reputation value is injected into the Libswift process which will use it as input or a *reciprocity* algorithm. The algorithm will rank the newly connected peer alongside with the existing ones, and will prioritize transfer speeds accordingly. This feedback loop closes with the local Libswift instance reporting back to BarterCast3 its activity metrics (e.g., upload, download), thus influencing future reputation computations.

The PPSPP draft RFC[1] offers total freedom to the actual protocol implementation in choosing reciprocity algorithms. Examples of existing algorithms are: BitTorrent's Tit-for-Tat [6] and Give-to-Get [19]. This section will introduce some

[1]This refer to revision 07 of the IETF PPSPP Internet-Draft: `http://www.ietf.org/id/draft-ietf-ppsp-peer-protocol-07.txt`.

concepts and the existing infrastructure before moving on to specific design, implementation and integration details. This should clarify the context of our contributions, and, most importantly, the existing limitations. During the final part of this work we discovered some weaknesses of this approach – we will discuss these in Section 5.2.2.

4.1 Libswift Concepts

The PPSPP design strives to be as simple and generic as possible so it can be adapted to various applications with, possibly, quite different requirements. The existing IETF draft explains the protocol by making a few assumptions such as using UDP as an under-laying transport protocol or Low Extra Delay Background Transport (LEDBAT) [25] for congestion control. Implementations are given freedom in choosing other solutions, but the reference implementation, Libswift, is currently trying to follow the RFC draft as much as possible. As a convenience, we will refer to Libswift or PPSP interchangeably while discussing the concepts needed to understand our approach for the reciprocity algorithm.

Channels. The central entity to one particular Libswift instance is the *channel*. This represents a connection between 2 peers belonging to a specific swarm. This means that transfers belonging to 2 different swarms, but made to the same peer will use 2 channels. At the same time, a swarm can contain multiple channels – one for each connected remote peer. The is one special channel that is always available and used to initiate connections (i.e., similar to a listening socket). The concept of *channels* is homologous to the one of *ports* in other transport protocols such as TCP or UDP. Note that even if Libswift is running on top of UDP, it is using only a single port[2], multiplexing connections using channels. As with ports, channels are also assigned locally unique numbers which are referenced in the packets exchanged during protocol operation.

Messages. PPSP peers exchange datagrams that can contain one or more *messages*. A connection is initiated using a HANDSHAKE message which contains a channel identification number, swarm identification and other protocol options. A HAVE message is used to advertise available content by seeders. A DATA message carries the actual content, but may also contain additional information useful to the congestion control mechanism (i.e., timestamps for LEDBAT). The atomic content portion exchanged in DATA messages is called a *chunk* – its size can vary, but the default one is 1KB. Successful receipt is confirmed using ACK messages which may also contain information for the congestion control mechanism. The content integrity mechanism will use the INTEGRITY and SIGNED_INTEGRITY messages (e.g., containing cryptographic hashes). REQUEST messages can be used

[2]this design simplifies NAT traversal

by pull-based applications, but push paradigm is also supported by PPSP. Peers can withdraw their requests using CANCEL messages. CHOKE and UNCHOKE messages may be used to signal that requests will no longer be answered (e.g., due to congestion), or, respectively, to lift this restriction. PEX messages are used by the peer discovery mechanism. Figure 4.2 shows the messages exchanged when a new connection is initiated.

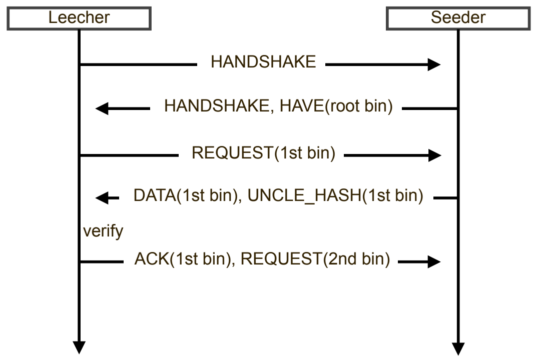

Leecher
Seeder
HANDSHAKE
HANDSHAKE, HAVE(root bin)
REQUEST(1st bin)
DATA(1st bin), UNCLE_HASH(1st bin)
verify
ACK(1st bin), REQUEST(2nd bin)

Figure 4.2: PPSP operation – new connection and first DATA transfer.

To completely explain the picture, PPSP suggests using a compressed content addressing scheme named *binmaps* – they form a tree of aggregated content intervals starting from the bottom chunks (e.g., `HAVE(root bin)` represents the whole content using just a single number rather than a bitmap full of 1s). Individual chunks by the bottom bin numbers (e.g., `DATA(1st bin)`). The Merkle Tree of cryptographic hashes used for content integrity is also obtained as the download progresses – the `UNCLE_HASH(1st bin)` message will contain the hashes needed to verify the received DATA. Since these details are not essential to our discussion, interested readers should refer to the PPSPP draft RFC.

Congestion Control. As previously mentioned, Libswift uses the Low Extra Delay Background Transport (LEDBAT) [27] congestion control mechanism. The goal of LEDBAT is to transport non-interactive traffic, such as PPSP, with *lower than best effort* while fully utilizing available bandwidth. This prevents other normal traffic generated by the machine (i.e., HTTP) from being affected by the P2P client running in the background. The algorithm works by keeping track of one-way delay variations between the a sender and a receiver. The delay is computed by adding timestamps to data packet going out and acknowledgment packets coming in. As opposed to TCP, which can only back-off after the queues on the path are filled up (and other streams are also getting affected), LEDBAT can sense increasing queuing delays by comparing the current delay with a historical base. Notably, it has been showed that the algorithm has some fairness issues [4] with concurrent LEDBAT-controlled streams.

4.2 Design

First, we should consider that Libswift is not queuing the outgoing datagrams in the traditional way – by using a buffer. Since it is build around *libevent*[3], it uses timer to schedule send events in the future.

