Performance Assessment of lib*swift*

Thomas Schaap

Delft University of Technology

Performance Assessment of lib*swift*

Master's Thesis in Computer Science

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

Thomas Schaap

14th August 2012

Author
Thomas Schaap

Title
Performance Assessment of lib*swift*

MSc presentation
August 30, 2012

Graduation Committee

prof. dr. ir. D.H.J. Epema (chair)	Delft University of Technology
dr. ir. J.A. Pouwelse	Delft University of Technology
dr. S.O. Dulman	Delft University of Technology

Abstract

A performance comparison has been done between lib*swift* and other P2P clients to assess whether lib*swift* can be made the fastest P2P client currently available. A modular testing framework targeted at testing and measuring P2P clients has been developed and has been succesfully used to run several experiments with the clients and to debug and improve lib*swift*.

The results mainly compare lib*swift* and libtorrent; μTorrent has been found unreliable under Linux and HTTPS was only used as a baseline measurement. lib*swift* has also been compared to itself with different block sizes. Compared to libtorrent lib*swift* performs quite well, but still suffers from two deficiencies: degrading download performance when many peers try and download the same swarm and large memory usage when confronted with very large files. lib*swift* usually uses far fewer resources than libtorrent, though, while giving similar performance. Especially for use on mobile and other constrained devices or for joining large amounts of swarms lib*swift* seems to be a good choice already.

During the assessment several problems in lib*swift* were identified and resolved. In particular a hard limit on the number of files lib*swift* could handle was removed.

Making lib*swift* the fastest P2P client can certainly be done in the near future: only two deficiencies remain and lib*swift* already shows several strong points.

Preface

My Master of Science thesis, which you are holding, has not come about naturally. I have been passionate about several topics for quite some time and wanted to combine them: networking, security and programming. The Tribler project seemed like an excellent choice for this: I wanted to introduce server-less authentication into Tribler, allowing for controlled communities with only authenticated members. It was a perfect combination of networking and security and would surely require good programming. Only one thing was lacking: the current status of Tribler simply had no need or place for such features. Exploring the project had caught my attention, though, and work was needed on the new lib*swift* protocol: a grand idea of simple file hosting backed by lib*swift* was to be explored — and combined all my interests. Only one thing was lacking: the current status of lib*swift* might not be able to cope with such ambitions. An assessment of the performance was needed before such ambitions could proceed. That assessment became my Master of Science thesis and I have happily observed it included most of my interests. Only one thing is now lacking: the next master student to further lib*swift*'s ambitions.

I would like to extend my thanks to several people that have helped me a lot during the project. Firstly, I would like to thank my supervisor dr. ir. Johan Pouwelse for his support, enthusiasm and many ideas during this project. My thanks also go to ir. Riccardo Petrocco for testing the testing framework and our discussion about that framework and lib*swift*, and to dr. Arno Bakker for answering all my questions about lib*swift* and his comments on this thesis. I would also like to thank Margot Wehrmeijer for her support during the project and for proofreading this thesis. Finally, I would like to thank prof. dr. ir. Epema and dr. Dulman for their participation in my graduation committee.

Thomas Schaap

Delft, The Netherlands
14th August 2012

Contents

Chapter 1

Introduction

Over the years the internet has seen more and more traffic and a shift from simple textual information to more multimedia use. YouTube currently already serves over 3 billion hours of video every month and sees its archive growing with 72 hours of video every minute [38]. In a study towards trends on IP traffic Cisco foresees significant continuing growth of overal traffic, a majority of which is and will remain video. Peer-to-peer (P2P) traffic is foreseen to fall behind in favour of video streaming, but will still grow from 4.6 TB per month in 2011 to 10 TB per month in 2016 [8] [7].

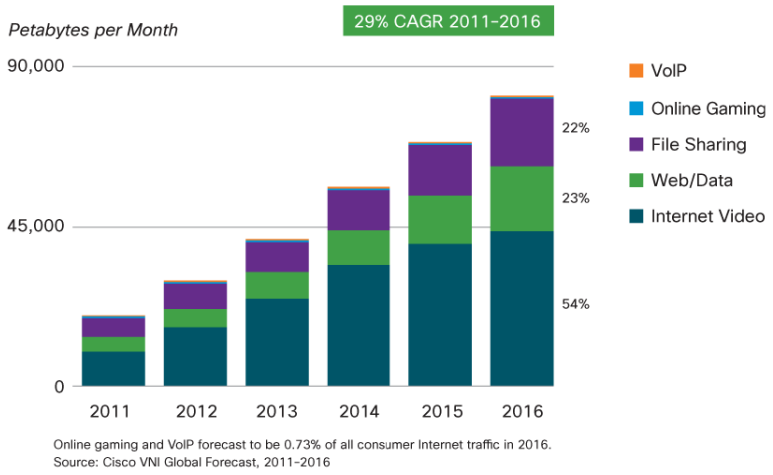


Figure 1.1: Current (2011) and projected traffic according to Cisco's VNI Global Forecast, 2011-2016 [8]

P2P technologies are used more and more to deliver video and other content. During the 2006 FIFA World Cup, for example, measurements were made on several P2P TV clients by Silverston and Fourmaux [27]. A rough combination of

their numbers [1] indicates that at least 1000 peers were watching the same soccer game on SOPCast during their measurements. Major companies also use P2P technology to speed up distribution of content. Game companies like Blizzard, for example, use BitTorrent clients to distribute patches [5].

A drawback of current P2P applications is that they require standalone clients. To make use of a P2P protocol a user needs to install extra software, such as a plugin or often a complete application. All those P2P protocols also build on top of the existing protocols, which have not been designed for multiparty communication, for their data transfers.

1.1 lib*swift*

lib*swift* [28] is a new P2P protocol that tries to bridge the gap between the current infrastructure and the P2P paradigm. Unlike UDP and TCP, which are often used as the basis to run P2P protocols on top of, lib*swift* is designed for multiparty communication. It can also run both on top of the existing infrastructure, using UDP or TCP packets, or replace part of that infrastructure entirely to make the P2P paradigm a core component of tomorrow's internet. This also means it does not have the disadvantage of protocols like multicast, which was developed for multiparty communication but could not run on top of the existing infrastructure and has hence never been widely adopted.

Currently, lib*swift* aims to become an internet standard through the IETF and is also being poised as the P2P technology to use for delivering streaming video [1]. Being an internet standard will hopefuly mean that operating systems will provide support for the protocol, which means no extra software is needed to use lib*swift*. In this vision any software on a computer can transparently utilize the strength of a P2P network to transfer data. A webbrowser, for example, currently always streams a YouTube video directly from YouTube's servers, but in the future it could just use lib*swift* to retrieve the content. YouTube's servers might become a permanent node in the larger network of lib*swift* nodes or they might even no longer be needed.

Positioning lib*swift* as the network to use will require it to perform at least as good as alternative networks. Users cannot be expected to adopt technology with inferior performance just because it has some nice potential future uses, with the possible exception of early adopters. If lib*swift* is to gain traction it needs to compell users to switch. It needs to be the fastest P2P technology currently available.

1.2 Organization

The rest of this report is organized as follows. Chapter 2 details the main research topic of this report and examines its implications. Chapter 3 describes the

[1] The average lifetime of a peer and the observed churn were used, assuming roughly half of the churn to be arriving peers

evaluation framework that has been developed and used in the project. Chapter 4 describes the environment for the experiments and introduces all the clients that have been evaluated. Chapter 5 describes the actual experiments and their results. Chapter 6 finishes with conclusions, trying to answer the question whether lib*swift* can be made the fastest P2P client, and some recommendations for future work.

Chapter 2

Problem Description

This chapter describes the main research goal of this report. The first section introduces the research question and explores its implications. The second section explores the requirements for a testing framework that is needed during the project and discusses some previously existing frameworks.

2.1 Research Question

Can lib*swift* be made the fastest P2P client currently available?

Answering this question requires not only giving proof that lib*swift* is faster than other clients, but also requires identifying where lib*swift* performs suboptimally and improving that. Both of these requirements mean that some way of performance evaluation is needed, both for lib*swift*, in order to find bottlenecks and measure its performance, and for other P2P clients, to compare the results to those of lib*swift*. To improve the speed of lib*swift* the results of the evaluation can be analysed to identify bottlenecks, which can then be improved upon. This leads to the following (idealized) development cycle:

- Performance evaluation
- Performance analysis
- Bottleneck identification
- Improvement
- Performance evaluation

The last performance evaluation not only verifies the improvement but will also be used as input to the next cycle which can then directly continue with the performance analysis.

This report focusses on the performance evaluation of lib*swift* and the comparison with other clients. R. Petrocco et al have been using the results to go through

the complete development cycle, improving lib*swift*'s performance. Their results will be published in [23]. This report also does not include streaming video, but focusses on basic file transfer.

For the comparative performance evaluation two important questions remain: how to evaluate the performance of the clients, and what is "good performance" in the first place?

2.1.1 Performance Evaluation

How to evaluate the performance of lib*swift* and a P2P client in general?

Answering the latter part of this question means answering both, since not only is lib*swift* a P2P client, but any results of the performance evaluation of lib*swift* need to be comparable with the results of the performance evaluation of other clients, which means they should be tested in the same way. A generic way of testing P2P clients is required.

A generic framework that would allow repeating an automated test for different clients and that reports detailed results would satisfy both the need to compare different clients and the need to identify bottlenecks in lib*swift*. Such a framework would be the answer to this question.

Several existing frameworks for testing lib*swift* and other P2P clients have been reviewed and a new framework has been developed for performance evaluation.

2.1.2 Good Performance

What are the characteristics of a P2P client with good performance?

The naive answer would be: fast download speed. But a simple webbrowser often gives better download speeds than P2P clients and yet people use P2P clients.

Exactly what is considered good performance depends on the usage scenario. The time to deliver content is of course a major factor. But according to Cisco [7] the amount of traffic coming from non-PC platforms will rise significantly. Such devices often have more resource constraints than a PC, creating a preference for software that uses resources sparingly. The vision of streaming all of YouTube via a P2P network also requires the technology to work with vast amounts of data and large networks.

The assessment in this report focusses on time to download, CPU usage, memory usage, large amounts of data and large networks.

2.2 Evaluation Framework

This section describes an evaluation framework as would be required for testing P2P clients. Section 2.2.1 details the requirements for the framework and in Section 2.2.2 existing frameworks are evaluated according to those requirements.

2.2.1 Requirements for the Framework

It was anticipated that a testing framework would either be chosen or (partially) developed. To support this choice a number of requirements for the framework have been formulated. Each requirement is detailed below together with its rationale.

Simplicity. A testing framework is ideally usable by anyone. This means that it must not be too complex. The framework has little dependencies and can be used without much initial work. Good documentation, both in the form of documented code or configuration and in the form of user manuals, is imperative.

Maintainability. The testing framework is going to be around longer than just one project, hence it needs to be maintained. Since maintenance on messy or unintelligable code is very expensive, the framework should be clean and understandable. Documentation of the code is an important factor for this requirement.

Extendable. lib*swift* is constantly being extended and it is anticipated that this will be the case for some time. It is likely that new features will need to be tested and possibly new features are needed in the testing framework. Having an extendable testing framework would help in testing such extensions. An extendable framework would also facilitate a more heterogeneous setup of experiments.

Support both local and remote hosts. Initial debugging is often done on the local machine for a number of reasons, while large tests require the use of multiple remote hosts. Both should be supported.

Support for (bad) network conditions. Testing using an ideal network is nice, but networks are not ideal. In order to test a client well one will need real world conditions at some point, or at least an emulation of them. It should be possible to introduce bandwidth limiting and some network errors.

Single command. The testing framework is ideally both idiot proof and strong. This means that it can do a lot, but also that it works out of the box. A single command such as "./doTest" would be an ideal start.

2.2.2 Existing Frameworks

At the start of the project a number of testing environments for lib*swift* were available: manifold (and several incarnations of it), NAT traversal and the P2P testing framework. Pioneer, a partner of the Tribler group in the P2P-Next project [21] with whom a settopbox is being developed that runs lib*swift*, has also developed their own internal testing methods for lib*swift*. Each of these will be discussed briefly below.

manifold is the default testing framework for lib*swift*. Manifold supports acquiring the latest sources, automated setup of the environment, building the code, running multiple lib*swift* nodes, parsing their logs, rendering graphs using gnuplot [16] and outputting the results in HTML format. It also supports netem [19] for network emulation, given that the user has sudo [33] rights on the target machines. As such, manifold supports almost all required functionality, handling everything from source to analysis.

The manifold code is written using bash [14] scripts. It contains many hard-coded decisions and parts of it were written for very specific situations. The code has some modularity built in elegantly using filenames to match modules. It also supports some configuration. Neither the modularity nor the configuration of manifold are very extensive, though. Manifold supports all required functional features but lacks non-functional features, in particular extendability.

NAT traversal is an extension written for lib*swift* that allows automated NAT traversal. A small testing framework was written for this extension, including a number of custom parsers and other scripts. This framework is very specific, not configurable, and uses many different languages. Although it has certainly had its use in testing NAT traversal it meets almost none of the requirements for this project.

The P2P testing framework is a framework developed at University Polytehnika of Bucarest. This framework was developed after their paper [15] describing a modular framework for testing P2P applications. That paper describes exactly what is needed in this project. Their implementation supports running multiple (different) clients on multiple nodes, parsing their logs, rendering graphs from those logs using R [32] and outputting the results in HTML format. It also supports limited traffic control using tc [2] to limit the network speed. Other features are support for virtualized hosts, some form of churn emulation and the possibility to gather external performance statistics, given that the user has sudo [33] rights on the target machine.

The P2P testing framework is written in bash [14] script and has a modular setup for the clients in that it has a large table switching on the client name and calling different scripts from there. Tests using the framework are rather well configurable. The P2P testing framework is a good start on implementing the ideas from [15], but not everything that is described in the paper has been implemented. For example, the authors of [15] have stated a common interface for nodes as a goal, but the only common interface they implemented is SSH. Nonetheless, a lot of the paper's intentions have been implemented in the framework.

Given that what the paper describes comes very close to the requirements set out for this project, the P2P testing framework is a good starting position for further development. Using the current implementation directly is not viable, though: the introduction of several new extension points, such as nodes, would require al-

most all the code to be rewritten. Writing a new framework based on this existing framework would be faster.

Pioneer's internal methods are very extensive, using among others MongoDB as a document database and Python as the base language for the framework. Given the extensive setup of the framework, however, it was deemed too complex as a basis for a testing framework that can be used by anyone. No further investigation has been done into Pioneer's particular methods.

None of the frameworks discussed meets all the requirements. manifold misses several non-functional features, in particular extendability; the NAT traversal testing suite is far too specific; and Pioneer's internal methods are too complex. The P2P testing framework was implemented after the authors' paper [15], which describes exactly what is needed in this project. The implementation itself, however, is still lacking in extendability. The P2P testing framework has been used as the basis for a newly developed testing framework that is described in Chapter 3.

Chapter 3

Evaluation Framework

The existing frameworks introduced in Section 2.2.2 do not sufficiently meet the requirements for this project. A new evaluation framework was developed as part of this project that is essentially an evolution of the P2P testing framework developed by Milescu et al. The choice to evolve that framework's design and not to start from scratch entirely is based on [15], which describes exactly what is needed for this project. Contrary to this, the actual implementation has been written from scratch, but has borrowed several ideas of the original.

"Evolution" has been a keyword in the development of this framework: the framework has been used in parallel to its development, requiring many incremental improvements over the course of time. Section 3.1 summarizes the decisions made in the course of the project concerning the framework's development. The current state of the framework is described in Section 3.2 and in Section 3.3 an example is given to show how the framework is used in this project. An overview of the incremental improvements is included in Appendix D.

3.1 Decisions

This section describes the decisions taken during the evolution of the framework. Most decisions are based on practical needs and are a trade-off between different requirements.

Note that two language choices are described below: one for the commanding host, the platform language, and one for the nodes. The commanding host runs the framework itself, which then contacts several other hosts, the nodes, where the actual experiment is run. The commanding host is usually not part of the nodes and only tells the nodes what to do and when.

The platform on which the framework runs is Python [24]. Initially bash [14] was used, but it turned out to have too many limitations. This impacts the Simplicity requirement since it requires Python to be available. This also heavily impacts

the Maintainability requirement since the Python code is much cleaner and clearer. The framework as a whole is also much more robust.

Python was chosen after considering several alternatives, for a number of reasons:

- Python is widely available nowadays. If it is not already pre-installed it is easy to install for anyone with enough knowledge to use the framework;
- Python is a scripted language, meaning that the source code is also the code that gets executed, eliminating a lot of potential problems;
- The framework does not need the best possible performance to be fast enough: a few seconds extra delay for using a scripted language is no problem, as long as the timing of the experiment is not affected;
- Python, when installed, is always available from the command line, contrary to, for example, PHP [31];
- Python is a strong language with useful features for the testing framework's core (object oriented, strong set operations);
- Experience with Python was readily available.

Hosts on which clients are tested, the nodes in an experiment, can be very diverse. A common language is required, though, to write the commands to the nodes in. Bash is commonly available on server platforms and is also available on both the DAS4 system [34] and the hardware available within the Tribler group. For this reason it was chosen as the common language for commands to the nodes, requiring nodes on which tests are run to have bash available.

Simplicity was redefined. The testing framework allows one to describe a complete real world test with clients, files, hosts, executions and more, all of which are quite abstract. The definition language used to configure the testing framework is also necessarily abstract. To be able to use the testing framework effectively one has to be able to describe the scenario one wishes to test, or there is no point in using the testing framework in the first place. Efforts to keep it simple enough to run with almost no configuration were stopped: if the user cannot describe the scenario they want to run, they should not be using the framework. Similarly, if the user cannot get around with a couple of plaintext files, they will not be able to get around with the results, either. Simplicity does not mean anyone can use the framework, but means that anyone who can use the framework can do so without much effort. This is mainly a requirement on the documentation and the declarative configuration language.

Maintainability was divided in two parts. The core of the framework is rather complex: everything is necessarily very abstract and dynamic. Although more complexity is normally deemed less maintainable, it is not considered a problem. The trade-off is between Maintainability and Extendability: in order to have the required generic Extendability some Maintainability needs to be sacrificed. This is compensated for as much as possible by extensive documentation. The modules, on the other hand, cannot compromise on Maintainability: advanced users of the framework are expected to write new modules to extend the framework to their needs. Compromising on Maintainability in the modules means compromising on those users' ability to do so and hence means compromising on Extendability.

Single Command. This requirement was initially introduced to ease a transition from testing with manifold, which does offer such a single command, to testing using the new framework. However, a single command to just start a test turned out to be very artificial: what test would it run? It is not clear to the user, nor is it of any use since it is quite unlikely that the test that would be run by default is the test the user wants to run. Documentation helps the user to set up tests quickly.

3.2 Current Framework

The current framework consists of a core script, several core modules and many extensions to those core modules. Figure 3.1 shows an overview of the core. The core script contains two important classes, the `CampaignRunner` and the `ScenarioRunner`. The `CampaignRunner` holds one `ScenarioRunner` for each scenario in that campaign, each of which are executed sequentially. A `ScenarioRunner` holds a large dictionary of all `coreObject`s in the scenario, which are retrievable by subtype (`builder`, `source`, `parser`, ...) and their unique name within that category. Apart from `execution` all `coreObject` subclasses are abstract and have subclasses that are the actual modules. All of those `coreObject`s know about the scenario they are in; this means each object in the scenario can influence all of the scenario. Most modules interact with multiple other modules. Each core module has a well defined interface for those interactions. Shown in the diagram are the required cross references between modules that are always there:

1. `host` objects can have a `tc` (traffic control) object and know about `file` and `client` objects that will be used on it;
2. `client` objects have a `source` and a `builder`, and can have a number of `parser`s;
3. `execution` objects have a `host` and a `client`, can have a number of `file`s and can have a number of `parser`s;
4. A `workload` object knows which `client`s to apply itself to.

The static classes are always accessible and provide several utility functions, such as logging and meta data generation.

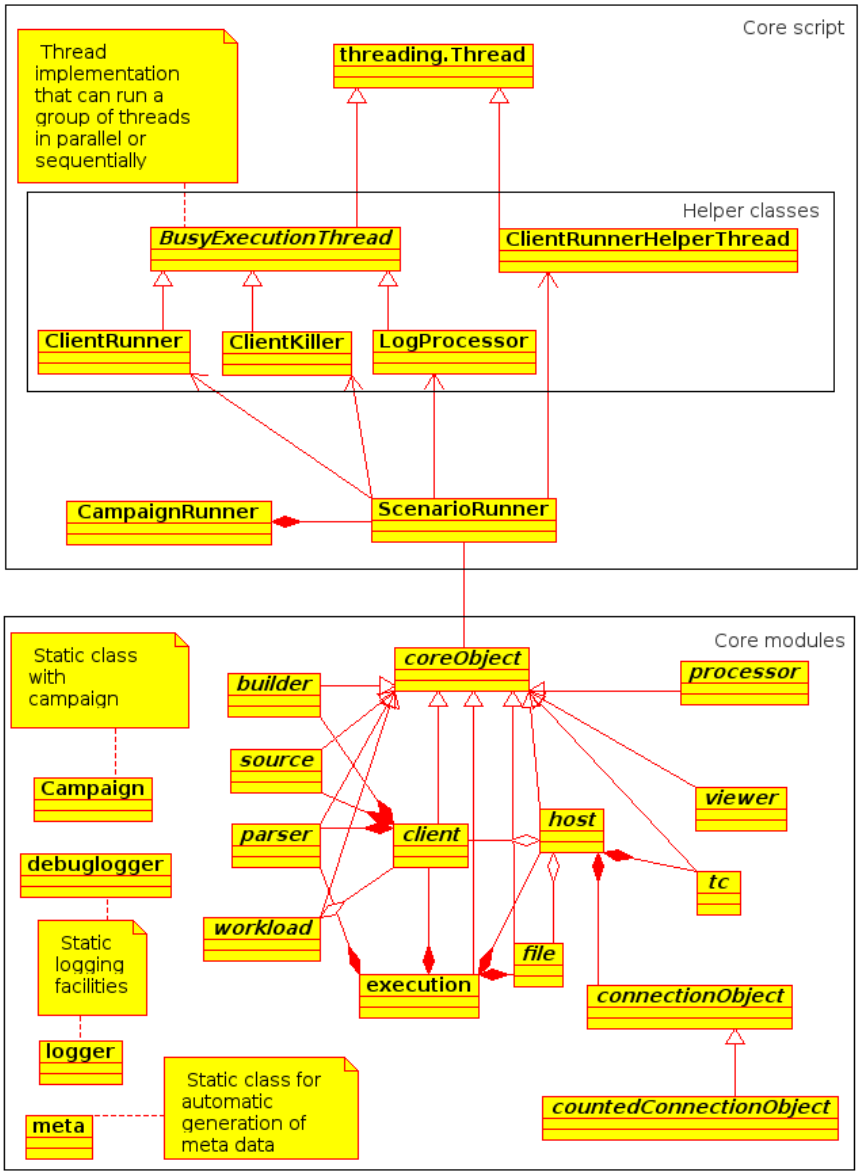


Figure 3.1: Overview of the core of the P2P testing framework.

The core scripts contain the logic to have everything work together and to run a complete scenario from start to finish. The core modules have most functionality that is usually required already built into them allowing their subclasses, the actual modules, to focus on their specifics. Care was taken to have most situations where objects of different types need to work together handled by the core scripts and

core modules.

Each of the abstract subclasses of `coreObject` is an extension point to the framework. This means support can be added for, among others, new P2P clients to be tested, different types of hosts to be tested on, data sources for the data to be transferred in tests and anything from parsing to viewing the logs; just about every aspect of the framework is described by modules. A new module can be written with considerable ease with the exception of `tc` (traffic control) subclasses, due to the hazardous nature of emulating (faulty) network conditions, and `host` subclasses, due to the complex nature of generic and transparent host communication. Besides being easy to extend the framework includes a lot of modules by default, providing both a framework that is usable out of the box as well as some good examples for implementing new modules.

The framework includes many features that have been used directly or indirectly during the project. See Appendix A for a complete list. For this report the most important features are the accuracy of client timing — clients are started within milliseconds of the time they are configured to start — and the generic support for monitoring CPU and memory usage of a client.

3.2.1 Repeatability and Variance

The reasons a framework was required in the first place were the repeatability and comparability of tests. The comparability is provided by running the same experiment in the same environment using different clients of which the output is processed to be in the same format. This is supported by the framework: just replacing the client and parser module changes the client that is tested without affecting anything else. The repeatability is provided by controlling all aspects of the experiment and setting them up the same way every time an experiment is run.

To verify the repeatability provided by the framework a small experiment was done where the libtorrent client (see Section 4.3.5) was used to download the DVD image of Ubuntu 10.04.4 using the BitTorrent network. This experiment consists of one seeder, one leecher and a BitTorrent tracker, all on different nodes inside the local network in the DAS4 system (see Section 4.1). The experiment has been repeated five times.

The download progress of the five runs is shown in Figure 3.2. The times that were measured to download the DVD image inside the local network were 24, 25, 26, 25 and 22 seconds. The mean of these measurements is 24.4 seconds. The mean absolute deviation in these measurements is 1.12 seconds, or 4.5% of the mean time.

To compare this controlled experiment with the uncontrolled world outside the DAS4 the same experiment was repeated, only without the seeder or BitTorrent tracker inside the local network. This time the original BitTorrent trackers in the torrent metadata file were used and hence the seeders were those that happened to be present on the internet at that time. Note that the choice for the Ubuntu DVD image was made because it is usually well-seeded: downloading the DVD

image with a regular BitTorrent client showed tens of seeders being active. This experiment was also repeated five times.

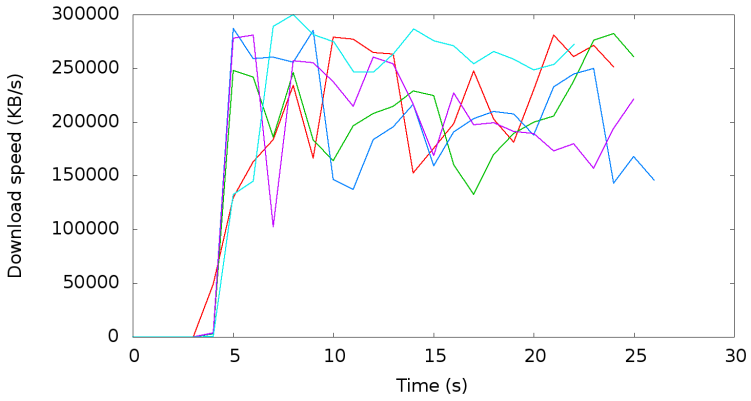


Figure 3.2: Progress of retrieving the Ubuntu 10.04.4 DVD within the local network: 5 measurements with similar patterns.

The download progress of the five uncontrolled runs is shown in Figure 3.3. The times measured to download the DVD image from the global internet were 264, 225, 171 and 169 seconds. The fifth measurement only received 9.47% of the file in 300 seconds, showing a rather constant download speed all that time. This measurement could be extrapolated to a downloading time of about 3000 seconds. Not considering the extrapolated measurement the mean time to download is 207.25 seconds. The mean absolute deviation in these measurements is 37.25, or 18% of the mean time. That is not taking the extrapolated measure into account.

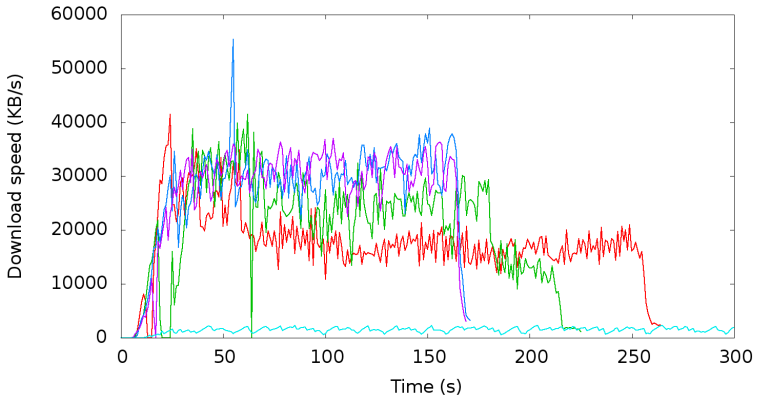


Figure 3.3: Progress of retrieving the Ubuntu 10.04.4 DVD from the internet: 5 measurements with rather different patterns.

Comparing the mean absolute deviation in the measurements yields only 4.5%

deviation for the controlled, internal experiment and 18% deviation for the uncontrolled experiment. This already shows the repeatability introduced by the controlled environment. It should be noted that the libtorrent client only outputs data every second, which means the resolution of the measurements is whole seconds. Since the complete time of a single controlled run is only about 25 seconds that resolution is quite large. However, even if the deviation of each measurement in the controlled environment would be increased by one second the mean absolute deviation increases to only 8.6%, which is still considerably better than 18%.

Another way to see the repeatability is by looking at the download progress over time in the measurements. Figure 3.3 shows the measurements where the DVD is retrieved from outside the internal network. Some of the downloads progress similar, downloading at constant high speeds, but others show decreasing speeds, speed cut-offs or just low speeds. Figure 3.2, however, shows a similar pattern for each of the downloads made from the internal network. The randomness introduced by the P2P network is clearly visible, but the pattern is the same: high speeds revolving around the same average speed.

Finally an aspect of repeatability needs mentioning that is not visible in the above experiments: timing. Imagine an experiment using one seeder and twenty leechers. The leechers start one by one, one each 20 seconds, the first 20 seconds after the seeder started. Each leecher quits exactly 60 seconds after it started. The seeder will also quit exactly 60 seconds after it started. The seeder has one file that takes 30 seconds to transfer from one machine to another and each leecher will try to download that file. Does the last leecher receive the complete file? It should be obvious that by the time the last leecher starts, the seeder is long gone and, in fact, any leecher should see at most three peers running at any given time. The timing of such a scenario is of great importance — allowing a second more or less for the seeder or any of the leechers could already severely skew the results. To ensure such precise timing automation is required.

3.3 Framework Usage

To show how the framework functions for an end-user, and also to show how it was used in this project, a small example experiment was done. The example experiment consists of a single seeder with a single file (fake data, 10 GB) and four leechers that are started 30 seconds after the seeder started. The leechers are divided over 2 nodes and try and download the 10 GB file. The complete experiment is set to stop after 60 seconds. The experiment was run once using lib*swift* with 8 KB blocks and once using libtorrent.

The input of an experiment using the framework consists of the modules needed by the experiment and the configuration files. All modules used in this experiment are available by default. The complete configuration is given in Appendix B. Placing the files described there in the `TestSpecs/` and `TestSpecs/scenarios/` directories, one could run

```
./ControlScripts/run_campaign.py TestSpecs/example_experiment_campaign
```

This command runs the campaign, which consists of two scenarios, first checking their configuration and then running them sequentially. After the campaign is finished the framework will tell where the results are. This directory contains a structure under which each scenario has its own directory with the complete `scenarioFile` and subdirectories `executions/`, `processed/` and `views/`. Inside the `executions/` directory are all the files for each individual execution, which is a single execution of a single client on a single host. The example scenarios each have six executions. Both raw logs as built while running the client and parsed logs obtained by running the parsers on those raw logs are present in the execution directories. The `processed/` directory contains post-processed data, such as graphs, statistics and other information. The `views/` directory contains any views, which are combinations of any raw and processed logs as well as post-processed data that give a complete view of a scenario's results. In the example experiments the `collection.html` file and its accompanying thumbnails can be found in the `views/` directory.

The format of the raw logs depends entirely on the client but, in the case of the default modules, always includes the upload speed, download speed and progress of the files in one form or another. All the provided client specific parsers convert this into a more usable format: relative time in seconds (starts at 0), progress in percentage, upload speed in KB/s and download speed in KB/s. The CPU/memory usage log contains very raw numbers. The provided cpulog parser converts these into a more usable format: relative time in seconds (starts at 0), percentage CPU usage, residential memory usage in bytes and virtual memory usage in bytes. The cpulog parser also creates a file with peak data: the total CPU time in seconds and the maximum of each memory usage.

These consistent output formats are used by the gnuplot post-processing to generate the graphs in `processed/` and by the statistics post-processor to generate statistics on the complete scenario, such as average resource usage, how many leechers completed downloading the file and how long that took them on average. All the data is finally taken together by the htmlcollection viewer to create a complete overview of the scenario's results in an HTML page. Some of the results of the example experiment are included in Appendix B, which also includes excerpts from the various intermediate data files.

For this report a few more steps were taken beyond what the framework does. The framework gives an excellent insight into what happened inside a single run and provides very detailed output that has been very useful in understanding what is really happening with each client. What the framework does not provide is the combination of data from several runs. The experiments in this report all cover a number of runs with a single varying parameter, for different clients. To give an insight into the behaviour of the client when that parameter changes, each run was compressed into a few numbers of aggregated data. The compressed data of all runs was then taken together and plotted against the varying parameter. Although this is mostly a manual task, the framework does facilitate it by providing the statistics

and the uniform data formats, both of which are consistent, machine readable data.

A future extension of the framework could provide this functionality in an automated way. It would be great, for example, to be able to tell the framework to set up a seeder and a number of leechers and to run that scenario ten times, each time increasing the size of the file to be transferred. For this project such an extension was not deemed worth the time: generating the scenarios with changing parameters requires only a little scripting and the same goes for combining the data again. Extending the framework in a generic way to allow for automation of such tasks could take weeks.

Chapter 4

Experimental Setup

This chapter describes the clients and environment used to conduct the experiments. Section 4.1 introduces the environment in which the experiments have been run. Section 4.2 introduces the protocols that have been compared and Section 4.3 describes the clients that have been tested and their particular settings.

In this chapter several different networking paradigms are touched, which have their own terminology. For consistency the terms "client", "seeder", "leecher" and "peer" have been used to refer to, respectively, a software package, an instance of a client that already has all the data and can upload it, an instance of a client that wishes to download the data, and any instance of a client.

4.1 Environment

All experiments have been performed on the TU Delft cluster of the DAS4 distributed supercomputer [34]. Each node of this cluster has a dual quad-core CPU at 2.4 GHz, 24 GB memory and more than enough local disk space for each experiment. The cluster nodes are connected with each other via an internal Infiniband [17] network running at 10 Gb/s. The internal network has been used for all experiments, allowing the clients to use the full 10 Gb/s bandwidth. To make sure no connectivity issues would arise the torrent tracker (see Section 4.3.2), for those clients that need it, was run on a separate node inside the cluster: this ensures it recognises every other node by its internal IP address on which it is reachable from all other nodes.

During all experiments care has been taken not to mix leechers and seeders on the same DAS4 node: a node contains either leechers or seeders. Except for their specific configurations clients start as if they were being run for the first time. In particular this means that any resume-seeding mechanisms are never used and the data to be seeded is always checked for correctness before the actual seeding starts. This leads to long initialization times for seeders when a lot of data is to be seeded. Care has been taken not to start a leecher before the seeder's initialization has completely finished.

The torrent files needed for BitTorrent clients are generated with 1MB chunk sizes and are updated in each experiment run to include exactly one tracker: the instance of the torrent tracker that will be run during the experiment. Smaller chunk sizes have been tried to achieve a better comparison with lib*swift*, but not all torrent clients accept such low values. This gives the BitTorrent clients a distinct advantage when looking at the time needed for checking the correctness of data — the block size will be shown to have a severe impact on that.

Generated fake content is used to ensure real disk I/O while avoiding significant data transfers between the commanding host and the DAS4 nodes during experiment setup. The data is structured such that it is much cheaper to generate locally than it is to transfer or even store, so it is highly unlikely the data would be shared or requested by anyone outside these experiments. This minimizes the chance of pollution from the outside world. The data is also structured such that the speed of integrity verification is the same as with real data: the data is non-trivial and non-repeating.

Swarms in the experiments have always been kept to the size of 1 file and hence the terms "file" and "swarm" are completely interchangable in these experiments. Swarms consisting of multiple files have no real impact on any of the P2P clients under test: they all regard a swarm's data as one large contiguous string of data blocks. Whether these are stored in one or more files doesn't matter much apart from the time the OS needs to open or close a file. Since the latter is equal for all clients the impact should also be equal for all clients.

Resource measurements in the testing environment have been restricted to CPU and memory usage. Network usage is of course included, but since using the network is the core business of these applications it is not included under resource usage. Power consumption would be great to include, but the testing environment has no way to measure that. Such a measurement would not yield relevant data, anyway: power consumption matters most on mobile devices which contain very different hardware from the high performance machines used in the experiments.

None of the experiments in this project have been repeated, due to time constraints. This means all datapoints have been collected only once and no averages over multiple data points are shown.

4.2 Protocols

This section gives a short description of the protocols that have been used in the experiments. First HTTPS is introduced, which is used for a base comparison, followed by lib*swift* and BitTorrent.

4.2.1 HTTPS

HTTPS [10], HyperText Transfer Protocol (HTTP, [12]) Secure, is widely used on the internet for secure communications. Like unsecure HTTP the protocol is com-

pletely asymmetrical with completely different seeder and leecher[1] software. In HTTPS the leecher software sends a connection request to a seeder which then responds by providing its certificate and public key. The certificate, which is usually signed by a trusted certificate authority or other certificate holder, is checked for validity and trustworthiness, after which some encrypted messages are exchanged to decide on a symmetric encryption method to use during the rest of the session. These initial messages are encrypted using asymmetric cryptography in which the public key of the seeder plays an important role. Together this connection request, certificate verification and key exchange form the connection setup.