While the congestion control mechanism regulates the UDP connection of the peer (i.e., all multiplexed channels), our *reciprocity* mechanism acts on top of this by scheduling send events for channels connected to higher-reputation peers more often. Libswift schedules send events for each channel individually when a new REQUEST is received, and reschedules them after each send. The time in future for each (re)scheduling is dictated by the congestion control mechanism, while the *reciprocity* algorithm will get a chance to modify this. This concept is similar to the Weighted Fair Queuing improvement described by Petrocco at al. [22], but we support different weights by using a slightly modified controller. Also, our design can use different algorithms for this which, currently, can be chosen at compile time. Most likely, it is trivial to also implement this change at runtime.

The Libswift instance can be controlled by other applications via a command gateway which is exposed as a TCP socket. The controlling application or module (e.g., BarterCast3 in out case), can open a connection to this address and exchange command messages. To get the list of connected peers, the controller can subscribe to `MOREINFO` messages that will be sent periodically. These messages will contain statistics (e.g., download and upload speeds) along with the peers. BarterCast3 can check for new peers and start computing their reputation. The reputation algorithm can take longer, so we rely on BarterCast3 to inject the priorities back into Libswift, when they are ready. For this, we added a generic `RECIPROCITY` message to the Libswift command gateway. The content of this message is passed to the reciprocity algorithm implementation for further parsing. In the simplest case, the `RECIPROCITY` message will contain a `PEERWEIGHTS` sub-message which contains IP:port and weight pairs for each peer. This design allows the controlling module to easily send other messages to the reciprocity algorithm, such as different tuning parameters. For example, we use an `ACTIVE` command in the experiment described in Section 5.3 to turn the reciprocity mechanism on and off at runtime.

The actual byte-counting and record generation is done when a channel is closed – this usually means that the transfer is done. When starting, BarterCast3 subscribes to close events via the command gateway using a `SUBSCRIBE` message, and each time it receives one it will generate or update a so-called barter record containing transfer statistics of the 2 peers. This is double-signed by the participants and disseminated into the network by Dispersy.

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

4.3 Implementation

Each Libswift channel is handled by a `swift::Channel` object. Send events can be scheduled by using the `Channel::Reschedule()` method, which will obtain a future time from the congestion control mechanism and will set a *libevent* timer. Our reciprocity policy can be asked to modify this scheduling before the *libevent* timer is set.

We added a class member of type `ReciprocityPolicy` (see Listing 4) to the Channel class. Listing 3 shows how this is integrate in the Reschedule() code.

```
void Channel::Reschedule () {
    tint recip_send_interval = 0;
    if (reciprocity_policy_->IsActive())
        recip_send_interval =
            reciprocity_policy_->SendIntervalFor(this);
    // ...
    next_send_time_ = NextSendTime();
    tint duein = next_send_time_ - NOW;
    if (recip_send_interval) {
        duein = max(duein,
            last_send_time_ + recip_send_interval - NOW);
        next_send_time_ = NOW + duein;
    }
    // ...
    evtimer_add(evsend_ptr_, *tint2tv(duein));
    // ...
}
```

Listing 3: Snippet of Channel::Reschedule() using the ReciprocityPolicy object.

As you can see in Listing 3, nothing will happen is the `IsActive()` return `false`. The default implementation will do exactly this. Other algorithms need to inherit the `ReciprocityPolicy` class, or the `BaseReciprocityPolicy` one, if more boilerplate functionality is needed. Before discussing the latter, we will cover a few more details about the methods.

```
class ReciprocityPolicy {
public:
    virtual void AddPeer(const Address& addr);
    virtual void DelPeer(const Address& addr);
    virtual tint SendIntervalFor(Channel *channel);
    virtual void ExternalCmd(char *message);
    virtual bool IsActive();
};
```

Listing 4: ReciprocityPolicy abstract class.

AddPeer(addr). This can be used to keep track of which peers are subjected to the reciprocity policy. Currently, all peers for which a channel is open will be added, but you could imagine a scenario where you need add and remove peers. For example, Libswift could detect attacking peers and ignore them, even if BarterCast3 is sending out reputation values. It is the responsibility of the reciprocity algorithm to take, or not take these peers into account.

DelPeer(addr). Deletes a previously added peer.

SendIntervalFor(channel). This is the main functionality that needs to be implemented by a reciprocity algorithm. It will establish a new `next_send_time_` for the given channel according to some logic.

ExternalCmd(message). As discussed in Section 4.2, the `RECIPROCITY` message received by the Libswift command gateway are forwarded for further parsing to this method.

IsActive(). Returns true is the reciprocity policy is active. It might be disabled at runtime via commands.

On the actual algorithm side, we experimented with a simple P controller, which is used by all the policies presented in the next section. The feedback of this controller is based on the size of incoming requests from a particular peer, relative to the total size of outstanding requests. We empirically set the P coefficient to 0.8, as shown in Listing 5. While this provides acceptable performance, in-depth analysis of better control loops is required.

```
// ...
double feedback_ratio =
    (double)channel->hint_in_size() / tot_req;
double desired_ratio = GetPeerWeight(channel->peer());
// ...
double error = (desired_ratio - feedback_ratio);
double ratio = feedback_ratio + error * 0.8;
```

Listing 5: P controller for dividing bandwidth.

4.4 Usage

Basic policies are currently implemented as Libswift extensions – you can find them in the `ext/recip` directory. The `BaseReciprocityPolicy` (BASE) implements basic external command handling. It supports the `PEERWEIGHTS` and `ACTIVE` messages described in Section 4.2. It also keeps track of peers as they are added, or deleted, but feature is not used at the time of this writing. The `EqualReciprocityPolicy` considers all peers equal – this is essentially similar to the Weighted Fair Queuing described by Petrocco at al. [22].

The SelfishReciprocityPolicy divides bandwidth according to the relative reputations of the peers, but tries to maximize the upload capacity of the seeder in case some of the leechers cannot handle the speed that was assigned to them. Finally, the WinnerReciprocityPolicy, assigns full priority to a single peer, the one with the best reputation.

Libswift can be compiled with reciprocity support by setting the RECIP preprocessor variable, or passing -DRECIP as a compiler flag. Additionally, a specific reciprocity algorithm can be loaded using the RECIP_TYPE variable. Currently, you can use ”1” for the BASE policy, ”2” for SELFISH, ”3” for WINNER, and ”4” for EQUAL.

Chapter 5

Experiments

This chapter describes and analyzes results from three incremental experiments that were built as part of this work. They seek to empirically demonstrate our approaches and provide a basis for further in-depth evaluation. Section 5.1 starts with stress-testing Libswift using the newly built support for very large files. Next, in Section 5.2, we focus our attention on the newly implemented reciprocity policy mechanism in Libswift. Finally, in Section 5.3, we also introduce BarterCast3 into the picture, by building a full integration experiment that runs multiple Tribler instances on the DAS-4 supercomputer.

5.1 Terabyte Seeding

The goal of this 1TiB seeding experiment is to evaluate the ability of Libswift in supporting power users that might generously donate their resources. The experiment is deployed on the continuous integration platform used by the Tribler team[1]. The objective is to run it when new versions of Libswift are released in order to detect eventual performance regressions.

We are deploying a simple scenario with a 2-peers swarm: one seeder, followed by one leecher. They both remain connected until the end of the transfer. Both peer instances run on the same machine, but on different CPU cores. Note that the current Libswift implementation is single-threaded, single-core. We are using the virtual filesystem described in Chapter 3 to generate a single 1TiB file. Because computing the Merkle hash tree takes a very long time, we precomputed the `.hash` and `.mbinmap` files, and they are used by all runs. For the same reason, the swarm is set to use the maximum recommended chunk size of 8KiB. Compared to the default chunk size of 1KiB, this requires 8 times less hashes. It still takes around 5 hours to generate the hashes for a 1TiB file on a 2.10GHz core – this is equivalent to a hashing speed of around 58MiB/s. Even if this is a one-time operation, for a particular piece of content, we believe the hashes can be computed more efficiently

[1] `http://jenkins.tribler.org/job/Experiment_Libswift_Terabyte_Seeding/`

in parallel, or by taking advantage of the GPU. This might be required by power users interested in using Libswift to share very large amounts of new content (e.g., media producers).

The machines used to run this experiment are powered by Intel Xeon processors clocked at 2.40GHz, with 8GiB of memory. Since one experiment run takes a very long time, since it transfers 1TiB at less than 20MiB/s, we are only presenting a fraction of its evolution in the following figures.

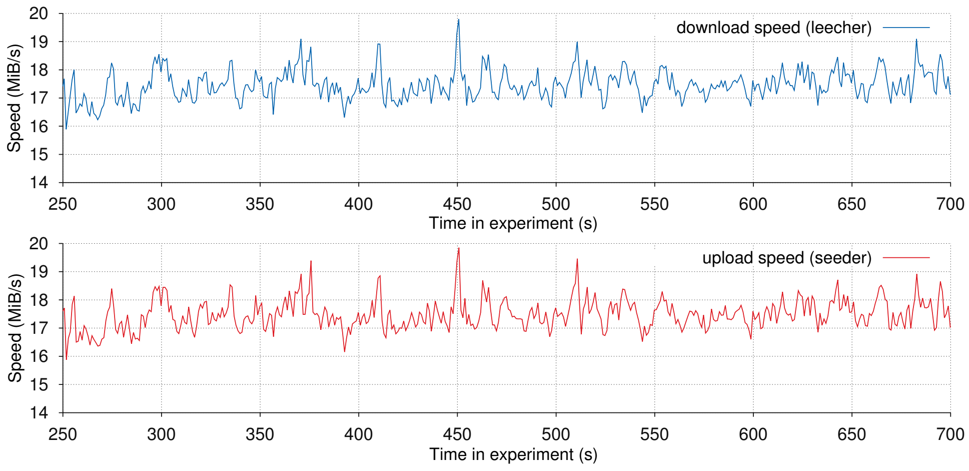


Figure 5.1: Download/upload speed.

Figure 5.1 shows the download speed of the leecher (top), and the upload speed of the seeder (bottom). The values are sampled every second. While this only shows a fraction of the total run time, the results are similar throughout the experiment. Considering the isolated environment in which we are running the experiment, we did not expect the throughput instability shown in the plot. We could speculate that this is normal LEDBAT behavior yielding promptly to other TCP traffic on the testing machines, but investigation this is out of the scope of this work. We obtained more stable results on a local machine.

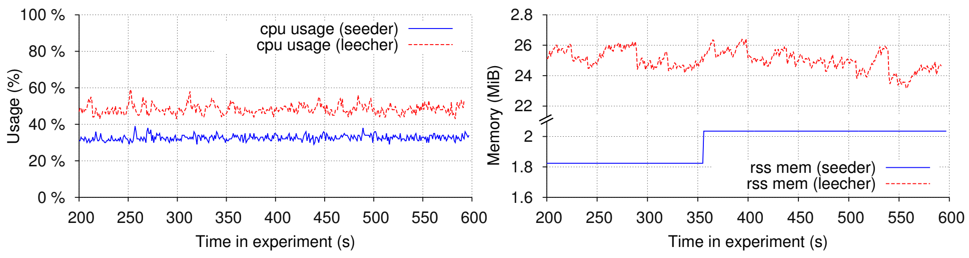


Figure 5.2: Memory usage for seeder and leecher (Resident Set Size). The seeder is running in zerostate, while the leecher is running in normal mode.