After the connection setup the leecher can send a URI to the seeder to request the document referred to by that URI to be sent. The seeder then proceeds with sending the data over the connection, whether this data be the document or an error describing why the seeder will not send the document. All communication over the connection is encrypted, and hence also decrypted at the receiving side, with the symmetrical encryption method and key that were decided on during the conection setup.

Several optional parts of HTTPS are often used to further secure the communication, such as authentication methods and key changing algorithms, but these do not change the basic way the protocol functions.

4.2.2 lib*swift*

lib*swift* considers the content it shares, usually a single file, to be a contiguous stream of blocks of equal size, 1 KB in size by default. A Merkle Hash Tree (MHT) [25] is constructed from these blocks using the SHA-1 [20] hashing function. The MHT is constructed by hashing each block to form the leaf nodes of the tree, and by having each non-leaf node be the hash of the concatenation of its children. Zero-hashes, a 20 byte string of zero bytes, are used for leaf nodes that don't have a corresponding block in the data file (most data files do not have a size equal to a power of 2) and the parent of two zero-hashes is also set to be a zero-hash. A file in lib*swift* is always identified by its root hash, the root of the MHT, which is also the only metadata sent to a lib*swift* peer to request (part of) that file.

The drawback of using an MHT is its significant size. It consists of a full binary tree of hashes. That means $2 \cdot n - 1$ hashes of size h for n leaf nodes. The size of a hash is constant in lib*swift*: $h = 20$ bytes, the length of an SHA-1 hash. The number of leaf nodes n depends on the size of the file f and the size of a single block b: $n = 2^{\lceil {}^2\log \frac{f}{b} \rceil}$. In other words: the closest power of 2 larger than the number of blocks in the file. For the default block size of 1 KB and a 700 MB file, for example, the number of blocks would be $n = 2^{\lceil {}^2\log \frac{700\text{MB}}{1\text{KB}} \rceil} = 2^{\lceil {}^2\log(700 \cdot 1024) \rceil} = 2^{20} = 1048576$ and the size of the MHT would be $h \cdot (2 \cdot n - 1) = 20 \cdot (2 \cdot 2^{20} - 1) \approx 40 \cdot 2^{20}$ bytes or about 40 MB.

[1] When talking about HTTP and HTTPS one usually refers to the seeder and leecher as the server and client, respectively

Connections are built in lib*swift* by contacting another peer and by sending the root hash the contacting peer is interested in. Both will assign a channel number to the connection and tell each other about that; the channel number of the other party is always included in packets sent to that party to tell them what the packet is about. Data can be exchanged once the channel is established. This data can exist of specific data blocks or small pieces of metadata that are required to verify the integrity of received data blocks, such as a number of hashes from the MHT. All data transfers are by design exactly one packet in size, which ideally includes the packet sizes of underlying protocols. This restriction ensures that loosing a single packet has no larger impact than just that single packet. If, for example, lib*swift* would run over a default ethernet network and would use 16 KB block sizes, each ethernet packet would contain rougly 1 KB of a block. Loosing a single ethernet packet would in that case mean loosing the whole 16 KB block which hence has to be retransmitted entirely, wasting 15 KB of bandwidth.

When a lib*swift* peer receives a data block it verifies the validity of that block. This is done by reconstructing part of the MHT: the SHA-1 hash of the block is calculated and is combined with hashes that are already known, such as the hashes of nearby blocks and the hashes of inner nodes of the tree that were also received from the sender. Only if the result of this block's hash and all the known hashes combined can be verified to yield the root hash of the file, the block will be accepted.

lib*swift* peers have the ability to inform their peers about other peers they communicate with, allowing those peers to contact each other as well.

4.2.3 BitTorrent

BitTorrent [6] also considers the content it shares to be a contiguous stream of blocks of equal size, though usually block sizes are much larger than what lib*swift* uses: 256 KB, 512 KB, 1024 KB, etc. A metadata file is created, called a metainfo file or .torrent file, which contains the names of the files, the size of a single block, the tracker addresses and most importantly the SHA-1 [20] hash of each block. A BitTorrent peer always has the full .torrent file available before transferring the data, either as a seeder or as a leecher, so no metadata needs to be transferred between BitTorrent clients. A torrent is identified by the SHA-1 hash of the critical part of its metainfo file, also called the info-hash.

The size of a BitTorrent metainfo file depends mostly on the size of the collection of hashes, which in turn depends on the size of the file f and the chosen size of a block b. For each block in the file a single hash of 20 bytes is included, leading to a total size for the collection of hashes of $\lceil \frac{f}{b} \rceil \cdot 20$ bytes. For a block size of 1 MB and a 700 MB file, for example, the collection of hashes would be $\lceil \frac{700\text{MB}}{1\text{MB}} \rceil \cdot 20 = 700 \cdot 20 = 14000$ bytes.

To know other peers that take part in a torrent swarm a BitTorrent peer first contacts one or more BitTorrent trackers to which it sends a request to receive a number of peers for the torrent it is interested in. This request also registers the

peer at that BitTorrent tracker as partaking in that torrent swarm.

Connections are built in BitTorrent by contacting another peer and by sending the info-hash the contacting peer is interested in. Both will exchange information on their progress on the file and will then continue to keep each other informed about whether they are interested in the other's data and can exchange data by requesting a part of the torrent which is then sent in return. The connections are normal TCP connections, which include the usual protections against packet loss.

A BitTorrent peer usually verifies the data it receives by calculating the SHA-1 hash of a block and comparing that to the collection of hashes in the metainfo file.

BitTorrent peers do not exchange information about peers they communicate with: all peers are expected to contact the trackers in the metainfo file, instead.

μTP

The μTorrent Transport Protocol, or μTP [3], is a UDP implementation of the BitTorrent protocol, designed to allow for better use of the network, while also competing less aggressively with other network traffic. BitTorrent using μTP works the same as the normal BitTorrent protocol.

4.3 Clients

Several clients have been tested to not only assess the performance of lib*swift*, but also to compare it to the performance of other (P2P) clients.

4.3.1 HTTPS — lighttpd / aria2

As a baseline test for throughput performance the HTTP and HTTPS protocols have been considered. HTTP was left out since HTTPS is known to be very efficient already and HTTP completely lacks any type of cryptographic verification, which would be unfair when compared to P2P clients. HTTPS does do some verification: during connection setup the identity of the server is verified and all data after that is encrypted using symmetric encryption and hence has to be decrypted.

Where P2P clients use the same software for seeders and leechers, HTTPS is a classic server/client oriented protocol and is completely asymmetric. Because of this not one but two programs are used to run HTTPS experiments. lighttpd is a stable, fast, small and easily configurable HTTP-server and aria2 is a fully featured multi-protocol downloader with good performance and also easily configurable.

The choice for lighttpd [18] was made after considering the major HTTP servers. While almost every HTTP server with a large install base supports HTTPS and is considered very stable, most of those servers require full-fledged installs and custom configuration, and are considered large pieces of software. These servers perform excellent in their role of webserver, but they are hard to handle when building automatically deployed experiments. lighttpd is about the only one that meets the requirements for the experiments: it is considered very stable (running websites

like YouTube and major torrent sites with thousands of hits per second [36]), it was built to handle stress situations, it is fast, it is small, it is easily configurable, it does not need installation and it needs only a few modules to run in the experiments.

Since the HTTPS test is set up as a competitor against P2P clients, it needs at least support for multiple sources for the same leecher. Aria2 [29] is one of few downloaders that can concurrently handle multiple HTTPS sources for the same download. It is a command-line tool that can be run natively on Linux and can be configured completely using the command line. It can also generate statistics on its progress every second, which is very useful for gathering the required data.

The specific software packages used are lighttpd 1.4.29 and aria2 1.13.0, both built from source with SSL support. For lighttpd the dirlisting, indexfile, staticfile and status modules are included. Specific configuration: lighttpd listens to a port configured in the test description. The openssl [30] program is used to generate a self-signed certificate which is loaded in the SSL engine of lighttpd. A simple loop is used to request the status page every second using wget [13] which creates running statistics of the server progress. Aria2 is configured to allow for as much concurrent sources as available and to use those sources in order, basically ensuring maximum concurrent use of sources, and to keep on trying. It is told to output statistics every second in machine readable format. Aria2 will not check certificates for HTTPS to prevent it from rejecting the self-signed certificates used with lighttpd. None of the other downloading modules for Aria2 are activated. The sources are passed directly to Aria2, which is a bit of an unavoidable cheat given that HTTPS has nothing similar to a tracker: either you already know exactly where to look, or you never will.

It should be obvious that, while lighttpd/aria2 will support multiple concurrent sources for each downloader and multiple concurrent downloads from each source, they will not exchange peer information, nor will the downloaders exchange information like P2P clients do.

HTTPS is not included in all experiments. In those experiments where a comparison with HTTPS makes sense it has been included and this is explicitly noted in the experiment description.

4.3.2 opentracker

While not really a client under consideration, opentracker [9] is used extensively during the experiments to allow BitTorrent clients to function. It is a BitTorrent tracker that is known to be small, reliable and stable. It can be configured to run on a specific port and just runs out of the box. Opentracker does not have versioning, but CVS revision 1.68 was used.

All torrent files used in the experiments are changed just before being sent to the nodes to point to the tracker that is active for the current test, which points every BitTorrent peer to the running opentracker instance.

4.3.3 lib*swift*

The core client of the experiments, lib*swift* is tested with default settings and different chunk sizes. The client used is a modification to the swift-like-ftp branch [2], SVN revision 27030, which has been expanded to include a swarm manager that can on-the-fly decide to deactive and reactivate swarms depending on which are needed. The modified client is available on github [3], branch WIP, revision 34a21ed4cfdde006ca73df89ee0a5e6969fa5f9b. When seeding lib*swift* is passed the directory to watch for data to seed. When leeching lib*swift* is passed the hashes of the files to retrieve along with the filenames to use for them. One seeder is always instructed to listen to a particular port and all other leechers and seeders are given that primary seeder as the tracker to connect to.

The need for an actual seeder to be the tracker shows a current deficiency in the implementation of lib*swift*: a swift process will not do peer exchange for swarms it is not part of. This is currently work in progress. It does give an advantage to lib*swift* in that lib*swift* peers always know of at least one seeder.

4.3.4 μTorrent

Among BitTorrent clients μTorrent (or uTorrent, as is usually written) [4] is a big player. It is one of the most often used clients [11] and supports many features, amongst which its own μTP [3] download protocol which takes care of improved download speed.

uTorrent is a closed source client, but binaries are freely available. Two versions have been used in the experiments: the Windows version (3.1.3 build 26837), using Wine [37] as a compatibility layer, and the latest native 64-bit Linux client (build 27079). To interact with uTorrent the webui, an HTTP based remote interface for controlling uTorrent, is included and activated — a python script runs the client and communicates with the webui to instruct the client and gather statistics. Automated bandwidth management is disabled, since the client can just go flat out, and both the client and the webui are instructed to listen on specific ports which are determined just before starting the client — uTorrent will not choose a random open port itself and the port of the webui needs to be known to interact with it. Although uTorrent is instructed to monitor a directory for torrent files to download, the torrent files for the tests are actually added by interaction with the webui since there appear to be race conditions with the monitoring function that cause it not to pick up all torrent files.

The Windows version of uTorrent is a very feature rich graphical client which tries to install itself and activates many addons. In order to keep it usable, and not to have it distracted by other things than what it should do, it is kept in check with specific configuration. Disabled are: local source discovery (not only are the tests run on a local network, the tests should remain independent and not find each

[2] http://svn.tribler.org/libswift/branches/arno/swift-like-ftp/

[3] https://github.com/schaap/swift

other), DNA (a browser integration feature), automated NAT configuration (the client should stay internal), any connections to websites or other services such as updates, any automated installation or integration and automated restart on crash (wrests away control from the framework and will break the test since the connection with the framework is gone). Also all histories, external references and internal caches have been cleared. The configuration of the Windows version is done by taking a prebuilt uTorrent configuration file, stripping that of all unneeded elements and checksums and changing it as needed when starting the run.

A last setting on uTorrent that requires attention is the use of the μTorrent distributed hash table (DHT). Using the DHT has an important advantage: it allows the client to discover peers that are not returned by one of the trackers of a swarm. In the experiments in this project, however, the tracker is always present and is set up to return all peers in the swarm. Several important downsides to using the DHT exist as well: connections can be made to the outside world, which might inject data about the test into a network that survives until the next test begins, thus influencing later tests which should be independent. Using the DHT can also create overhead. It is likely that with a large number of files the client might have some noticeable overhead doing the necessary announces and administration. It has been decided no to use the DHT, mainly to prevent pollution of the experiment.

uTorrent has been included in the experiments and some of the test results have been included in this report, but mainly to show their unreliability. The performance of uTorrent under Wine is such that it is assumed to underperform due to its non-native environment. A known bug in the combination also breaks any severe stress tests: with lots of data (about 3GB, maybe less) uTorrent crashes consistently. A repeated run of transferring a 1GB file from 1 seeder to 24 leechers showed the number of leechers that succesfully obtained the file to vary between 12 and 20. Although such results might just be a consequence of how the client operates, that unfavourable conclusion cannot be drawn when the client is not running according to its own specifications (i.e. natively under Windows). The Linux native versions are currently marked alpha status and rightfully so: they perform very poorly.

4.3.5 libtorrent

libtorrent [26] is a BitTorrent library written in C++. Several clients are based on it (they mention 31 public projects) and it supports most of the BitTorrent extensions as well as μTorrent's μTP.

libtorrent was used using a simple runner application, available at github [4], which just starts the engine, adds the torrents to it and then proceeds to poll the engine for statistics to output. Features such as DHT, local source discovery, UPnP and NAT-PMP are not started. See Section 4.3.4 for the reasons not to include them. The DHT has not been included to keep a level playing field. Including is

[4] https://github.com/schaap/p2p-clients

likely to increase the resource usage of libtorrent without additional benefits, but might also pollute the experiments. Only the core engine is used, which also includes μTP — the choice for classic TCP transfers or μTP is internal to libtorrent's core. Specific configuration allows multiple connections per IP and removes limits on the number of active torrents and the number of peers, both active and received from trackers. The version of libtorrent used is 0.16.0.0 revision 4683.

Chapter 5

Experiment Results

Several experiments have been conducted using the P2P Testing Framework to assess the performance of the clients, lib*swift* in particular. This chapter presents the experiments and their results. The environment in which the experiments were run is described in Chapter 4 and the way the P2P Testing Framework was used is described in Section 3.3.

A single download from one location to another is measured first in Section 5.1. This experiment also takes a look at an important parameter for lib*swift*: the block size. Section 5.2 focuses on a more realistic scenario where a popular file is downloaded from many seeders. A flash crowd, an important phenomenon in P2P networks, is created in Section 5.3. Finally two experiments regarding large-scale sharing are discussed in Section 5.4.

The resolution of the data gathered during these experiments is limited. To keep the data clear all data is rounded to the lowest resolution. Most timing data is therefore rounded to whole seconds, the exception being total CPU time which was always available with 0.01 second resolution. Transfer speeds have always been rounded to kilobytes. It should also be noted that clients report their completion as a percentage of data blocks. A 3 MB file, for example, would only contain 3 data blocks for the BitTorrent clients (block size 1 MB) and its completion in a BitTorrent client would therefore have a granularity of about 33%.

A bug in the logic for measuring resource usage means no data on resource usage is available for clients that are run using a wrapper script. The affected clients are uTorrent (both native and under wine) and the seeder of HTTPS, lighttpd. As explained in Section 4.3.4 the results of uTorrent cannot be trusted, which hence also holds for any measurements on its resource usage. The absence of lighttpd's resource usage is too bad, but not a great problem: lighttpd is known for being lightweight and efficient. Knowing that lighttpd was designed for efficiency and a small memory footprint and knowing it runs heavy sites like YouTube and several large torrent websites, it is assumed both the memory footprint and the CPU usage of lighttpd is almost always lower than that of any P2P client's.

5.1 Downloading a File

The most simple use of a P2P client is to transfer a file from one location to another. This experiment uses a 1 GB file that is seeded by 1 seeder and downloaded by 1 leecher. HTTPS is included in this experiment as a baseline comparison.

5.1.1 Blocksize

lib*swift* has an important parameter which can easily be modified: the blocksize. This parameter is of severe impact, as can be seen in Figure 5.1. The transfer of a single 1 GB file takes less time with larger blocksizes, and the resources used by the leecher and seeder, both peak residential memory usage and total CPU time, also become less with larger block sizes.

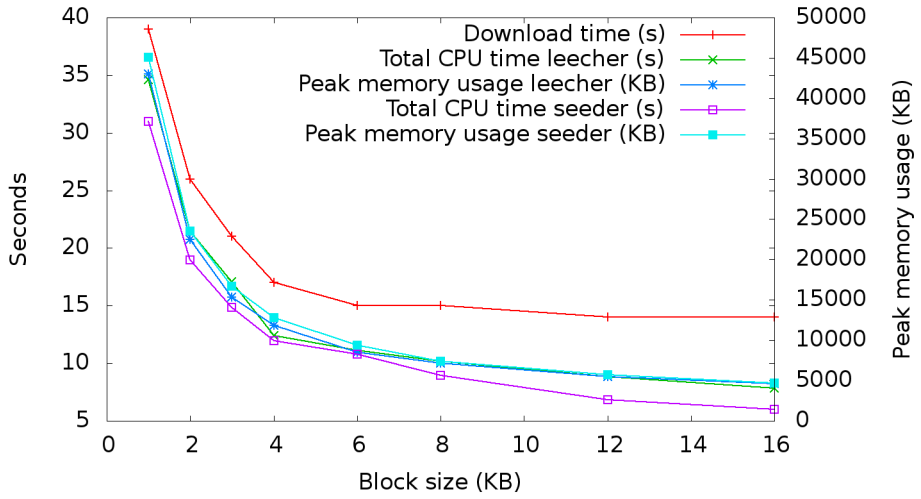


Figure 5.1: Download time and resource usage when downloading 1 GB using lib*swift* with different block sizes.

Block sizes larger than 16 KB have been tested as well, but they simply do not work. Increasing the block size to only 17 KB simply causes the whole transfer to fail consistently, just as it does with even larger block sizes. This limit is due to an internal send buffer in lib*swift* that is set to twice the maximum block size, which is currently configured to be 8 KB. No efforts were made to increase this buffer to allow testing larger block sizes. As explained in Section 4.2.2 increasing the block size much beyond the maximum packet size could lead to a lot of wasted bandwidth. See also the discussion regarding MTU below.

Because of the impact of the block size the rest of the experiments will focus on three different block sizes for lib*swift*: 1 KB, 8 KB and 16 KB. 1 KB, although it's the worst performing block size, is currently the standard block size for lib*swift* and is also the only block size that is smaller than the standard MTU on ethernet net-

work interfaces (1500 bytes). 8 KB is considered because it is the largest block size smaller than 9000 bytes, which is the most often used MTU in networks allowing jumbo frames [35]. 16 KB is considered since it is the best performing block size. The MTU is deemed important in this choice because the design principles behind lib*swift* state that a single lib*swift* packet should never be spanned over multiple packets; and hence should be smaller than the MTU, since a packet larger than the MTU is split in multiple packets before transmission.

5.1.2 Comparison

The basic performance of each client is visible in Figure 5.2. The difference between lib*swift* with 1 KB blocks and lib*swift* with larger blocks is distinct. Interestingly, HTTPS is already slower than the P2P clients. In this simple experiment libtorrent performs (slightly) better than the other clients.

Neither uTorrent client is shown in Figure 5.2 since both failed to transfer the file.

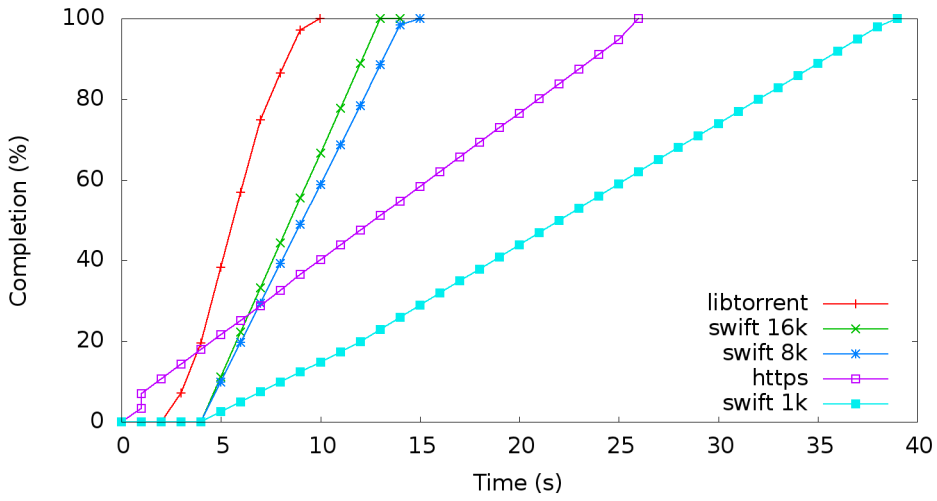


Figure 5.2: Progress over time when downloading a 1 GB file from 1 seeder to 1 leecher.

5.2 Downloading Popular Files

Popular files tend to be well available with many people seeding the file. This experiment uses a 1 GB file that is seeded by multiple seeders and downloaded by 1 leecher. This experiment illustrates how well clients behave with very popular files and also gives a good measure on the download speed a leecher of a client can sustain.

The seeders share the nodes they run on: 3 seeders per node up to 66 seeders,

divided over 22 nodes. For larger numbers of seeders they were equally divided over 22 nodes, with up to 24 seeders per node for the runs with 528 seeders. The impact of having 24 seeders running on one node with only 8 cores has been judged by looking at the results and the resource usage of each seeder: none of the clients show clear signs of slowing down due to sharing all the resources.

The number of seeders will grow from 3 to 528 with increasing intervals. HTTPS is included in this experiment for a baseline comparison, but only partially since larger experiments did not give any new information while still taking time to run.

5.2.1 Resource Usage

Unsurprisingly, most clients are faster when a few seeders are added, but beyond 6 seeders there's no real difference. The only noticeable result is lib*swift* with 1 KB blocks actually becoming slightly slower with many seeders, which suggests some overhead.

More interesting is the peak residential memory usage of the leecher when the number of seeders increases. This is depicted in Figure 5.3. While the other clients are stable in their memory usage, libtorrent needs more memory when increasing the number of seeders but continues to drift between 30 MB and 60 MB with no clear pattern when the amount of seeders increases further. The seeders in the runs with hundreds of seeders, however, more often than not seem not to transmit any significant amount of data. This drifting of libtorrent's memory usage is probably due to libtorrent's leecher only contacting seeders at a maximum rate or even capping the amount of seeders it will contact for a single download. When looking at the runs with lower numbers of seeders, where all seeders transfer significant amounts of data, libtorrent's leecher is estimated to use about 1.5 MB per seeder it contacts.

lib*swift*'s leechers show no visible increase in memory usage when the amount of seeders increases. The constant memory usage is explained purely by the size of the MHT for the data file. Using the formula from Section 4.2.2 and filling in a file size of 1 GB and block sizes of 1 KB, 8 KB and 16 KB, the size of the MHT is about 40 MB, 5 MB and 2.5 MB, respectively. Those number are a constant of about 3 MB less than the ones depicted in Figure 5.3. This shows the significance of the MHT's size.

The peak virtual memory usage of the clients' leechers show that lib*swift* requests only slightly more memory than it has resident, while both libtorrent and aria2 reserve much more: about 250 MB and just over 100 MB, respectively.

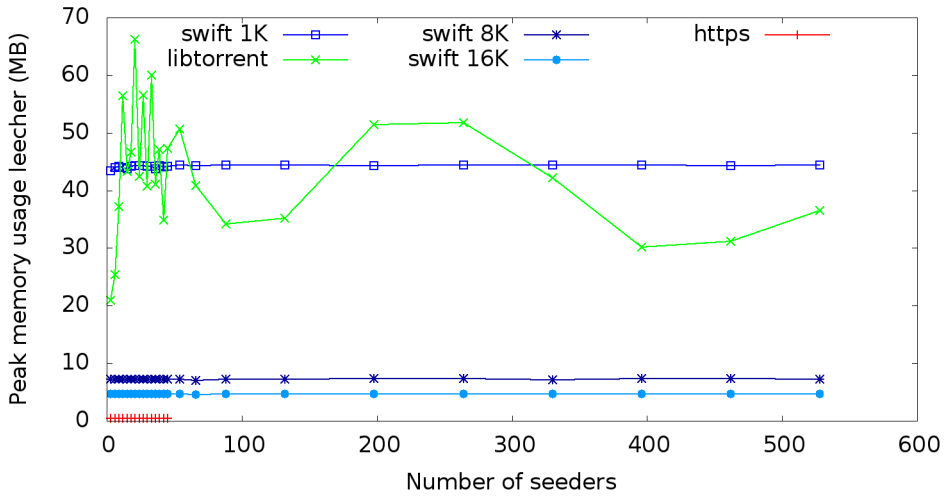


Figure 5.3: Peak residential memory usage of the leecher for increasing numbers of seeders.

5.2.2 Sustained Download Speed

From the tests run for this experiment the download speed a leecher of a client can sustain has been estimated. For each client the data point with the lowest download time was chosen, preferring larger numbers of seeders in case of a tie, and the data on the leecher's download speed during that run, gathered every second, was analysed. To estimate the sustained download speed during a run the graph of the download speed during that run was considered and a start and end point for measurement was chosen, such that anything in between can be considered "high speed downloading". The typical pattern shown by a leecher is (1) being idle for a few second, then (2) starting the download after which the speed increases until it reaches (3) a high speed. It then continues downloading at high speeds, often increasing the speed some more, until (4) the download is nearly done and the speed drops sharply, followed finally by (5) being idle again. These points are illustrated in Figure 5.4. Especially pinpointing (3) is rather subjective, but as a rule of thumb the speed at (3) was required to be at least 50% percent of the maximum speed achieved and near the average speed during the following seconds. Pinpointing (4) is often very simple: the speed curve shows a very sharp drop to 0 at the end and (4) is the last point of high speed directly before that drop. All speed measurements between (3) and (4) were considered, not including (3) and (4) themselves. The average of these measurements was calculated and used as the sustained download speed.

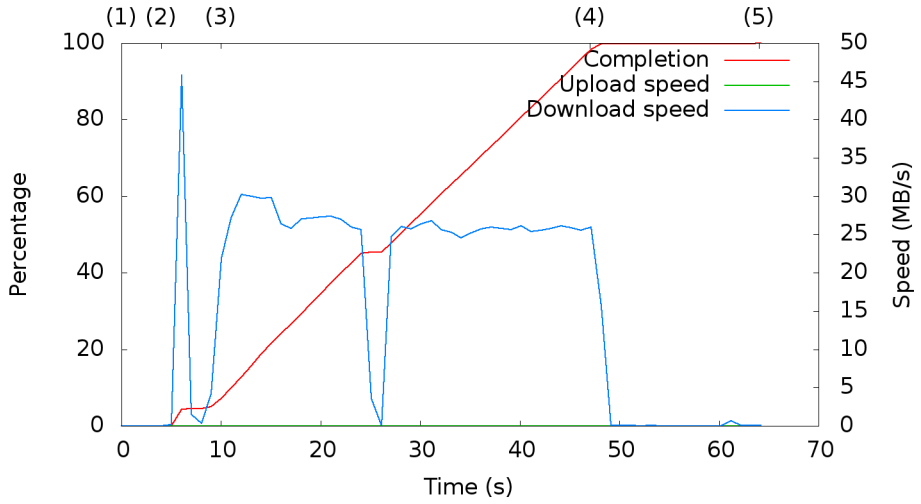


Figure 5.4: Example of the area considered when calculating the average sustained speed. Measurements between (3) and (4) are considered.

Figure 5.5 shows the resulting figures for the leecher's sustained download speed. The very high performance of libtorrent is immediately obvious: well over 1.5 gigabit per second sustained downloading. lib*swift* performs nicely with both 16 KB and 8 KB blocks, almost 1 gigabit per second sustained. lib*swift* with 1 KB blocks, on the other hand, cannot outperform HTTPS.

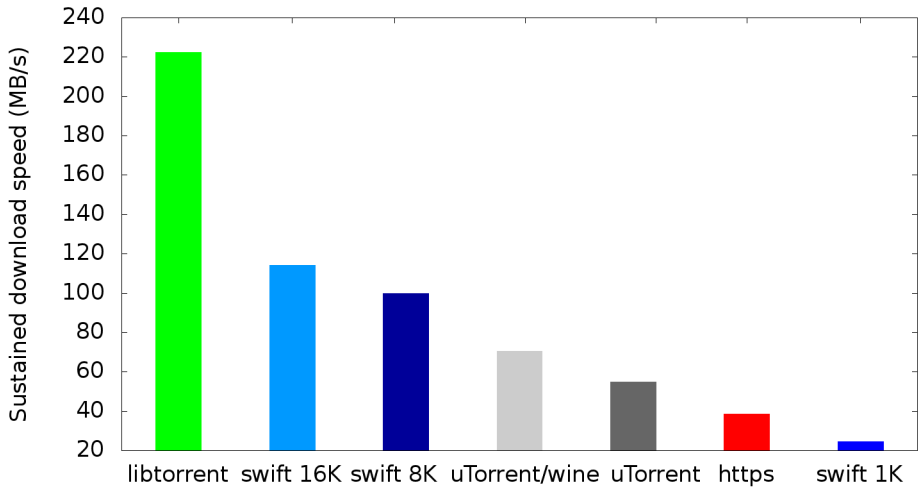


Figure 5.5: Estimated average sustained download speed.

With regard to libtorrent it should be noted that the data, reported by the library itself, has been found to be slightly erroneous at times. At least with small amounts

of data to be transferred, the amount of data reported to have been received was actually smaller than the complete data file that was completely downloaded — that would only be possible if some compression was used, which libtorrent does not do (both BitTorrent and μTP disallow compression). However, the numbers never seem to be very far off and are, in this case, consistent with the measured downloading time. Even so, the error introduced by the estimation is likely larger than the error caused by this data.

5.3 Flash Crowds

Flash crowds are an important phenomenon in P2P networks that can significantly degrade performance [39]. The clients have been stress-tested using an extreme version of a flash crowd: a seeder is being idle and at once many leechers come online and contact the seeder to download the same file. This experiment uses a 1 GB file that is seeded by 1 seeder. Multiple leechers are started at exactly the same time and download the file. This experiment illustrates how well clients behave in a flash crowd and also gives a good measure on the upload speed a seeder of a client can sustain.

The leechers share the nodes they run on: 3 leechers per node up to 66 leechers, divided over 22 nodes. For larger numbers of leechers they were equally divided over 22 nodes, with up to 24 leechers per node for the runs with 528 leechers. The impact of having 24 leechers running on one node with only 8 cores has been judged by looking at the results and the resource usage of each leecher: only lib*swift* with 1 KB blocks uses too many resources to share them effectively in the larger runs.

The number of leechers will grow from 3 to 528 with increasing intervals. HTTPS is included in this experiment for a baseline comparison, but only partially since it requires linearly more time when increasing the numbers of leechers.

5.3.1 Download Time

The most important measurement in a flash crowd is how long each leecher takes to get the file. This is shown in Figure 5.6. HTTPS obviously needs linearly more time to provide the file to more leechers. All P2P clients perform better than this. libtorrent, in particular, shows very nice behaviour. The time needed to download the file decreases until about 15 leechers, at which point it remains stable up to 66 leechers, from where it starts climbing again. libtorrent shows the true strength of P2P networks: the leechers are helping each other to get the file. lib*swift* performs (much) better than HTTPS which means the leechers do help each other. However, the time needed to download the file does increase when more leechers are started at the same time, especially when only a small amount of leechers are involved. This is still a deficiency of lib*swift*. For very large numbers of leechers, however, lib*swift*'s and libtorrent's download times seem to converge again, which means

they scale equally well for very large flash crowds.

uTorrent is not in Figure 5.6. The leechers of uTorrent native failed to retrieve the file in this experiment, no matter the size of the flash crowd. uTorrent wine was only partly tested since it also has many crashing leechers which inflate the amount of time needed for the runs and give very unreliable data. uTorrent wine's performance, as far as leechers were succesful, was similar to lib*swift* with 8 KB blocks, only slightly slower.

Note that lib*swift*'s performance, when using 1 KB blocks, is impacted by the leechers sharing the nodes they run on, leading to a slightly degradation in performance with more than 176 leechers.

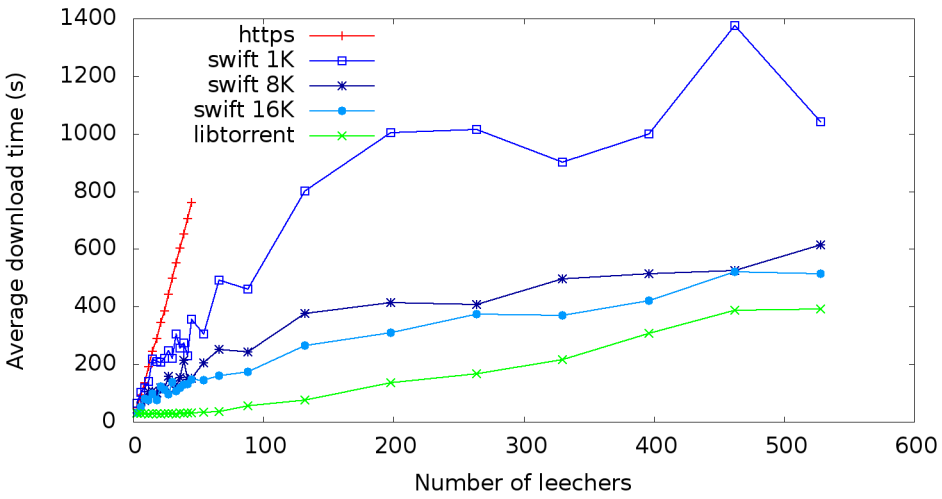


Figure 5.6: Average download time of a leecher when increasing the number of leechers.

Internally, lib*swift* has been capped to a maximum number of connections each peer may make to other peers. By default this number is 20. An initial investigation into the impact of this constant has been done by rerunning the flash crowd scenario, with a flash crowd of 45 leechers, with a lib*swift* client that had this constant set between 5 and 45, with steps of 5. The result is shown in Figure 5.7. This suggests that the choice of 20 is not bad at all: around 20 seems optimal for this scenario. This suggests that the decreasing performance is not due to this constant.

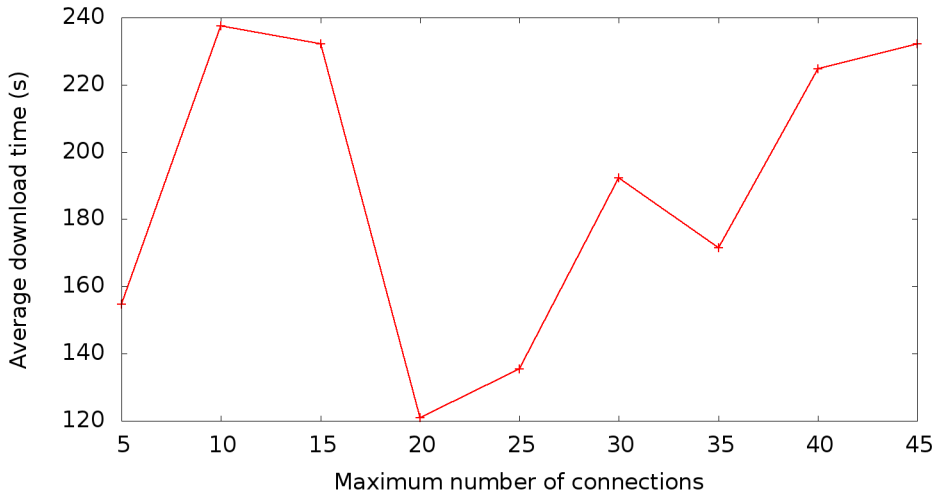


Figure 5.7: Average download time of a leecher in a flash crowd of size 45 when varying the maximum number of connections for each peer.

5.3.2 Resource Usage

The peak residential memory used by the seeder shows that all clients' seeders keep some per-leecher data in memory. This is shown in Figure 5.8. In the case of lib*swift* the bulk of this data consists of several data structures that keep track of the state of the leecher, including data like what parts of the file is requested and what parts still need to be acknowledged. This data depends on the size of the file and the block size. Indeed the memory used by lib*swift* with 1 KB blocks is approximately 8 times as large as the memory used by lib*swift* with 8 KB blocks.

The peak virtual memory needs of the lib*swift* processes is only slightly larger than shown in Figure 5.8, but libtorrent consistently requests about 250 MB of total virtual memory.

5.3.3 Sustained Upload Speed

From the tests run for this experiment the upload speed a seeder of a client can sustain has been estimated. Choosing a single run to analyse is not trivial so a run where all clients perform relatively well was chosen: 12 leechers. The calculation is analog to the one for the sustained download speed, see Section 5.2.2.