As shown in Figure 5.2, the experiment is also tracking memory usage for each peer. We focus on comparing a peer running in normal mode (the leecher) and one running in zerostate mode (the seeder). As depicted, the former has higher usage than the latter. The zerostate mode was designed to minimize the amount of state information that is being kept in memory for each swarm. The majority of this metadata is formed by the integrity hashes, which, in zerostate mode, as opposed to normal mode, are always read from the disk instead of being mapped in memory. The RSS[2] usage of the leeching peer is almost constant because the OS is swapping out the memory containing hashes that are no longer used.

Note that peers in our scenario are joining a single swarm. A peer in normal mode would be using a lot more memory. On the other hand, running in zerostate will probably incur high I/O activity on the storage device. Unfortunately, we were not successful in reliably measuring I/O usage for our FUSE filesystem.

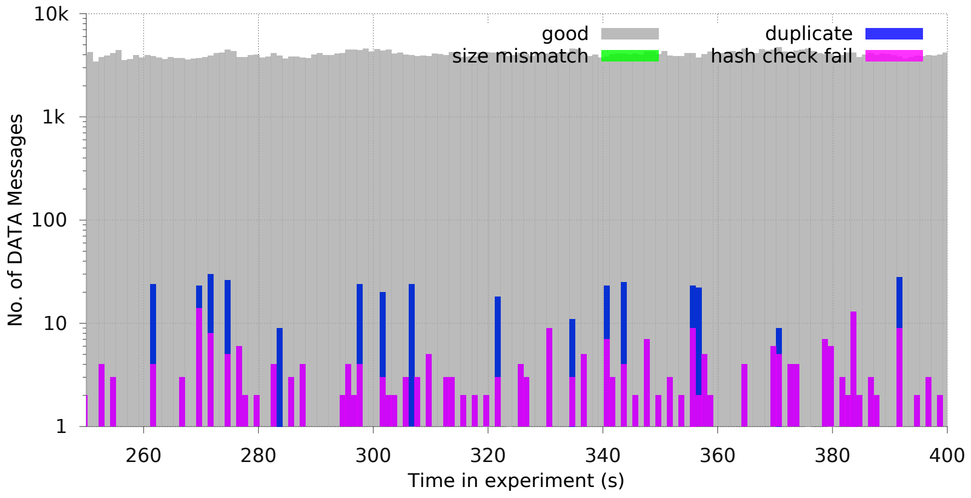


Figure 5.3: Number of received DATA messages at the leecher, failed chunks included.

Figure 5.3 shows the number of DATA messages received by the leecher. Each bar represents the sum for one second and is divided into the number of good messages, and the number of messages that failed 3 kinds of correctness checks. The first one, *size mismatch* checks if the size of the actual content matches the chunk size of the swarm – this should only happen in rare cases such as packet corruption or network failure. We did not encounter this error during our tests. *Duplicate* messages can appear if the original message was successfully received, but the acknowledgment was not yet sent. *Hash check failures* occur when the hash of the received data chunk does not successfully verify against the rest of the tree. This can happen as a result of the hash tree being corrupted (e.g., because other required hashes where not yet received), or if the actual data was modified. Details about the content integrity checking mechanism can be found in the PPSPP specification.

[2]The Resident Set Size represents a process's memory than is actually in RAM, and not swapped out or waiting on-demand loading.

Since our virtual filesystem returns predefined content, we trivially confirmed that the problem is not with the actual content, but with missing or incorrect hashes. Since the algorithm is expected to take care of sending and processing all required hashes before the data chunks that need them are checked, further investigation into this problem is required, but it is, again, out of scope for this experiment.

To conclude, we believe this continuous integration stress experiment provides valuable insights into the operation of Libswift. Preliminary results showed good stability and decent CPU usage, considering the unbounded transfer speed and the fact that all peer instances run on the local machine. From another point of view, the experiment did reveal some possible subtle problems that need further attention. In scenarios with a large number of swarms, better memory management might be needed if the zerostate mode proves demanding on the underlying storage. Unfortunately, we were not able to draw a final conclusion about I/O load in the zerostate case while using the FUSE filesystem. Note that this test would be more interesting in case of a very large number of files, as opposed to big files. This case is also not covered by this experiment, but is supported by the virtual filesystem. Finally, parallelization of hash computation might be a useful improvement for power users that need to distribute large amounts of data.

5.2 Libswift Reciprocity

The aim of this experiment is to demonstrate the possibilities of the new Libswift reciprocity mechanism detailed in Chapter 4. We start by analyzing its behavior with different reciprocity policies, and continue with a deeper look at possible concerns regarding the impact on *libevent*'s timer scheduling.

5.2.1 Policies

To demonstrate how different policies can be implemented, and observe their behavior, we use a basic setup with a single seeder and two competing leechers. The peer discovery mechanism is disabled, so leechers do not communicate with each other. To simplify things, we are also bypassing the command interface by using hard-coded priorities for each experiment.

Each leecher is downloading a 50MiB file from the seeder. The download speed of both leechers is capped at 1MiB/s, and they compete for the seeder's 1MiB/s upload bandwidth. Both caps are set using Libswift's mechanism (`--uprate` and `--downrate` flags). As the plots show, there is a slight inaccuracy incurred by this mechanism (i.e., the sum of download speeds exhibited by the leecher is higher than the seeder's cap). In order to depict the real-time speed adjustment, the second leecher joins the swarm 20 seconds later than the first one.

Figure 5.4 shows, for each policy we implemented, upload speed of the seeder, download speeds of the 2 leechers, and progress percentage for the leechers. We are mainly interested in the evolution of the 2 download speeds. These are sampled

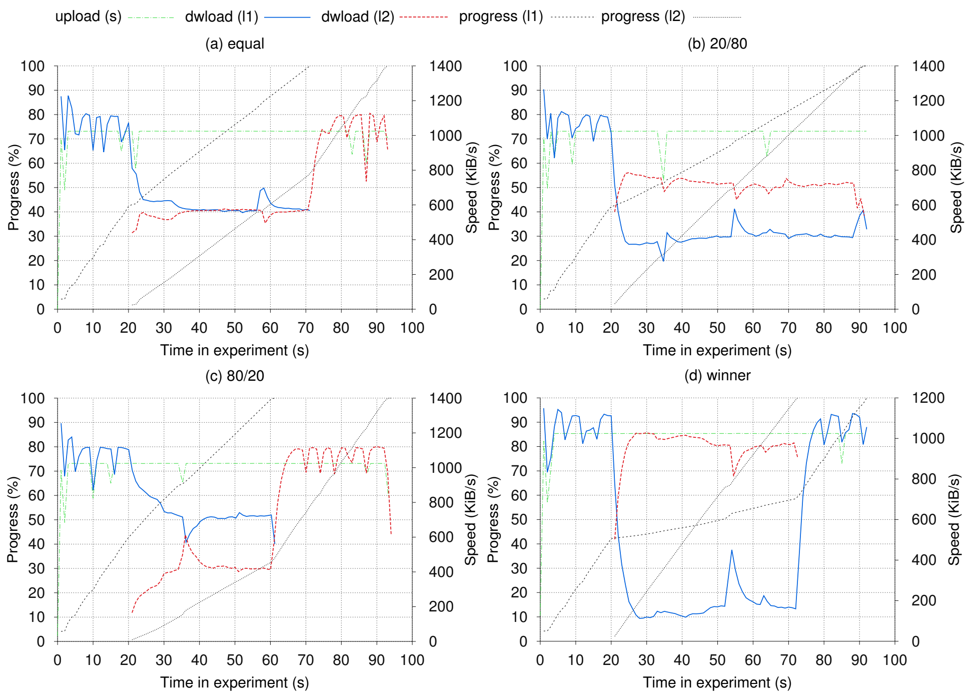


Figure 5.4: Four Libswift priority policies. One seeder, two leechers scenario.

every second from a moving average with a 2-seconds interval in order to reduce noisy variation in the measurement.

Besides adding the reciprocity mechanism, we also changed the requests scheduling interval of the leechers to 2 seconds from the original 1 second. Briefly, the speed and congestion control mechanism is based on how many requests are being send by a leecher, and how many are answered in timely fashion by a seeder. Each time the leecher needs to send new requests (which happens a lot more often than 2 seconds), it computes a targeted total number of requests for the next 2 seconds (in our case) such that it fills up the available bandwidth. The target value is also constrained by other factors such as LEDBAT congestion control mechanism. Note that, as presented in Chapter 4, the reciprocity mechanism is also based on the assumption that the leecher will lower the number of requests if it observes delays in receiving the requested data. Without the mentioned change, we obtained highly unstable results (i.e., peers stopping completely, high variance in speed).

The top left plot of Figure 5.4 shows 2 leechers competing with equal priority. We observe that the controller throttles the speed of the first leecher almost immediately, while both reach the same speed after 15 seconds. The top right and bottom left plots show upload bandwidth being split between 2 peers having very different priorities – 80% and 20%. The first run gives high priority to the second leecher, while the second run gives lower priority to the second peer. Both runs exhibit a higher response time until an accurate bandwidth division is reached – approximately 25-30 seconds. The last plot, bottom right, show a special *winner-takes-all* case in which the second peer has full priority over the first one. Note that even if this is the case, we are still assigning a minimum residual bandwidth for the first peer. If we attempt to go lower than this, the connection will be dropped. Note this this could also be an acceptable policy.

The occasional spikes that can be observed are caused by LEDBAT trying to optimistically increase speed, but being immediately throttled down by the reciprocity mechanism. In the beginning if the 80/20 case the speed overshoots, but is settled back very rapidly, in just one step. This behavior might be alleviated by using a more complex controller (e.g., PID or a cascaded P).

5.2.2 Source of Inaccuracy

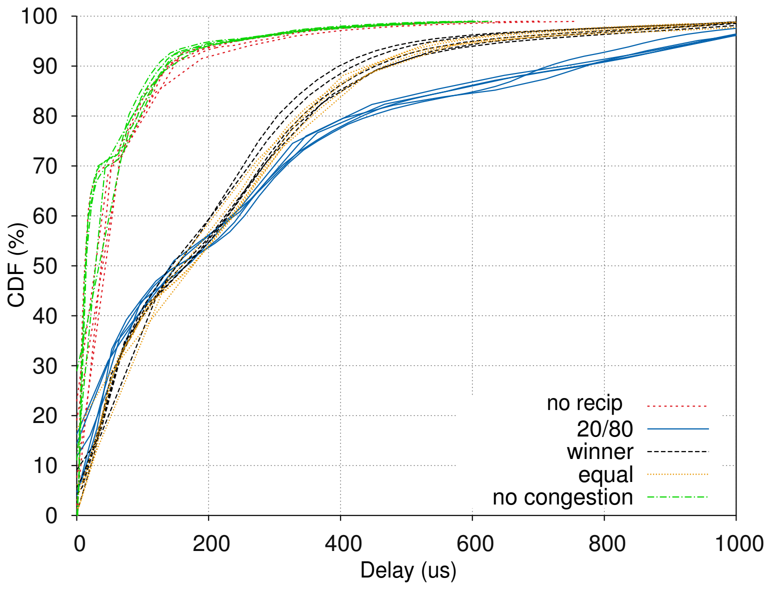


Figure 5.5: Seeder send event delay CDF.

This experiment investigates the inaccuracy and throughput instability that we previously observed. Since the reciprocity mechanism is postponing the triggering deadline of send events as set by the congestion control mechanism, we speculate that this might negatively influence the event scheduling done by *libevent*. This could delay the handling of send events, which might be the source of inaccuracies and instability in speed depicted by previous plots.