Figure 5.9 shows the seeder's sustained upload speed with 12 leechers. libtorrent reaches well over 1 gigabit per second. HTTPS and lib*swift* with 16 KB blocks and 8 KB blocks are very close while lib*swift* with 1 KB blocks can sustain only about half their upload.

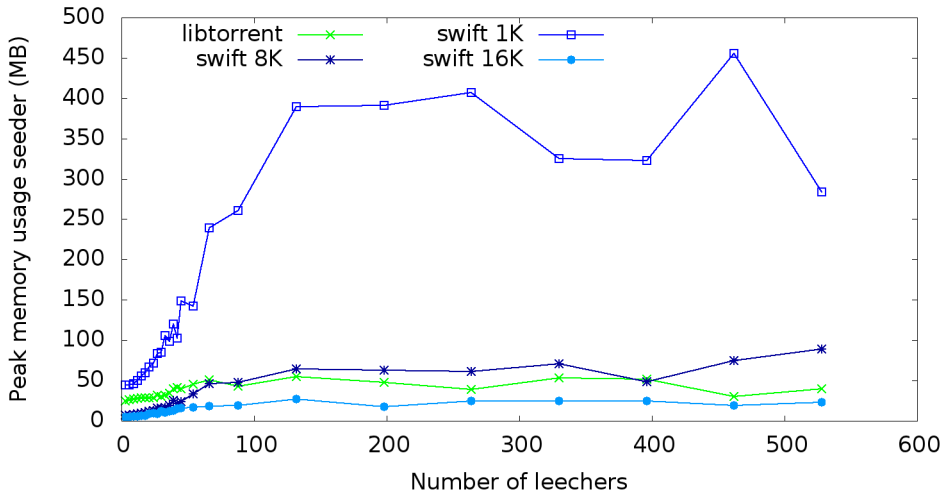


Figure 5.8: Peak residential memory usage of the seeder when increasing the number of leechers.

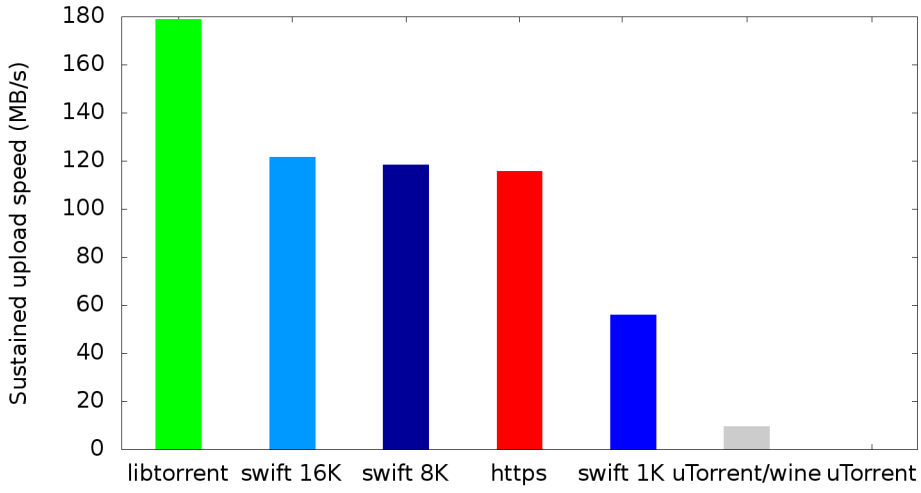


Figure 5.9: Estimated average sustained upload speed.

5.4 Large-scale Sharing

An interesting use of P2P technology would be to provide access to a large archive. Consider, for example, a YouTube replacement built using P2P: this would create a very large archive to share. This section covers two aspects of such large archives: the amount of files, tested using small files, and the size of the files, tested using a set number of files.

5.4.1 Sharing Many Files

This experiment uses 1 seeder and 300 leechers. The seeder seeds an increasing number of files, each 2.5 MB in size. 330 seconds after the seeder starts, the first leecher is started. This period was set after testing how long each client's seeder needs to completely initialize itself and all its seeds; its purpose is to ensure the seeder is always seeding idly when the first leecher arrives. The leechers are started linearly spread over 300 seconds, which is one every $\frac{300}{299}$ seconds, and are expected to arrive at the seeder only seconds later. Each leecher downloads one 2.5 MB file at random from the seeder. The complete experiment lasts no longer than 1200 seconds and the seeder has a 10 second delay to make sure the tracker is up and running, which means the leechers have 560 seconds to finish their downloads from the moment the last leecher starts.

The number of files has been increased from 1000 to 10000 with steps of 1000. To investigate the smaller numbers 10 and 100 files have also been used.

lib*swift* Improvement

This experiment has uncovered a large deficiency in lib*swift*: all required files were kept open during the complete lifetime of the client. The operating system imposes a limit on the amount of open files which caused lib*swift* to crash during this experiment due to opening more than 1024 files, the default limit under Linux.

During this project lib*swift* was enhanced to allow it to close the files of swarms that are not being actively used. For such swarms some metadata — root hash, block size, total size, completion, sequential completion starting at byte 0, file names and the primary tracker — is always kept in memory while the files themselves are being closed and the swarm is almost completely removed. The client still recognizes requests for the swarm and can completely reactivate it when needed. To the outside world this is completely transparent: a leecher can request data from any of the swarms available on the seeder and the seeder can provide it, whether the swarm is active or not.

lib*swift* is still not entirely unlimited in the amount of swarms it can have active at the same time. An active swarm has all its files opened, which is usually three files per swarm: the data file itself, a file that contains a copy of the MHT and a file that contains a bitmap of which blocks have already been received. This means that with a limit of 1024 files per process a single lib*swift* process can have roughly 300 concurrently active swarms. For safety the internal limit has been set to 256.

Resource Usage

In general, both lib*swift* and libtorrent handle many files well. All leechers are able to download the file they request and do so within 10 seconds. uTorrent, on the other hand, cannot handle this experiment: uTorrent native only sees a few of the leechers actually downloading the file and uTorrent wine performs well with only

a few files, but already has almost 90% of the leechers fail with 1000 files. With more than 1000 files uTorrent wine crashes due to the amount of data.

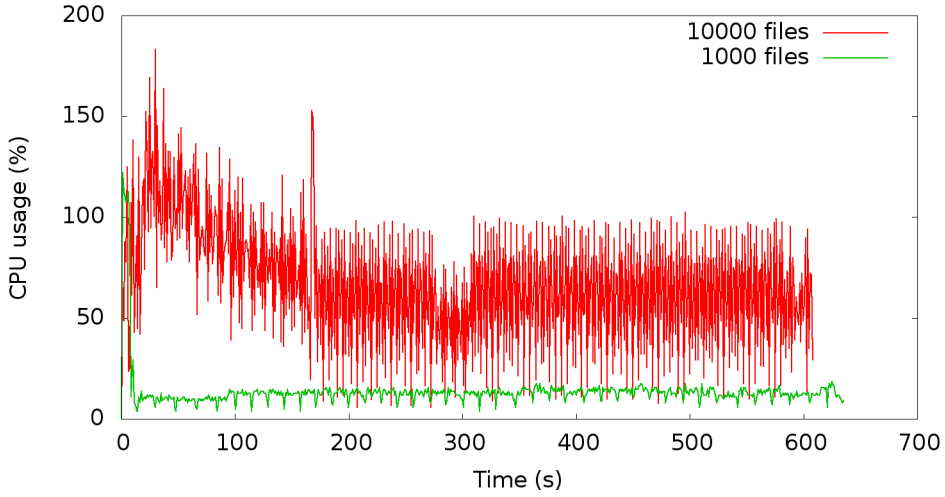


Figure 5.10: CPU usage of libtorrent's seeder for different numbers of files seeded.

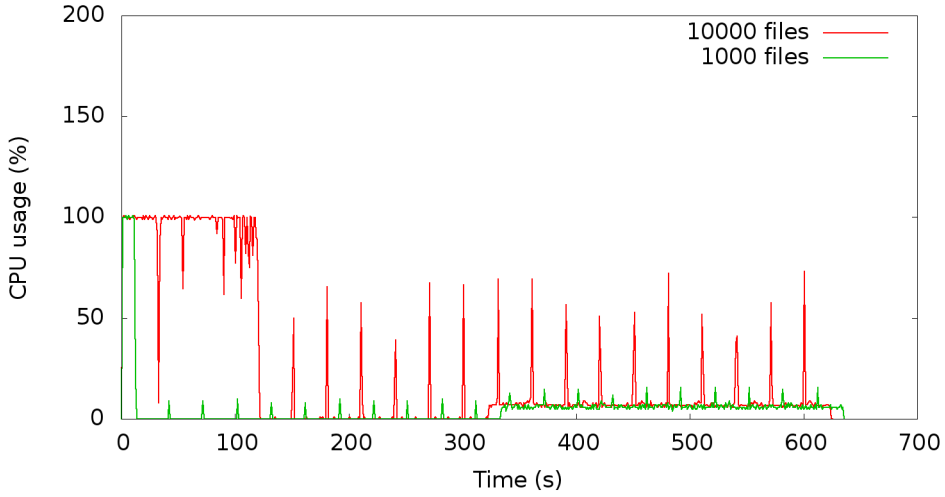


Figure 5.11: CPU usage of the seeder of lib*swift* with 1 KB blocks for different numbers of files seeded.

When the number of files seeded increases every client's seeder needs linearly more time to initialize and also sees increased resource usage. The CPU usage of each seeder increases approximately linearly, but libtorrent's seeder shows much more increase in CPU usage than lib*swift*'s seeder. Figures 5.10 and 5.11 show the profiles of CPU usage of libtorrent's seeder and lib*swift*'s seeder, respectively, for

1000 and 10000 files. The worst case for lib*swift* has been taken for the comparison: 1 KB blocks. Note that the CPU usage is measured in percentage, with 100% being full usage of 1 CPU core; the nodes of the DAS4 each have 8 CPU cores.

Both seeders use a lot of CPU in the beginning during their initialization: they are verifying the correctness of the data they are to seed. Note that libtorrent utilizes multiple cores for this task while lib*swift* shows a rather constant 100% CPU usage, which indicates it uses only 1 core. This is a deficiency in lib*swift* that slows down the initialization of swarms from data that has never before been loaded by the client; data that *has* been loaded before can be initialized considerably faster since the integrity check can be skipped if the metadata of the swarm was saved to disk. Initializing the data can also be done offline, in a different process or even on different hardware, and the data can then be offered to the seeder as if it had been loaded before, thereby skipping the verification check in the seeder process.

libtorrent's seeder reported 100% completed after 163 seconds, which means it has completely initialized itself and all its seeds. lib*swift* is done intializing when its CPU usage drops, which is around 130 seconds. After the initialization has been done the seeder is seeding idly. libtorrent's seeder still uses a lot of CPU during that time. Only at 330 seconds is the first leecher started. This is clearly visible in lib*swift*'s CPU profile.

Also clearly visible in the CPU profile of lib*swift* are periodic spikes. These spikes occur at the moment the seeder decides to check the directory with files it is monitoring for new files. This is not implemented very efficiently and hence gives this small spike, which is otherwise completely unrelated to the actual seeding process. If another method of managing the files seeded by the seeder were to be used, these spikes would not occur.

The average CPU utilization during initialization, idle seeding and active seeding has been calculated for these runs and are summarized in Table 5.1. libtorrent was found to have some increased CPU usage for some time after reporting 100% complete. To make sure only real idle seeding is taken into account in the average, the period between reporting 100% complete and 200 seconds after the seeder started is not taken into account in any of the averages. The active seeding time was taken to last from the first second to the last second where increased upload was reported. Note that for libtorrent DHT was not activated, as explained in Section 4.3.5. It is expected that enabling DHT will increase the CPU usage while seeding both idly and actively.

		Initialization	Idle seeding	Active seeding
libtorrent	1000 files	109.29%	10.81%	14.43%
	10000 files	90.04%	39.32%	46.14%
lib*swift*	1000 files	93.21%	0.11%	5.53%
	10000 files	96.30%	1.51%	4.42%

Table 5.1: CPU usage of the seeder during several phases while seeding 1000 or 10000 files.

Both the CPU profiles and the averages in the different phases show a clear advantage of lib*swift*: virtually no increase in CPU usage while seeding more files. libtorrent already uses about 40% CPU while just being idle with 10000 files.

Figure 5.12 shows the peak residential memory usage of the clients' seeders when the number of files increase. The curves for lib*swift* show an interesting and advantageous side effect of the improvement made to support many files. When confronted with many files some files will be deactivated on-the-fly, freeing their memory for other files. This means that from a certain point the memory usage will no longer grow a lot when adding more files. The amount of memory used by the seeder is composed of the size of the MHT for all the active files and the size of the metadata for all the inactive swarms.

The peak virtual memory usage for lib*swift* is only slightly higher than its peak residential memory usage, but libtorrent consistently requests between 250 MB and 300 MB of virtual memory. This does not grow consistently with the amount of files, though.

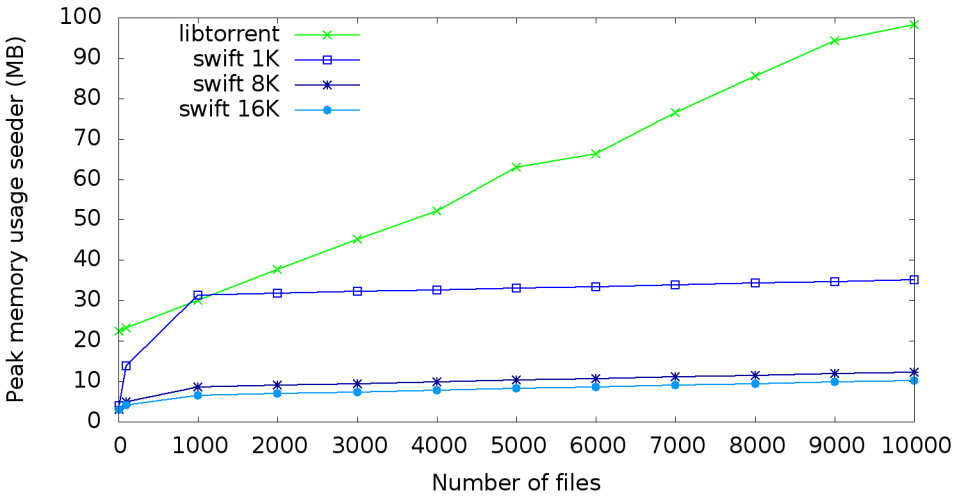


Figure 5.12: Peak residential memory usage of the seeder when increasing the number of files seeded.

5.4.2 Sharing Large Files

This experiment uses 1 seeder and 5 leechers. The seeder seeds 1024 files of increasing size. After an, increasing, safe amount of time that ensures the seeder and all its seeds are entirely initialized the first leecher starts. The 5 leechers are started linearly spread over 300 seconds, which means one leecher every 75 seconds. Each leecher tries to download a random file from the seeder. The leechers are given enough time to finish the download. uTorrent has been considered in the beginning of this experiment, but uTorrent native cannot finish a single download with only 1 MB files and uTorrent wine crashes under the amounts of data in this experiment.

Varying file sizes for this experiment have been used: 1 MB; 10 MB to 100 MB with steps of 10 MB; 250 MB to 1000 MB with steps of 150 MB. The total file size is 1024 times larger, since 1024 files of the chosen size are used. This means that the smallest experiment uses a total file size of 1 GB and the largest experiment uses a total file size of 1000 GB. The unusual increment of 150 MB between 100 MB and 1000 MB is a trade-off between time and accuracy: increasing the accuracy strongly increases the time needed on the DAS4 system to get the data. Since time on the DAS4 was a scarse resource when this experiment was done this choice was made.

Download Speed

While both libtorrent and lib*swift*, with 8 KB or 16 KB blocks, can easily handle the amounts of data in this experiment, libtorrent is a bit slower in transferring the larger files. lib*swift* with 1 KB blocks is about three times slower than when using 8 KB blocks. These download times are shown in Figure 5.13. Inspection of the details of a libtorrent run with 1000 MB files shows no bottlenecks in the resource usage of the seeder or the leechers. This suggest the slowdown is inherent to the protocol, but no further investigation was done.

Resource usage

The CPU usage of all clients' seeders grows linearly with the size of the files. This is almost exclusively the CPU time needed during initialization and is approximately the same for each clients' seeder except for lib*swift* with 1 KB blocks, which uses about 25% more. The peak residential memory usage of the seeders, shown in Figure 5.14, gives a more interesting insight into how the clients handle the large files. The most outstanding feature is the spike of lib*swift* with 8 KB blocks. Multiple reruns have shown this is a rather random effect: it depends on when the operating system decides to start swapping out the memory of the seeder. This figure does not show the actual memory needs of the seeders, but rather the largest amount of memory that was resident at any point during its run. The actual memory needs are much larger, anyway, but most of it is in virtual memory with only small amounts of that virtual memory actually being resident. Figure

5.14 gives a skewed view on how much memory is actually needed by the client's seeder. In Figure 5.15 the profile of the memory usage is shown for lib*swift* with 8 KB blocks and libtorrent, both seeding files of size 1000 MB.

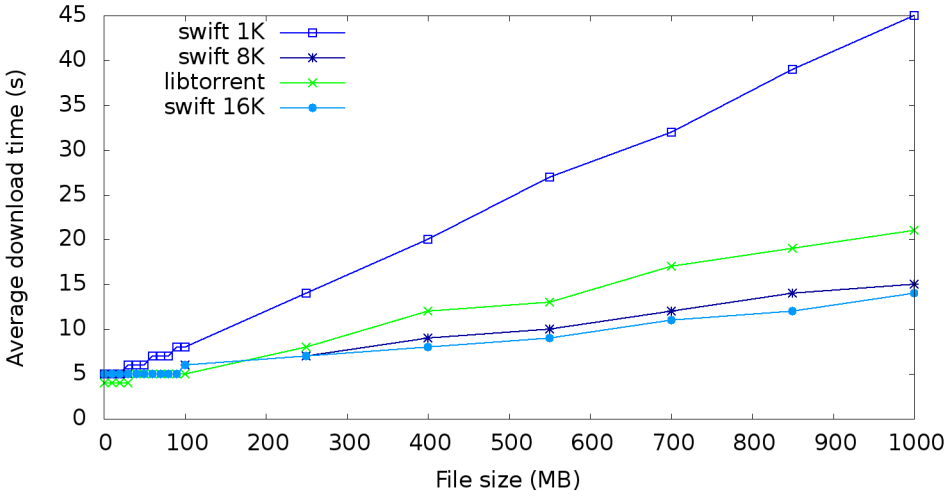


Figure 5.13: Average time needed to download a single file for increasing file sizes.