To argue this point of view, we instrumented each triggered send event by logging its targeted due time alongside its actual run time. The targeted triggering time is the one set by LEDBAT, and, if active, increased by the reciprocity mechanism. We ran 2 baseline tests in order to exclude the effects of congestion: *no recip* and *no congestion*. The reciprocity mechanism is completely deactivated in both of them, but, for the second one, we also increased the bandwidth cap of the seeder such that there is no longer a competition between the leechers. Note that all other tests use the same scenario as described in Section 5.2.1. The remaining 3 tests, *20/80*, *winner*, and *equal*, have the reciprocity mechanism on, and the priorities are set in the same way as we previously discussed. To also exclude possible transient conditions of the test environment (e.g., CPU load), each test is run 5 times.

Figure 5.5 shows the cumulative distribution function of the send event scheduling delay for each test. The delay is computed as the difference between the targeted time and the actual run time of the event. The plot clearly supports our initial concern by showing increased delays when reciprocity in active as compared to the baseline tests. We can observe how 90% of events in the baseline tests are below approximately 170us, while the limit increases to 400us, and 730us when reciprocity is active. Delays above the 99% threshold are not plotted, but note that a very small number of these remaining 1% reach values of more than 100ms. While the 99% of delayed events that are plotted seem to already support our suspicions, we should take a deeper look at the not-plotted extreme outliers.

In conclusion, we believe this might be the cause of occasional instability, and it might be useful to re-implement the whole congestion control mechanism such that it also handles reciprocity. Using a different queuing solutions (e.g., with traditional buffers), instead of *libevent* timers, might also be a good alternative, or at least should be considered for a more in-depth future evaluation.

5.3 Full Tribler

The aim of this final experiment is to provide a proof-of-concept demonstration of the complete mechanism used by Tribler for *rewarding good behavior*. It shows the process of building up a peer's reputation in the network, and then making this available to other content providers. Finally, a seeder will divide his upload bandwidth between 2 peers he never seen before, thus demonstrating the indirect reciprocity concept.

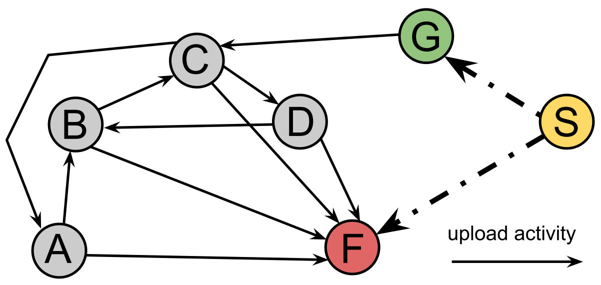


Figure 5.6: Indirect trust scenario.

For this experiment we adapted the latest Tribler development version (at the time of this writing) so it can be run on the DAS-4 supercomputer. Every module that is not essential to our experiment was disabled: GUI, Dispersy communities (except BarterCast3) caches, torrent collecting, the BitTorrent engine, and the DHT. A few minor changes were needed to push the reputation values into the Libswift process via the command gateway.

Each peer generates a 200MiB file of predefined content (i.e., repeating bytes containing his peer id) that will be used in the experiment. We precomputed the hashes for all possible 256 files. We need these because it is not possible to find the endpoint (IP and port) of a seeder without a DHT, which is not present in our setup.

The experiment is implement using the Gumby framework. This helps setting up dependencies on the DAS-4 head node, and submitting the jobs. Once the experiment has started, Gumby will parse and execute a scenario file (see Listing 6 for the one describing this experiment). Porting the scenario file parser, previously developed by the Tribler team, from an older testing framework to the new Gumby was part of this work.

The scenario file specifies a function, together with its arguments (e.g., `leech <source peer> <roothash>`), that will be run a time relative from the start of the experiment (e.g., `+0:6`). Finally, each command can apply to one or more peers (e.g., {`A,B`}).

```
+0:0 generate_file {A,B,C,D,G,S}
+0:6 seed 04bce4b980f66b041084d15d1198aa54f719534b {A}
+0:6 seed 6280cd598cdd2c4d60b022f30cb9dc0a664aef82 {B}
+0:6 seed b5256892f8badd85da83833f63e99c22a2b1f2e8 {C}
+0:6 seed 69b5756df844841ccbeca3017e318a3da7fbc79f {D}
+0:6 seed e46bd426d86ed68a4e0871a4b80e29400e70351c {G}
+0:6 seed ffc608bd527d0acfcceb42972b52651c78b93ff0 {S}
+0:10 leech B 6280cd598cdd2c4d60b022f30cb9dc0a664aef82 {A}
+0:10 leech C b5256892f8badd85da83833f63e99c22a2b1f2e8 {B}
+0:10 leech D 69b5756df844841ccbeca3017e318a3da7fbc79f {C}
+0:10 leech B 6280cd598cdd2c4d60b022f30cb9dc0a664aef82 {D}
+0:10 leech A 04bce4b980f66b041084d15d1198aa54f719534b {C}
+0:15 leech G e46bd426d86ed68a4e0871a4b80e29400e70351c {C}
+0:20 leech A 04bce4b980f66b041084d15d1198aa54f719534b {F}
+0:22 leech B 6280cd598cdd2c4d60b022f30cb9dc0a664aef82 {F}
+0:24 leech C b5256892f8badd85da83833f63e99c22a2b1f2e8 {F}
+0:26 leech D e46bd426d86ed68a4e0871a4b80e29400e70351c {F}
+0:30 activate_reciprocity {S}
+0:120 leech S ffc608bd527d0acfcceb42972b52651c78b93ff0 {F}
+0:130 leech S ffc608bd527d0acfcceb42972b52651c78b93ff0 {G}
```

Listing 6: Full Tribler scenario specification.