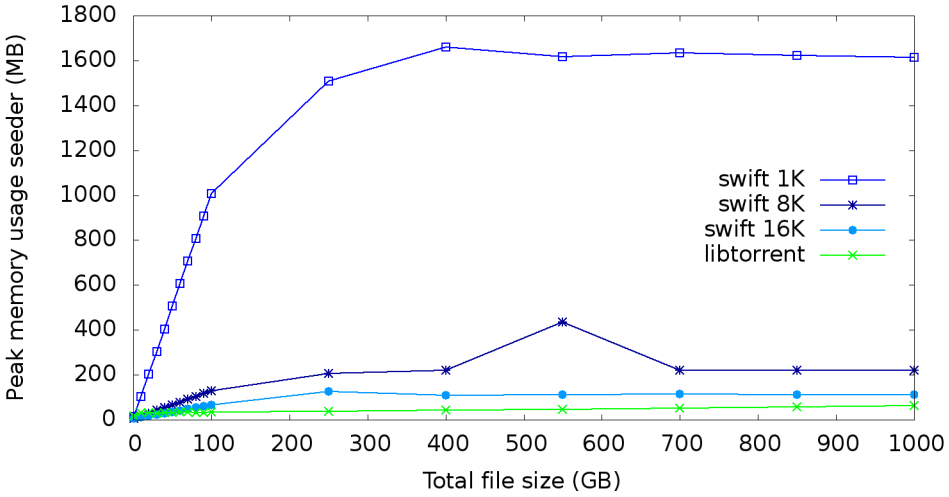


Figure 5.14: Peak residential memory usage of the seeder when increasing the size of the files seeded.

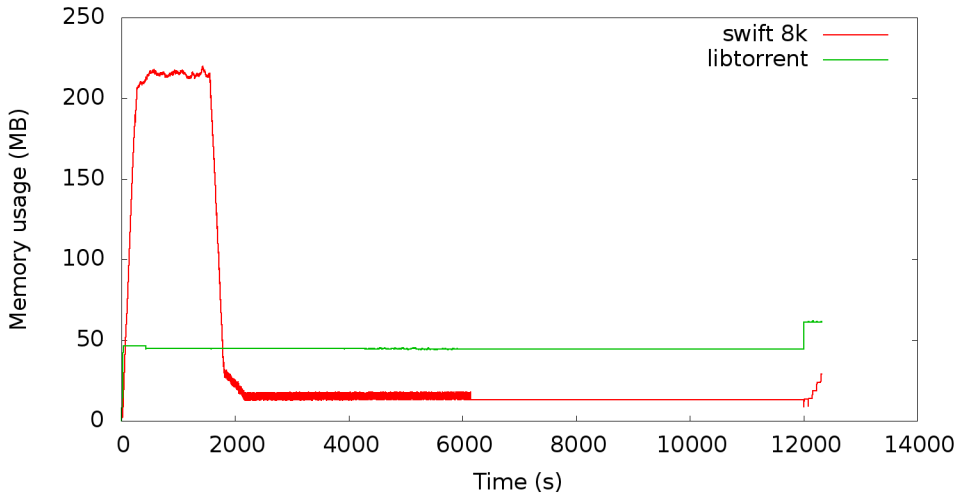


Figure 5.15: Residential memory usage of two seeders for file size 1000 MB.

Figure 5.15 gives insight into how the memory is used by the seeders. lib*swift* is obviously very memory hungry during its initialization. Its virtual memory needs are in fact much higher than shown here: the peak virtual memory usage of lib*swift* grows linear with the size of the files. During the initialization lib*swift* with 8 KB blocks, for example, builds about 1.3 GB of virtual memory and lib*swift* with 1 KB blocks even requests 10.3 GB of virtual memory with 1000 MB files. Most of that memory, however, is backed by files and is not kept resident most of the time. libtorrent requires less virtual memory: it builds up only around 250 MB of virtual memory during initialization, which is later expanded to about 320 MB when it is actively seeding. libtorrent does need much more residential memory then lib*swift* while idly seeding: 45 MB versus only 15 MB. This shows that while lib*swift* is efficient with memory while seeding, it could be much more efficient when initializing new swarms.

A workaround does exist for the initialization phase of lib*swift*: the data can be initialized in an offline process, which can run next to the seeder process or even on different hardware, after which the data and its metadata are both offered to the running seeder process. The presence of the metadata will allow the seeder process to skip the verification of the data and to load the data almost instantly. This is a workaround, however, that only moves the problem away from the seeder process itself.

Chapter 6

Conclusions and Future Work

In this chapter some conclusions are drawn based on this report's results. Some future work is also recommended.

6.1 Conclusions

Can lib*swift* be made the fastest P2P client currently available? To find the answer to that question several experiments have been conducted to compare lib*swift* with other (P2P) clients and itself. lib*swift* has been found to be a good competitor, but one that is still lacking in some departments.

The most important deficiencies are the performance in flash crowds and the memory usage when confronted with large amounts of data. Section 5.3 showed that a flash crowd is not handled efficiently enough by lib*swift*, leading to degrading performance when more leechers arrive. Section 5.4 showed that when lib*swift* is confronted with a lot of data to initialize, it requires a lot of memory. The latter can easily be worked around by having an offline process that initializes the data.

lib*swift* also has some important advantages: when seeding data lib*swift* requires very little resources. This is especially noticeable when large amounts of data need to be seeded: lib*swift* needs a lot of resources to initialize, but once that is done it requires only little CPU and residential memory.

It is important to use the right block size for lib*swift*. Section 5.1 already concluded that 1 KB blocks are not efficient and indeed lib*swift* performs poorly compared to others when using 1 KB block sizes. This means that for now 8 KB blocks are probably the best choice, which also means using a block size that is likely to be larger than the MTU when transmitting data across the internet. It will work, but it breaks the design principles behind lib*swift*.

Is lib*swift* the fastest P2P client currently available? No. But this can certainly come to be in the near future. The experiments conducted for this project have shown only two remaining deficiencies in lib*swift* and have also shown several strong points.

6.2 Future Work

lib*swift* still needs some work. A flash crowd is not handled efficiently enough and memory consumption when confronted with large files is far too much. Tackling these problems, in the order they were given here, should be the first priorities when trying to get lib*swift* ready for widespread use. Several optimizations are also likely to be possible, such as using multiple CPU cores (see Section 5.4). Apart from these problems and optimizations lib*swift* is still very young and small bugs have been found during this project time and time again. A lot of testing will be needed to get lib*swift* to similar maturity as other P2P clients.

The P2P testing framework that has been developed has shown its use and is considered stable and effective. There are, however, still some bugs that should be resolved as well as extensions that would be very useful. Support for Windows nodes would be a major feature, just like support for virtual machines would be.

The experiments in this report compared just a few widely used P2P clients. There are many more clients that could yield very interesting comparisons with lib*swift*; in particular getting valid results for uTorrent, the largest BitTorrent client, would be very valuable. There have also been no constraints on the bandwidth available during the experiments. With the P2P testing framework ready such experiments are within arms reach and could give a broad overview of the performance of currently available P2P technologies.

Bibliography

[1] A. Bakker and R. Petrocco. Peer-to-Peer Streaming Peer Protocol (PPSPP). http://datatracker.ietf.org/doc/draft-ietf-ppsp-peer-protocol/, June 2012. (work in progress).

[2] Bert Hubert. Linux Advanced Routing & Traffic Control. http://lartc.org/, 2012. Retrieved July 19, 2012.

[3] BitTorrent, Inc. μTP. http://www.utorrent.com/help/documentation/utp. Retrieved July 24, 2012.

[4] BitTorrent, Inc. μtorrent, a (very) tiny bittorrent client. http://www.utorrent.com/, 2012. Retrieved July 19, 2012.

[5] Blizzard Entertainment, Inc. Networking help for the Blizzard Downloader. http://eu.battle.net/support/en/article/networking-help-for-the-blizzard-downloader, June 2012. Retrieved July 27, 2012.

[6] Bram Cohen. The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, June 2009. Retrieved August 6, 2012.

[7] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2011-2016. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html, May 2012.

[8] Cisco. The Zettabyte Era. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/VNI_Hyperconnectivity_WP.html, May 2012.

[9] Dirk Engling. opentracker - An open and free bittorrent tracker. http://erdgeist.org/arts/software/opentracker/. Retrieved July 19, 2012.

[10] E. Rescorla. HTTP Over TLS. http://tools.ietf.org/html/rfc2818, May 2000.

[11] Ernesto. uTorrent Keeps BitTorrent Lead, BitComet Fades Away. https://torrentfreak.com/utorrent-keeps-bittorrent-lead-bitcomet-fades-away-110916/, September 2011. Retrieved July 24, 2012.

[12] Fielding and Gettys and Mogul and Frystyk and Masinter and Leach and Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.txt, June 1999.

[13] Free Software Foundation. GNU wget. http://www.gnu.org/software/wget/, August 2010. Retrieved July 24, 2012.

[14] Free Software Foundation. GNU Bash. http://www.gnu.org/software/bash/bash.html, June 2012. Retrieved July 19, 2012.

[15] George Milescu and Răzvan Deaconescu and Nicolae Tăpus. Versatile Configuration and Deployment of Realistic Peer-to-Peer Scenarios. In *ICNS 2011, The Seventh International Conference on Networking and Services*, pages 262–267. IARIA, May 2011.

[16] gnuplot. http://www.gnuplot.info/, March 2012. Retrieved July 19, 2012.

[17] IBTA. About InfiniBand. http://www.infinibandta.org/, 2010. Retrieved August 11, 2012.

[18] LIGHTTPD, fly light. http://www.lighttpd.net/, 2012. Retrieved July 19, 2012.

[19] Linux Foundation. netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, November 2009. Retrieved July 19, 2012.

[20] National Institute of Standards and Technology (NIST). Secure Hash Standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, August 2002.

[21] P2P-Next Consortium. P2P-Next, sharing the next generation of internet TV. http://www.p2p-next.org/, 2012. Retrieved August 11, 2012.

[22] paramiko, SSH2 protocol for python. http://www.lag.net/paramiko/, May 2011. Retrieved July 19, 2012.

[23] Riccardo Petrocco, Johan Pouwelse, and Dick Epema. Performance analysis of the libswift p2p streaming protocol. In *12-th IEEE International Conference on Peer-to-Peer Computing (P2P12)*. IEEE, 2012. (to be published).

[24] Python Software Foundation. Python Programming Language - Official Website. http://www.python.org/, 2012. Retrieved July 19, 2012.

[25] Ralph Merkle. A Digital Signature Based on a Conventional Encryption Function. In Pomerance, Carl, editor, *Advances in Cryptology CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin / Heidelberg, 2006.

[26] Rasterbar Software. libtorrent. http://www.rasterbar.com/products/libtorrent/, 2005. Retrieved July 19, 2012.

[27] Thomas Silverston and Olivier Fourmaux. P2P IPTV Measurement: A Comparison Study. *CoRR*, abs/cs/0610133, 2006.

[28] *swift*, the multipart transport protocol. http://libswift.org/, 2010. Retrieved July 27, 2012.

[29] Tatsuhiro Tsujikawa. aria2, The Next Generation Download Utility. http://aria2.sourceforge.net/, 2012. Retrieved July 19, 2012.

[30] The OpenSSL Project. OpenSSL, Cryptography and SSL/TLS Toolkit. http://www.openssl.org/, 2012. Retrieved July 24, 2012.

[31] The PHP Group. PHP. http://www.php.net/, 2012. Retrieved July 19, 2012.

[32] The R Foundation. The R Project. http://www.r-project.org/, 2012. Retrieved July 19, 2012.

[33] Todd C. Miller. Sudo Main Page. http://www.sudo.ws/, 2012. Retrieved July 19, 2012.

[34] Vrije Universiteit Amsterdam. Distributed ASCI Supercomputer - Version 4. http://www.cs.vu.nl/das4/, 2012. Retrieved July 19, 2012.

[35] Wikipedia. Jumbo frame. http://en.wikipedia.org/wiki/Jumbo_Frames, July 2012. Retrieved July 25, 2012.

[36] Wikipedia. lighttp. http://en.wikipedia.org/wiki/Lighttpd, July 2012. Retrieved July 24, 2012.

[37] Wine HQ. http://www.winehq.org/, 2012. Retrieved July 24, 2012.

[38] YouTube Statistics. http://www.youtube.com/t/press_statistics/. Retrieved July 26, 2012.

[39] B. Zhang, A. Iosup, J. Pouwelse, and D. Epema. Identifying, analyzing, and modeling flashcrowds in bittorrent. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 240–249. IEEE, 2011.

Appendix A

Framework Features

This appendix contains the complete lists of extension points, features and important default modules provided by the framework.

A.1 Extension Points

- `client`, which defines how a client is to be run as a seeder or leecher (examples: lib*swift*, μTorrent);
- `source`, which defines how to retrieve the source code of a client (if any) (examples: local directory, git);
- `builder`, which defines how to build the source code of a client (if not binary) (examples: make, scons);
- `file`, which defines what data is to be transferred (examples: local file, generated fake data);
- `host`, which defines what hosts to connect to and how to operate them (examples: ssh, das4);
- `tc`, which defines a method of traffic control for hosts to create/emulate network conditions (examples: netem);
- `workload`, which defines a type of workload to generate for the clients (examples: linear, poisson);
- `parser`, which defines a parser for logs (examples: lib*swift* parser, CPU profiling parser);
- `processor`, which defines postprocessing for logs and parsed logs (examples: saving the host name, gnuplot scripts);
- `viewer`, which combines logs, parsed logs and processed data into a single view (example: html collection).

A.2 Features

- High modularity — every aspect is modular and every module is loaded at runtime;
- Strong declarative configuration DSL — typical scenario declarations are well under 100 short lines;
- Highly accurate client timing — clients are started within milliseconds of their precalculated start time;
- Parallel or sequential — parallel tasks can be set to run sequentially without losing accuracy with client timing;
- Robust — in the advent of crashes the framework does the best it can not to loose information and to clean up;
- Extensive configuration checking — whenever something is not in order the framework will help resolving it;
- Extensive debugging facilities for writing new modules;
- Support for locally or remotely retrieved and built source code at runtime;
- Support for CPU and memory profiling of clients;
- Support for runtime generation of .torrent files and root hashes;
- Multiple connections per host — in particular dedicated connections to run and monitor clients;
- Possibility to asynchronously send commands to a host.

A.3 Important Default Modules

- Client support for lighttpd/aria2 combination ([18],[29]), libtorrent [26], opentracker [9], lib*swift* and μTorrent [4];
- Support for local files, remote files and generated fake data;
- Host support for SSH which can benefit from paramiko's [22] features;
- Host support for the DAS4 system [34] which can fully automate reservations and uses custom multiplexed connections to allow a very large number of connections over the same head node (requires paramiko);
- Support for gnuplot [16] post-processing, including default scripts to visualize client progress and resource usage;

- Support for netem [19] (linux kernel module) traffic control;
- HTML collection output of scenario results;
- Several predefined workload generators.

Appendix B

Example Experiment

This appendix contains the configuration files and results of the example experiment.

B.1 Configuration

B.1.1 Campaign File

The campaign file `TestSpecs/example_experiment_campaign` describes the campaign.

```
# Two scenarios are described, both last at most 60
# seconds, excluding preparation and cleanup time.
# Both scenarios are described by the concatenation of two
# files: the generic scenario configuration and the client
# specific configuration. Note that those files are all
# expected in TestSpecs/scenarios/.

[scenario]
name=example_using_swift
file=TestSpecs/scenarios/example_experiment_scenario
file=TestSpecs/scenarios/example_experiment_swift
timelimit=60
[scenario]
name=example_using_libtorrent
file=TestSpecs/scenarios/example_experiment_scenario
file=TestSpecs/scenarios/example_experiment_libtorrent
timelimit=60
```

B.1.2 Generic Scenario File

The generic scenario file `TestSpecs/scenarios/example_experiment_scenario` contains most scenario configuration. Only the client specific configuration is missing.

```
# The configuration of the nodes to test on, all DAS4 nodes,
# all to be reserved for 600 seconds:
# 1 node called tracker
# 1 node called seeder
# 2 nodes called leechers
# Note that the reservation will be made as one on the local
# DAS4 cluster
[host:das4]
name=tracker
nNodes=1
reserveTime=600
user=tschaap

[host:das4]
name=seeder
nNodes=1
reserveTime=600
user=tschaap

[host:das4]
name=leechers
nNodes=2
reserveTime=600
user=tschaap

# The data that will be transferred: 10 GB of generated data
# A .torrent file and root hash with 8 KB block size will be
# generated. Both will be stored in caches: generating and
# especially hashing 10 GB of data takes a lot of time. Make
# sure the directories caches/ and caches/torrents/ exist.
[file:fakedata]
name=thedata
ksize=10485760
generateTorrent=yes
generateRootHash=8
torrentCache=caches/torrents
rootHashCache=caches/merkle

# The BitTorrent tracker
```

```
# The tracker is set up to listen on port 16000 and will
# change the .torrent file of file object thedata (the
# generated data above) to point to this tracker.
# The client is already present remotely in my home
# directory under ./opentracker/, so I can just use that
# (remoteClient=yes and location=./opentracker/).
[client:opentracker]
name=tracker
remoteClient=yes
location=./opentracker/
port=16000
changeTracker=thedata

# The executions bind it all together.
# - The tracker client will run on the tracker host.
# - The seeder client will run on the seeder host, is marked
#   as a seeder and will transfer thedata.
# - The leecher client is run twice on the leechers host,
#   and will transfer thedata. This will end up as 4
#   independent execution: 2 nodes and multiplied by 2.
# Note that each execution starts, by default, at 0 seconds,
# which is when the experiment begins. See the workload
# below for how this is changed.
[execution]
host=tracker
client=tracker

[execution]
host=seeder
client=seeder
file=thedata
seeder=yes

[execution]
host=leechers
client=leecher
file=thedata
multiply=2

# The workload will be applied to all (4) executions of the
# leecher client. It is configured to offset them by 30
# seconds and to spread them linearly using a 0 seconds
# interval. Combining they will start at (0*0)+30, (1*0)+30,
# (2*0)+30 and (3*0)+30 seconds. Or: all will start 30
```

```
# seconds after the experiment starts.
[workload:linear]
apply=leecher
offset=30
interval=0

# These are some standard post-processors that save some
# data about the executions.
[processor:savehostname]
[processor:saveisseeder]
[processor:savetimeout]
[processor:savefiles]
# gnuplot post-processing: plots the data output by the
# clients, after it was parsed, in some nice graphs. One for
# the completion/upload/download graph and one for the
# CPU/memory usage.
[processor:gnuplot]
script=TestSpecs/processors/simple_log_gnuplot
[processor:gnuplot]
script=TestSpecs/processors/simple_cpu_gnuplot
# The statistics processor builds a single line of
# statistics for the complete scenario. This is very
# useful when combining the data of multiple scenarios.
[processor:statistics]

# Finally a viewer: this will create an HTML page
# displaying all the output in a user viewable fashion.
[viewer:htmlcollection]

# Note the absence of clients other than the tracker: they
# are configured in separate files.
```

B.1.3 lib*swift* Scenario File

The lib*swift* specific scenario file `TestSpecs/scenarios/example_experiment_swift` contains the configuration for the lib*swift* clients.

```
# The swift clients.
# The clients will be have their CPU/memory usage monitored
# (profile=yes).
# By default only the swift parser would be used for a
# client of type swift; since I use profiling I want that
# log to be parsed as well, so I explicitly tell the
# framework to use both the swift and cpulog parsers.
```

```
# The client is readily available in my homedir in
# ./libswift-git/, so use that.
# The seeder will listen on port 15000 and wait for 60
# seconds before quitting. The leecher is set to expect its
# tracker on port 15000 of the seeder host.
# Both use block size 8192, or 8 KB.

[client:swift]
name=seeder
profile=yes
parser=swift
parser=cpulog
remoteClient=yes
location=./libswift-git/
listenPort=15000
wait=60
chunkSize=8192

[client:swift]
name=leecher
profile=yes
parser=swift
parser=cpulog
remoteClient=yes
location=./libswift-git/
tracker=@seeder:15000
chunkSize=8192
```

B.1.4 libtorrent Scenario File

The libtorrent specific scenario file `TestSpecs/scenarios/example_experiment_libtorrent` contains the configuration for the libtorrent clients.

```
# The libtorrent clients.
# The clients will be have their CPU/memory usage monitored
# (profile=yes).
# By default only the libtorrent parser would be used for a
# client of type libtorrent; since I use profiling I want
# that log to be parsed as well, so I explicitly tell the
# framework to use both the libtorrent and cpulog parsers.
# The client is readily available in my homedir in
# ./libtorrent/, so use that.

[client:libtorrent]
```

```
name=seeder
profile=yes
parser=libtorrent
parser=cpulog
remoteClient=yes
location=./libtorrent/

[client:libtorrent]
name=leecher
profile=yes
parser=libtorrent
parser=cpulog
remoteClient=yes
location=./libtorrent/
```

B.2 Output

Some of the output of running the example experiment will be shown here. Not everything is shown: this example is meant only to illustrate how the framework functions.