The first stage of our scenario creates a core of good behaving peers (A, B, C, D). They all download files from each other, building up reputation. At the end of this first stage a free-riding peer (F) initiates downloads from all core seeders. This will result in lowering his reputation since he is not uploading anything back to the community. At the same time, a good behaving peer (G) will make one upload to the core. To simplify operation, the reciprocity algorithm in Libswift is disabled during this initial phase.

In the second stage, peer S will receive requests from 2 peers he *never met before*, a free-rider (F) and a good peer (G). Peer S has its reciprocity mechanism turned on and will apply the winner-takes-all policy. If reputation values are correctly disseminated by BarterCast3 and injected into the Libswift process running at peer S, the free-riding peer (F) should be choked, while the good behaving peer G will get all the bandwidth. Note that the upload speed of S is capped at 5 MiB/s.

The high-level view of the transfers that take place in the scenario is depicted in Figure 5.6. The actual scenario specification is presented in Listing 6.

The experiment is deployed using the Jenkins continuous integration platform[3]. Initial results show delays in propagating the reputation records in some runs, but this is not necessarily an issue since we cannot expect real-time delivery. Once the reciprocity values reach the Libswift instance of peer S similar behavior to the one shown by Figure 5.4 (d) can be observed.

To conclude, we succeeded in building a full integration experiment that proves the indirect reciprocity concept in Tribler. Unfortunately, in-depth analysis of the system's behavior still remains an open subject. The ability to run the experiment on a clustered environment enables more complex scenarios. For instance, we can investigate scalability by replacing the core network (peers A, B, C, and D) with 1000 instances, but further engineering efforts are needed to reach this objective.

[3]`http://jenkins.tribler.org/job/Experiment_TriblerNoGui_Reciprocity`

Chapter 6

Conclusion

This thesis contributed to the Tribler team's quest towards *rewarding good behavior* in a fully decentralized P2P network. This was a first attempt to integrate the reputation gossiping overlay with the download engine and study the possibilities for free-riding prevention. We first analyzed the problem of generous donations by evaluating the capacity in which Libswift can support large seeding. Secondly, we incrementally moved towards a proof-of-concept integrated experiment that can demonstrate Tribler's approach for solving the free-riding problem through gossiped reputation and indirect reciprocity.

The following specific contributions were made as part of this project:

- We proposed and analyzed the performance of a filesystem-based solution for testing the reference PPSP implementation, Libswift, with very large files or a large number of them. Moreover, using this tool, we build an initial experiment aimed at evaluating the capability of Libswift to seed very large amounts of data.
- We implemented a reciprocity mechanism for Libswift with the objective of prioritizing the limited upload bandwidth of a seeder based on the reputation of its barter partners. We demonstrated different policies that can be implemented using this reciprocity mechanism, and discussed its limitations.
- We built a full integration experiment using the Gumby testing framework and aimed at demonstrating the link between the reciprocity mechanism and BarterCast3, the reputation dissemination solution. In our basic staged scenario, the test peer recognized the free-rider between 2 peers he never met before using only indirect information. Since this experiment was one of the first users of Gumby, we also made a few smaller contributions to the new testing framework.

The major issue of the Libswift reciprocity mechanism (and maybe the congestion control), which needs to be addressed in future work, is instability and accuracy of the bandwidth division. During our 1 seeder/2 leechers experimental scenario (see Section 5.2), we occasionally experienced complete connection drops for one of the leechers. The accuracy of the bandwidth allocation might be influenced by the send event scheduling issue analyzed in Section 5.2.2.

Though we only aimed at a initial demonstration of the integrated solution, the *Full Tribler* experiment detailed in Section 5.3 should be extended to a larger scale. We initially envisioned a setup with 1000 Tribler instances, but turned out to be too ambitious at this point. We need further engineering effort to allow this kind of setup on the DAS-4 supercomputer.

6.1 Future Work

We started this work with very ambitious goals, but the extensive number of technologies involved, the fact that most of them are in early and under active development, and the lack of documentation hindered our efforts to provide a production-level solution. However, we did analyzed and demonstrated a few key concepts, and we gathered the following list of directions for future research. Besides some obvious next steps, these pointers also cover a few rougher ideas.

Reciprocity and Congestion Control. Considering the event scheduling issues we encountered, it might be useful to develop an algorithm that handles both congestion control and reciprocity policy. Additionally, analyzing a traditional queuing method (i.e., using buffers), instead of *libevent*'s timers, might also be a next research direction.

Indirect Reciprocity in BitTorrent. Since Tribler is still mainly using BitTorrent, efforts to port the indirect reciprocity mechanism and link BarterCast3 with the BitTorrent engine might be interesting.

Advanced Experiments. First, to analyze scalability and overhead, the Full Tribler experiment needs to be upgraded to a larger scale (i.e., 1000 instances). Another experiment we had in mind was called *"converted opportunistic free-rider"*. Its aim was to demonstrate the real-time response capabilities of the reputation system to changing behavior of a peer from free-riding to contributing.

Testing Infrastructure. The first step would be to integrate the virtual filesystem into Gumby, the new testing framework. Additionally, it will probably be trivial to move away from the inefficient write operation in LFS and just use `/dev/null`. A more advanced solution would be to implement mocked version of the objects that wrap the read and write calls in Libswift. This requires modification to the binary (i.e., linking it against the mocked classes), but it also means more advanced control. This final approach could be use to simulate different demanding scenarios (e.g., corruption, low speed).

Security. There a number of security concerns about Libswift that are out-of-focus at this early development stage. A question related to our reciprocity mechanism is: What happens if malicious peers do not lower the number of requests they send as a result of bandwidth throttling? The assumption that this will happen is most likely false in real-world, thus a more secure bandwidth prioritization algorithm must be analyzed.