B.2.1 Raw Logs

The execution logs are built while the client is running. Some excerpts of the interesting parts of these raw logs are presented below. They can contain any output the client gives, including status information and debug information.

CPU and Memory Profiling Log

This is a very extensive log. Each log entry has three lines: a timestamp, the contents of /proc/PID/stat (where PID is the process identifier of the running client process) and the amount of virtual and residential memory used by that process. See `man proc`[1] for interpreting the second line.

Presented are three entries from the raw CPU log of a running libtorrent client. Each second line has been split into 5 lines due to space constraints.

```
12-08-01 13:06:34.849769874
9448 (libtorrent) S 1 9414 9414 0 -1 8192 1968 0 0 0 1198
  160 0 0 20 0 3 0 492156146 142876672 1750
  18446744073709551615 4194304 4235572 140737488349280
  140737488346240 140737327318301 0 0 6 0
  18446744073709551615 0 0 17 0 0 0 0 0 0
```

[1] http://linux.die.net/man/5/proc

```
139528  7000
12-08-01 13:06:35.870864063
9448 (libtorrent) S 1 9414 9414 0 -1 8192 1968 0 0 0 1293
  168 0 0 20 0 3 0 492156146 142876672 1750
  18446744073709551615 4194304 4235572 140737488349280
  140737488346240 140737327318301 0 0 6 0
  18446744073709551615 0 0 17 0 0 0 0 0 0
139528  7000
12-08-01 13:06:36.892029526
9448 (libtorrent) S 1 9414 9414 0 -1 8192 1971 0 0 0 1387
  176 0 0 20 0 3 0 492156146 142876672 1753
  18446744073709551615 4194304 4235572 140737488349280
  140737488346240 140737327318301 0 0 6 0
  18446744073709551615 0 0 17 0 0 0 0 0 0
139528  7012
```

libtorrent Client Log

Below the raw log as printed by a running libtorrent seeder is shown. The columns are: relative time, percentage completion, bytes uploaded since last row was printed, bytes downloaded since last row was printed.

```
43.997105   94.92   0   0
44.997025   98.52   0   0
45.996922   100.00  356 189
46.996464   100.00  9089857 329291
47.995964   100.00  25345086    792663
48.995850   100.00  5415460 157301
```

lib*swift* Client Log

This log shows a not so very well functioning lib*swift* seeder — the `DONE` line, which is one line in the log, is split into two lines due to space constraints. The `AddData` lines are debugging. The `DONE` line contains the information needed. Its contents are:

`DONE` number of bytes completed `of` total number of bytes to download `(seq` number of bytes completed sequentially from byte 0`)` total number of datagrams sent `dgram` total number of bytes sent `bytes up,` total number of datagrams received `dgram` total number of bytes received `bytes down`

The starting `DONE` is written as `done` if not all data has been received or found, yet.

```
AddData: retransmit of randomized chunk (0,140752)
AddData: retransmit of randomized chunk (0,140753)
DONE 10737418240 of 10737418240 (seq 10737418240) 337857
```

```
  dgram 2642173353 bytes up, 317270 dgram 7090088 bytes down
AddData: retransmit of randomized chunk (0,134023)
AddData: retransmit of randomized chunk (0,141824)
AddData: retransmit of randomized chunk (0,141825)
```

B.2.2 Parsed Logs

The wildly varying or very extensive raw logs are very inconvenient to work with. The parsed logs provide a nicer — and consistent — view of the data.

Client Log

Any client supported by default has a parser that transforms the client specific raw log into the format shown below. The columns are relative time, percentage complete, upload speed in kilobytes per second and download speed in kilobytes per second.

```
15 2.63679504395 257.041992188 28928.0087891
16 2.95097351074 5299.12597656 33416.4335938
17 3.23028564453 13523.0429688 29624.6806641
18 3.52279663086 100.864257812 31006.9257812
19 3.82133483887 22453.2216797 31824.6630859
20 4.10667419434 105.7109375 30349.7119141
```

CPU and Memory Profiling Log

The raw profiling log is parsed into the more usable format shown below. The columns are relative time, CPU usage in percentage, residential memory usage in bytes and virtual memory usage in bytes.

```
6.158401832 0.0 1980 25100
7.180806398 19.5617279745 6296 80400
8.203550699 42.0437444217 6440 80400
9.227300194 63.4920948119 6632 80400
10.24871526 46.9936283472 6780 80400
11.282581429 44.4931862357 6952 80400
```

B.2.3 Processed Data

The logs from the clients, both raw and parsed, are combined or further processed to better present them and give more insight into what happened during the run. Also some metadata is saved by several post-processors that make it easy to identify the clients later on.

Statistics

Taking the data of all leechers and the data of all seeders together, the statistics post-processor gives a summary of the run. This data is very useful when quickly comparing several runs, for example in a plot. The statistics on the leechers in the libtorrent scenario, for example:

```
4 17536 15689 2.21 0 0.0 0.7175 227912 193414
```

The columns are: (1) number of leechers, (2) maximum and (3) average of the peak residential memory usage in bytes of all leechers, (4) average of final cumulative CPU time in seconds used by each leecher, (5) number of leechers that completed their downloads, (6) average time in seconds those leechers needed to complete the download, (7) average of the final percentage of completion of all leechers, (8) maximum and (9) average of the peak virtual memory usage in bytes of all leechers. Similar statistics are provided for the seeders.

Graphs

The parsed client logs and profiling logs can be plotted using gnuplot. The results give a good insight into the progress of a client. Figures B.1 and B.2 show the graphs for one of the lib*swift* leechers. They are not very well readable when printed, but very clear when viewed on a screen.

The X axis in both Figures is the time in seconds, running from 0 to 30. In Figure B.1 the red line is the completion in percentage (scaled 0% to 100%) and the green and blue lines are the upload and download speeds, respectively (scaled 0 to 35000 KB/s). In Figure B.2 the red line is the CPU usage (scaled 0% to 100%) and the green line is the residential memory usage (scaled 0 to 10000 KB).

The graphs are most effective in conveying the information from the logs very quickly: the client did not finish the download and it looks like it simply did not have enough time to download it — downloading about 30000 KB per second is not very bad but the progress does not reach 10%. CPU and memory usage do not look like they are a problem for this leecher.

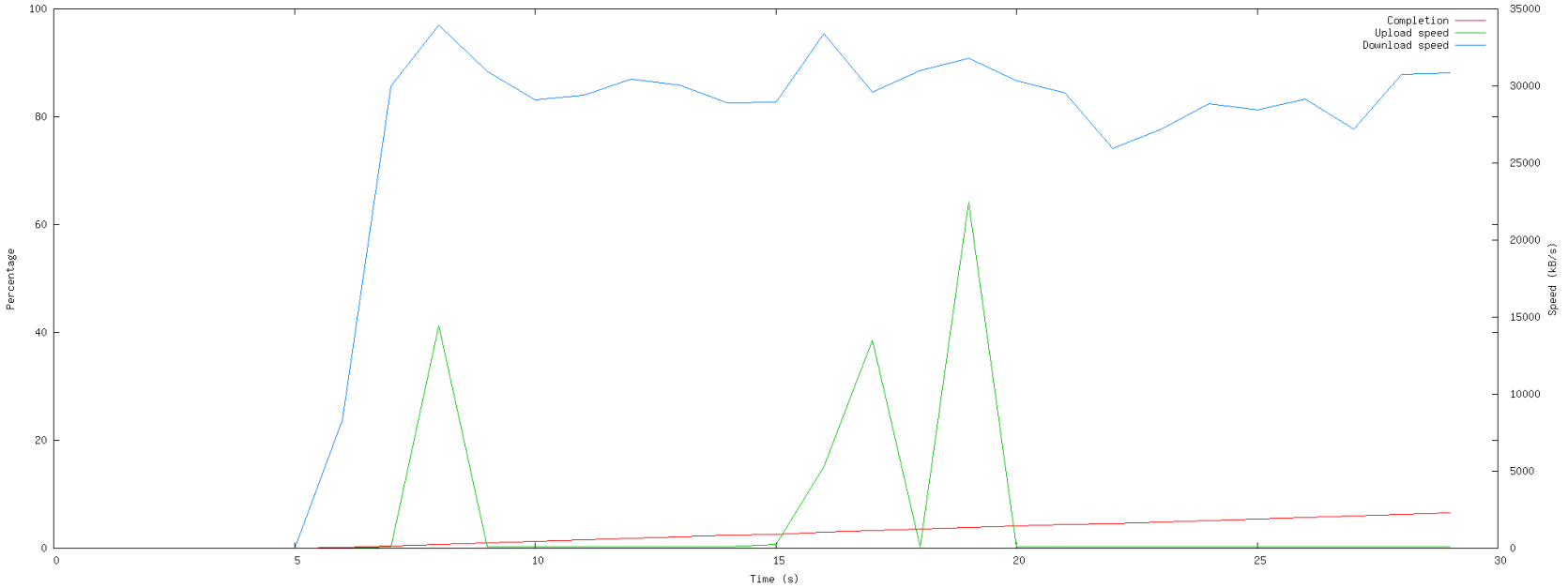


Figure B.1: Client log in graph form — one of the lib*swift* leechers.

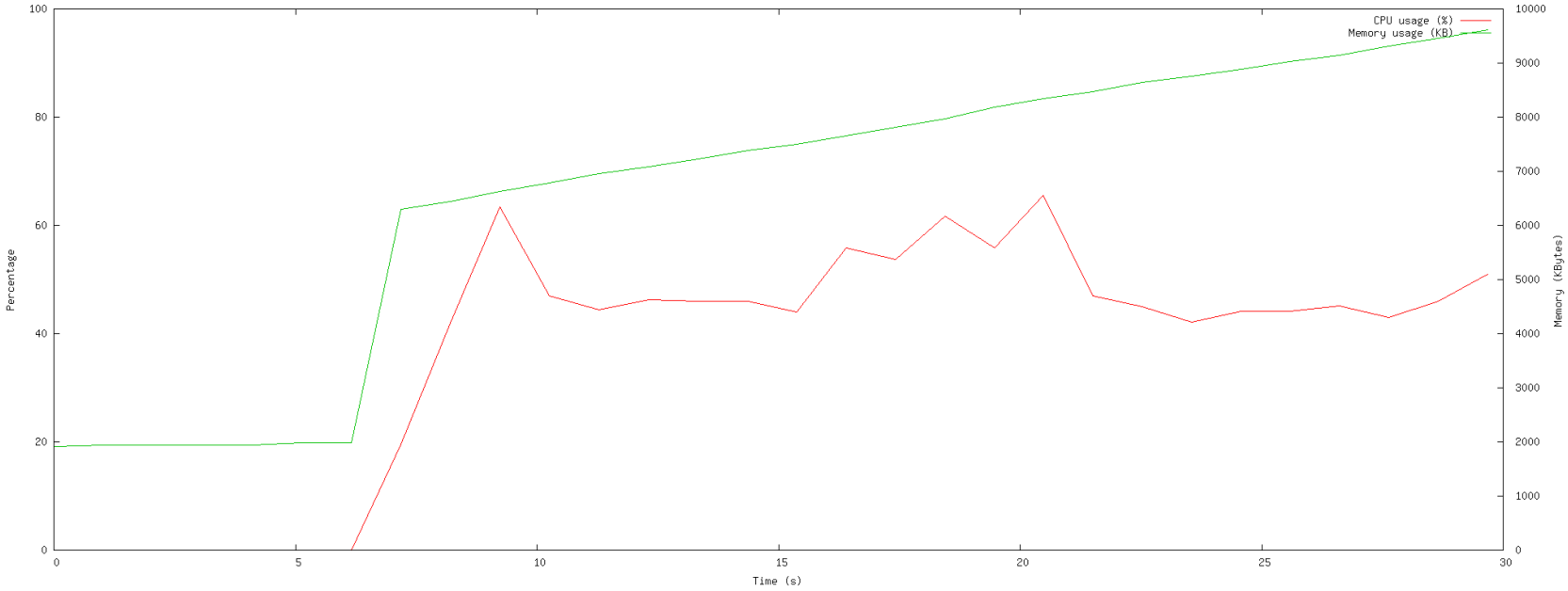


Figure B.2: Profiling log in graph form — one of the lib*swift* leechers.

B.2.4 Views

The last and most complete pieces of output are the views. By default the HTML collection is provided, which was also used for the example experiment.

HTML Collection

The HTML Collection gives an overview of all the clients running in a scenario, presenting their results in a table. A screenshot of the table is shown in Figure B.3.

The overview gives direct insight into all clients. With a bit of experience on reading the graphs one can see in a glance that none of the clients made it, while the seeder has been uploading constantly. An interesting bit of information that is also directly visible is that the CPU usage is not the bottleneck for the seeders nor the leechers.

The graphs on the table are just thumbnails, which link to their fully sized versions for further investigation, just like any file encountered that the viewer does not understand itself is included as a link. Not shown in Figure B.3 are the index into the table and the section with data that does not fit into the table, such as the statistics files which cannot be linked to a specific execution.

Executions

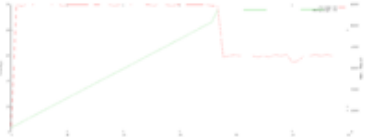
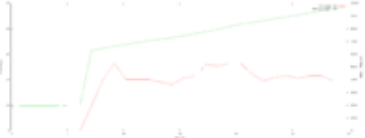
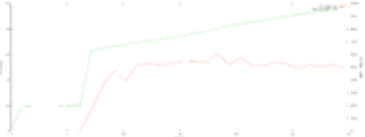
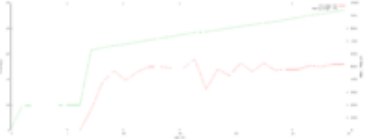
Execution number	Host name	graph_X.png	cpu_graph_X.png	timeout_X	files_X	isSeeder_X
1	seeder			0 s	files_1	YES
2	leechers!1			30.0 s	files_2	NO
3	leechers!1			30.0 s	files_3	NO
4	leechers			30.0 s	files_4	NO
5	leechers			30.0 s	files_5	NO

Figure B.3: HTML collection — the table of all executions in a single scenario run.

Appendix C

Framework Documentation

This appendix contains the main documentation of the evaluation framework. Section C.1 contains the main README file and Section C.2 contains the HOWTO file. The REFERENCE file has been ommitted.

C.1 README

The P2P Testing Framework is a framework for automated running of peer to peer clients. It supports different clients, different types of host and several ways of processing the output. Most importantly it is easily extended to include your needs. Campaign and scenario files decribe how a test is to be run (which client, which hosts, how to connect, how to process data, etc) in a simple declarative language. Python modules, subclasses of the core modules, are loaded to handle most of the specific parts, all glued together by the core of the framework.

This document contains the general documentation, including a version history, the design of the framework and the parameters to the core modules. The HOWTO document gives a more introductory description of how to use the framework and how to extend it, both including examples. For a complete documentation of the framework, please run 'doxygen doxy' inside the Docs directory.

= Version History =
See CHANGES for details.
2.4.0
    - Support for desktop notifications
    - Multiple instances of the same execution
    - Multiple root hashes per file
    - Fixes in the way multiple files are handled
2.3.0
    - Multiplexed connections on host:das4
    - Multiple file support
    - Argument selectors are supported when specifying references to hosts, files or clients
2.2.0
    - Changed connection setups for stability
2.1.0
    - Important extensions to the framework, including much better host support and multiple parser support
    - New workload module type
    - Strongly improved performance
    - client:libtorrent module added
    - file:remote module added
    - parser:cpulog module added, along with client CPU logging
    - processor:savetimeout module added
    - Logs will be salvaged on unsuccesfull runs
2.0.0
    - Complete port of the framework to python
    - Many changes in the internal structure, campaign and scenario files should still work

= Framework design =
The python framework consists of the core script (ControlScripts/run_campaign.py), the core modules (ControlScripts/core/) and the extension modules (ControlScripts/modules). The core script parses the settings on the command line as well as campaign and scenario files in order to create the CampaignRunner and ScenarioRunner objects. This includes loading all necessary core modules and extension modules. The ScenarioRunner object knows how to run a full scenario, which is basically just stepping through all stages and instructing the loaded objects on what to do for each stage.

== Core script ==
The core script initializes everything and glues the parts together. Its ScenarioRunner class is of interest to extension modules, since it contains all the objects in an execution. Those objects are managed using the addObject(...), getObjects(...), getObjectsDict(...) and resolveObjectName(...) methods.

The general flow of the framework is documented below in the Stages section.