Overlapping Overlays. We are trying to merge 2 P2P overlays that use different algorithms for peer discovery. On one side, the BarterCast3 employs its own algorithm for peer discovery and also guides dissemination based on reputation. On the other side, the Libswift instance becomes known to potential leechers via its internal mechanism. The set of Libswift peers that are competing as some moment in time could be totally different than the peers whose reputation values where received by the BarterCast3 instance. Further insight into the recall and accuracy of this merged system is needed.

Guided Leeching. Until now, we analyzed the reciprocity problem from the perspective of a seeder, but we could also look at this from the perspective of a leecher. For example, leeching from low-reputation peers gives them a chance to improve, while leeching from high-reputation peers separates them even more from the low-reputation ones.

Roaming Identity. BarterCast3 currently identifies peers by their Libswift endpoint (i.e., IP and port number). An improvement could be assigning different endpoints to the same identity such that reputation stays the same across different devices (e.g., your laptop and your mobile phone). Another related feature would allow reputation transfers between identities.

Bibliography

[1] Napster. `http://web.archive.org/web/20000815072716/http://www.napster.com/`. Website (accessed: 2000-08-15).

[2] BitTorrent. `http://www.bittorrent.com/`. Website.

[3] BitTorrent, Inc. Bittorrent and torrent software surpass 150 million user milestone. Online Article, January 2012. `http://www.bittorrent.com/company/about/ces_2012_150m_users/`.

[4] Giovanna Carofiglio, Luca Muscariello, Dario Rossi, and Silvio Valenti. The quest for ledbat fairness. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6. IEEE, 2010.

[5] Cisco. Visual networking index: Forecast and methodology, 2012-2017. `http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html`, May 2013. Online Article.

[6] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[7] Michal Feldman, Kevin Lai, Ion Stoica, and John Chuang. Robust incentive techniques for peer-to-peer networks. In *Proceedings of the 5th ACM conference on Electronic commerce*, pages 102–111. ACM, 2004.

[8] Dimitra Gkorou, Tamás Vinkó, Johan Pouwelse, and Dick Epema. Leveraging node properties in random walks for robust reputations in decentralized networks. In *Peer-to-Peer Computing (P2P), 2013 IEEE 13th International Conference on*. IEEE, 2013.

[9] Garrett Hardin. The tragedy of the commons. *New York*, 1968.

[10] Yan Huang, Tom ZJ Fu, Dah-Ming Chiu, John Lui, and Cheng Huang. Challenges, design and analysis of a large-scale P2P-VoD system. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 375–388. ACM, 2008.

[11] ipoque. Internet study 2007/2008. `http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf`, 2009. Online Article.

[12] A.L. Jia, B. Schoon, J.A. Pouwelse, and D.H.J. Epema. Estimating user interaction strength in online networks. Technical Report PDS-2013-007, Delft University of Technology, 2013.

[13] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the kazaa network. In *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, pages 112–120. IEEE, 2003.

[14] Bo Li, Susu Xie, Yang Qu, Gabriel Yik Keung, Chuang Lin, Jiangchuan Liu, and Xinyan Zhang. Inside the new Coolstreaming: Principles, measurements and performance implications. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1031–1039. IEEE, 2008.

[15] Jian Liang, Rakesh Kumar, and Keith W Ross. The fasttrack overlay: A measurement study. *Computer Networks*, 50(6):842–858, 2006.

[16] libtorrent. Seeding a million torrents. `http://blog.libtorrent.org/2012/01/seeding-a-million-torrents/`, note = Online Article, January 2012.

[17] Michel Meulpolder, LE Meester, and Dick HJ Epema. The problem of upload competition in peer-to-peer systems with incentive mechanisms. *Concurrency and Computation: Practice and Experience*, 2012.

[18] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. BarterCast: Fully distributed sharing-ratio enforcement in BitTorrent. Technical Report PDS-2008-002, Delft University of Technology, 2008.

[19] Jacob Jan-David Mol, Johan A Pouwelse, Michel Meulpolder, Dick HJ Epema, and Henk J Sips. Give-to-get: free-riding resilient video-on-demand in p2p systems. In *Electronic Imaging 2008*, pages 681804–681804. International Society for Optics and Photonics, 2008.

[20] Giovanni Neglia, Giuseppe Reina, Honggang Zhang, Don Towsley, Arun Venkataramani, and John Danaher. Availability in bittorrent systems. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2216–2224. IEEE, 2007.

[21] Flutra Osmani, Victor Grishchenko, Raul Jimenez, and Björn Knutsson. Swift: the missing link between peer-to-peer and information-centric networks. In *Proceedings of the First Workshop on P2P and Dependability*, page 4. ACM, 2012.

[22] Riccardo Petrocco, Johan Pouwelse, and Dick HJ Epema. Performance analysis of the libswift p2p streaming protocol. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 103–114. IEEE, 2012.

[23] 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 PDS-2008-005, Delft University of Technology, 2008.

[24] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[25] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. Ledbat: the new bittorrent congestion control protocol. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6. IEEE, 2010.

[26] Thomas Schaap. Performance assessment of libswift. MSc thesis, Delft University of Technology, August 2012.

[27] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental), Dec. 2012.

[28] M. Stuart. P2P-Next. `http://www.dcia.info/activities/p2pmslv2009/1-7%20P2PMS%20P2P-Next.pdf`. Online Article.

[29] Niels Zeilemaker, Mihai Capotă, Arno Bakker, and Johan Pouwelse. Tribler: P2P media search and sharing. In *Proceedings of the 19th ACM International Conference on Multimedia*, pages 739–742. ACM, 2011.

[30] Niels Zeilemaker, Boudewijn Schoon, and Johan Pouwelse. Dispersy bundle synchronization. Technical Report PDS-2013-002, Delft University of Technology, January 2013.

[31] Y. Zhang and N. Zong. Problem Statement and Requirements of the Peer-to-Peer Streaming Protocol (PPSP). RFC 6972 (Informational), July 2013.