== Core modules ==
The core modules are always available. Most of them are the parent classes for the extension modules and will be described below. The other core modules provide global

services.

See the description of the extension modules below or the HOWTO document for more information on using parameters. The REFERENCE contains a full reference of parameters.

=== core.campaign.Campaign ===
A static class with global properties that hold for all campaigns being executed. Elements such as loggers and paths to important local directories are found here, but also a reference to the campaign currently being executed. An important service provided by the Campaign class is the ability to load modules. Using the loadModule(...) and loadCoreModule(...) functions all core and extension modules can be dynamically loaded.

=== core.coreObject.coreObject ===
The parent class of all extension modules and hence the parent class of all extension module parent classes. This class provides a few basic functions, such as naming and cleanup.

=== core.debuglogger.debuglogger ===
An instance of this class is always available through core.campaign.Campaign.debuglogger. It is used for logging communication between the commanding host (the host running the framework) and the hosts doing the actual work. Host modules use this object for logging their communications.

=== core.execution ===
The core of the P2P Testing Framework revolves around executions of clients on hosts operating on files. The execution object contains that combination: host, client and file. It also knows whether the execution is a seeder or a leecher and has a unique number across the campaign and hence across the scenario.

Execution objects are declared along with the other objects and have the following parameters:
- host The name of the host object on which the execution should run. Required
- client The name of the client object which should be executed. Required
- file The name of a file object which is to be transferred. Optional, may be specified muliple times
- parser The name of the parser object which should parse the logs of this execution. Optional, can be specified multiple times. See the description of client extension modules for how parsers are selected.
- seeder Set to anything but '' to mark this execution as a seeding execution. Optional, defaults to ''
- timeout A non-negative floating point number that indicates a number of seconds to wait before actually starting the client after the scenario starts. Optional, defaults to 0
- keepSeeding Set to anything but '' to make sure this seeding execution has to end by itself before the scenario ends; normally seeders are killed when all leechers have finished
- multiply Specify a positive integer number of copies to be created of this execution. Optional, defaults to 1.

The host and file parameters of an execution are the most important places for use of argument selectors.

=== core.logger.logger ===
The generic scenario logger object. An instance is always available through core.campaign.Campaign.logger. It is used for logging about anything that needs logging. Several convenience functions are provided to handle exceptions and tracebacks.

=== core.meta.meta ===
Contains a few static functions that allow creation of meta data, such as Merkle root hashes or torrent files.

=== core.parsing ===
This module provides several functions that make it easier to parse arguments in the

scenario files. Often used are isPositiveInt(...) and isPositiveFloat(...).

== Extension modules ==
The extension modules provide all the actual functionality. Described here are the parent classes to the extension modules and their parameters. For the parameters of the specific extension modules see the documentation of their classes.

To get a better idea of how to use the parameters in the configuration of the framework, please see the HOWTO document.

=== host ===
Host modules provide the connection to a host and the services to run commands on that host and to send or retrieve files. Each host modules keeps track of, usually, a number of connections to the host which can be used to send commands and send or retrieve files.

Host objects also include information about traffic control that needs to be put on the host. Such traffic control is defined using parameters to the host object, but is implemented by the tc extension modules.

Parameters:
- name The name of the host object. This name is used to refer to the host object in throughout the scenario. Usually required (particular extension modules sometimes provide a default)
- remoteDirectory The path to a directory on the remote host which can be used to store temporary files in during the scenario. Optional, a temporary directory will be created by default (in /tmp usually)

By default the following host modules are provided:
- host:local Uses the local host, mainly for testing. If you wish to use the local host for serious scenarios consider using host:ssh to 127.0.0.1.
- host:ssh Uses a host that can be approached via SSH. This is the preferred way of contacting hosts.
- host:das4 Special handler for those with access to the DAS4 system.

==== tc ====
TC modules provide traffic control to a host. They probably have special requirements on the host and your access to it, so be sure to always check that. Traffic control can be used to simulate a different networking environment, e.g. with lower speeds and lossy connections.

TC modules don't have parameters of themselves, but use the parameters set on the host object they operate on. The following parameters are therefor set on the host object, but used by the TC module.

Parameters:
- tc The name of the TC module to load, without any prefixes. E.g. tc=netem to load the tc:netem module on the host. Optional, empty by default which disables TC
- tcInterface The name of the interface on which traffic control is to be applied. This should be an existing networking interface on the remote host. Optional, defaults to eth0
- tcMaxDownSpeed Maximum download speed to allow. To be specified in bits per second, possibly postfixed by kbit or mbit. E.g. tcMaxDownSpeed=10mbit for 10 mbit speeds. Optional, defaults to 0 meaning no restrictions
- tcMaxDownBurst Maximum burst in the download speed. Not allowed if tcMaxDownSpeed is not set. To be specified in bits per second. Optional, defaults to equal to tcMaxDownSpeed
- tcMaxUpSpeed Like tcMaxDownSpeed, but for upload speed.
- tcMaxUpBurst Like tcMaxDownBurst, but for upload speed.
- tcLossChance Chance to drop a packet. A floating point number between 0.0 and 100.0 inclusive, specifying the chance as a percentage. Optional, defaults to 0.0
- tcDuplicationChance Chance that a packet will be duplicated. A floating point number

between 0.0 and 100.0 inclusive, specifying the chance as a percentage. Optional, defaults to 0.0
- tcCorruptionChance Chance that a packet will be corrupted. A floating point number between 0.0 and 100.0 inclusive, specifying the chance as a percentage. Optional, defaults to 0.0
- tcDelay The delay to introduce on each packet in ms, given as a positive integer. Optional, defaults to 0
- tcJitter The maximum deviation on the introduced delay, as set by tcDelay, in ms. Optional, defaults to 0

By default the following tc modules are provided:
- tc:netem Uses the netem kernel module with the tc utility

=== file ===
File modules describe data to be transferred. Usually this will consist of one or more files. Also includes metadata for the file, such as Merkle root hashes or torrent files.

Parameters:
- name The name of the file object. This name is used to refer to the file object in throughout the scenario. Required.
- rootHash[xx] A Merkle root hash of the file. Consists of 40 hexadecimal digits. Replace xx in the parameter name with the chunksize in kbytes upon which the root hash is based (that is the size of the data from which each leaf hash is calculated). Postfix this by L if you wish to have legacy root hashes, i.e. where the root is always the 63rd level in the tree. Examples of parameter names: rootHash[1]=... rootHash[8L]=... Optional, may be specified multiple times but not for the same chunksize.
- metaFile A file with metadata, such as a torrent file. This file will be made available to all client executions, both seeders and leechers. Should be a path to a file on the command machine. Optional.

By default the following file modules are provided:
- file:local Specifies a local file or directory to use as data.
- file:remote Specifies a remote file or directory to use as data.
- file:fakedata Creates fake data on the remote host that is always the same, non-trivial, of configurable size and real.

The file:none module exists, but is deprecated: just don't pass any file parameters to the execution.

=== client ===
Client modules run the client application that are to be tested. To run this they use the services and information from all parts of the execution. The client module is responsible for everything regarding the client, from downloading and compiling via running to killing and retrieving logs. This burden is partly offloaded to the builder and source extension modules, and mostly present in the parent class.

Client objects also include information about where they are located and how they are to be built. Such information is defined using parameters to the client object, but is used by the source and builder extension modules.

On parser selection: each execution runs one or more parsers after all the raw logs have been retrieved. The first set of parsers in the following list is used:
1) All parsers specified in the execution object
2) All parsers specified in the client object
3) The declared parser object with the same name as the client module subtype (i.e. a parser object named 'swift' for [client:swift], no matter the type of the parser object or the name of the client object)
4) A new parser of the same subtype as the client (i.e. a new [parser:swift] for a [client:swift])
For steps 1 and 2 the actual parser objects are looked up by first looking at the

declared parser objects. If the name is among the declared parser object's names, that one is used. However, if no parser object is declared with that name, but the name is the same as a parser module subtype that subtype will be loaded. So to run a parser:cpulog with no parameters on a given client just specifying parser=cpulog with the client is enough.

Parameters:
- name The name of the client object. This name is used to refer to the client object in throughout the scenario. Optional, defaults to the name of the extension module used
- extraParameters Extra parameters to be appended on the command line to the client. Client specific. Optional, defaults to ''
- parser The name of the parser object to be used to parse logs from this client. Optional, defaults to a new parser with the same name as the name of the extension module used; may be specified multiple times
- profile Set this to anything but "" to include external profiling code that will inspect CPU and memory usage every second, which will be captured in the raw cpu.log. Optional, defaults to ''
- logStart Set this to anything but "" to log the starting time of the client, which will be captured in the raw starttime.log. Note that this uses the local clock of the remote host. Optional, defaults to ''

By default the following client modules are provided:
- client:http Uses lighttpd and aria2 to provided HTTP(S) downloads
- client:opentracker Allows running the opentracker BitTorrent tracker software, useful in combination with other BitTorrent clients
- client:utorrent Uses the uTorrent binary clients with the webui
- client:swift Uses the libswift command line client
- client:libtorrent Uses the libtorrent mini command line client distributed via https://github.com/schaap/p2p-clients

==== source ====
source modules instruct the framework how to retrieve the source or binaries of the client.

source modules do not have parameters of themselves, but use the parameters set on the client object they operate on. The following parameters are therefor set on the client object, but used by the source module.

Parameters:
- source The name of the source module to load, e.g. source=local to use source:local. Optional, defaults to source:directory
- remoteClient Set to anything but '' to signal that the sources are to be loaded, or found, on the remote host instead of the commanding host. Optional, defaults to ''
- location The location of the sources. The contents of this parameter depends on the source module used. Required.

By default the following source modules are provided:
- source:directory Assumes the sources or binaries to be present in the directory pointed to by location; if remoteClient is set this is a directory on the remote host, otherwise on the commanding host
- source:local Assumes the sources or binaries to be present in the directory on the commanding host pointed to by location; if remoteClient is set this means the local sources are first uploaded before the builder starts
- source:git The location is a valid git repository that can be cloned

==== builder ====
builder modules know how to compile the sources of a client.

builder modules do not have parameters of themselves, but use the parameters set on the client object they operate on. The following parameters are therefor set on the client object, but used by the builder module.

Parameters:
- builder The name of the builder module to load, e.g. builder=make to use builder:make. Optional, default to builder:none

By default the following builder modules are provided:
- builder:none The client has already been built. Compilation is skipped.
- builder:make Uses (GNU) make to build the client
- builder:scons Calls the scons building program to build the client

=== workload ===
workload generator modules can change the executions such that the arrival times of the clients simulate specific workloads.

Parameters:
- apply Specifies the name of a client object to apply the workload to; this means every execution of that client (but see applyToSeeders) will be changed to be included in the generated workload. Optional and may be specified multiple times. If apply is enver specified then all clients that are part of a (non-seeding) execution are added as soon as all objects have been loaded.
- applyToSeeders By default a workload generator will only change non-seeding executions. Set this to 'yes' to have it change seeding executions as well. Optional
- offset Starting time of the simulated workload from the start of the scenario in seconds. Optional, floating point

Please note that workloads do not accept parameters with argument selectors.

By default the following workload generator modules are provided:
- workload:linear Creates a division of the clients to arrive at a linear rate
- workload:poisson Creates a division of the clients to arrive like a poisson process

=== parser ===
parser modules know how to parse the output of a client. They are specified per execution or per client. A parser module takes as input the raw logs as retrieved from the remote host and outputs parsed logs.

Parameters:
- name The name of the parser object. This name will be used to refer to the parser object throughout the scenario. Optional, defaults to the name of the extension module used

By default the following parser modules are provided:
- parser:none A dummy implementation parsing nothing
- parser:http A copy of parser:none for easier use with client:http
- parser:aria2 The parser for logs from aria2 as retrieved by client:http
- parser:lighttpd The parser for logs from lighttpd as retrieved by client:http
- parser:opentracker A copy of parser:none for easier use of client:opentracker
- parser:utorrent The parser for logs from utorrent as retrieved by client:utorrent
- parser:swift The parser for logs from swift as retrieved by client:swift
- parser:cpulog A parser for CPU logs as generated by having the profile parameter set on a client
- parser:libtorrent The parser for logs from libtorrent as retrieved by client:libtorrent

All parsers that are provided by default, except for parser:cpulog, parser:none and its clones, provide the same output format. It is not required to use this format: any format is fine as long as it's documented.

=== processor ===
processor modules can process raw and/or parsed logs into nicer datasets or visualizations or whatever.

processor modules do not have generic parameters: just declaring their object to be

present is usually enough. Do look at the particular extension module you use for parameters it might need, though.

By default the following processor modules are provided:
- processor:savehostname Creates a simple text file for each execution with the name of the host object the execution ran on.
- processor:saveisseeder Creates a simple text file for each execution with "YES" in it if the execution was a seeder; "NO" is in it otherwise.
- processor:savetimeout Creates a simple text file for each execution with the timeout in seconds (float) before the client was launched.
- processor:gnuplot Runs a given gnuplot script for each parsed log in an attempt to create nice graphs.

For the processor:gnuplot two scripts are provided as well:
- TestSpecs/processors/simple_log_gnuplot Generates a graph for the output of the provided parsers for client logs
- TestSpecs/processors/simple_cpu_gnuplot Generates a graph for the output of parser:cpulog

=== viewer ===
viewer modules take all the data together and provide a nice view of the data.

viewer modules do not have generic parameters: just declaring their object to be present is usually enough. Do look at the particular extension module you use for parameters it might need, though.

By default the following viewer modules are provided:
- viewer:htmlcollection Creates an HTML page that describes the whole scenario.

= Stages =
This section described the workflow inside the testing framework for each scenario. The code below is pseudocode that matches what the ScenarioRunner does.

```
0) Read the combined scenario file
    a) Create the new object, call parseSetting(...) on it for each parameter and
    finally call checkSettings() on it
    b) With each host object: call .doPreprocessing() on it
    c) With each file object: call .doPreprocessing() on it
    d) With each object: call .resolveNames() on it
    e) With each execution:
        I) Add the client object in the execution to the execution's host's client set
        II) Add the file objects in the execution to the execution's host's files and,
            if needed, seedingFiles sets
1) Collect all hosts that are part of an execution in a set executionHosts
2) Call host.prepare() on each host in executionHosts, this prepares that hosts and sets
up connections to them
3) Collect all hosts that are part of an execution in a set executionHosts
4) With each workload generator
    a) Call workload.applyWorkload(), which changes the executions
5) On all client object, call client.prepare(), this will prepare the client binaries,
   including up/downloading source and compilations
6) With each host in executionHosts
    a) If the host requests TC
        I) Analyse the hosts and clients to see which TC (inbound/outbound) (port
        restricted/fully restricted)
           is needed
        II) Load the tc module for this host
        III) Check to see that the TC option needed can actually be loaded (calls
        tc.check(host) )
            0) If not, try and fall back to more restrictive TC
            0) If including fallbacks nothing is possible, fail the scenario
        IV) Save the way TC is to be done in the host
    b) Call client.prepareHost(host) for each client in host.clients,
```

```
       which contains all clients that will run on the host
7) With each host in executionHosts
    a) Call file.sendToHost(host) for each file in host.files,
       which contains all the files that will be seeded from or leeched to the host
    b) Call file.sendToSeedingHost(host) for each file in host.seedingFiles,
       which contains all the files that will be seeded from the host
8) With each execution
    a) Call execution.client.prepareExecution(execution)
9) With each host in executionHosts
    a) If the host reqests TC
        I) Install the TC (calls tc.install(host) )
10) With each execution
    a) Prepare an execution specific connection to the host of the execution
11) With each execution, in parallel (with each other and with step 12)
    a) Wait the specified timeout
    b) Call execution.client.start(execution) to start the client on the host
12) While the timelimit has not been reached
    a) Sleep at most 5 seconds
    b) With each execution that is not a side service
        I) Call execution.client.isRunning(execution) to see if the client is still
           running
            0) If so, stop checking the other executions and continue with 12
13) With each execution, in parallel
    a) Call execution.client.isRunning(execution) to see if the client is still running
        I) If so, call execution.client.kill(execution) to have the client killed
14) With each host in executionHost
    a) If the host requests TC
        I) Remote the TC (call tc.remove(host) )
15) With each execution for which the client is not a side service, in parallel
    a) Call execution.client.retrieveLogs(execution) to retrieve the client logs for the
       execution
    b) Call execution.runParsers(...)
        I) If parsers were set for the execution
            0) Call execution.parser.parseLogs(...)
        II) Otherwise
            0) Retrieve a number of parsers plist by calling
               execution.client.loadDefaultParsers(execution)
            0) Call p.parseLogs(...) for each p in plist
16) With each host
    a) Create a new connection to the host to use for cleanup
17) With each execution
    a) Call execution.client.hasStarted(execution) and
       execution.client.isRunning(execution)
       to find out if the client is running
        I) If so, call execution.client.kill(execution) to kill the client
18) With each file
    a) Call file.cleanup()
19) With each host
    a) With each client in host.clients, which contains all the clients that will/have
       run on the host
        I) Call client.cleanupHost( host )
20) With each client
    a) Call client.cleanup()
21) With each host
    a) If the host requests TC
        I) Try and remove TC (calls tc.remote(host) )
    b) Call host.cleanup(), which also cleans up the connections, including the cleanup
       connection
22) With each processor
    a) Call processor.processLogs(...)
23) With each viewer
    a) Call viewer.createView(...)
```

Steps 16 through 21 are the cleanup, which is at each call guarded against errors and will run always. Note that it can also run at any moment in time, e.g. due to an Exception being raised. So from any step before 16 one can always jump straight into 16. In case of such a jump the scenario stops after step 21. A particular jump is after step 6a: if the run is just a testrun step 6b will not be executed and once step 6 is done the jump to cleanup will be taken.

Step 15 exists in two different versions: at any point during steps 1 through 15 an error might occur; in that case a specially guarded version of 15 is ran before cleanup is started. During normal execution, step 15 is not guarded.

Please note the peculiar collections of executionHost: once before and once after preparing the hosts. The framework explicitly allows for the collection of objects to be changed by the preparation of the hosts, as long as any host that is added is guaranteed to have been prepared if it's also part of an execution. This is for example exploited by host:das4 which creates a host object for every node in its preparation phase.

C.2 HOWTO

This HOWTO will introduce you into the usage of the P2P testing framework. Topics covered are building tests, running tests, reviewing results and extending the framework. Throughout this HOWTO commands and files will be assumed to be run in the root of the P2P test framework, which is the directory that contains the ControlScripts directory. This is the working directory that is assumed throughout all documentation of the framework.

During the first parts of this HOWTO an example will be cosntructed that will instruct the framework to connect to some hosts using SSH, run the swift client to transfer a file, to plot some statistics on that and finally to present the results in HTML.

The last part of this HOWTO will demonstrate how to develop a new module for the framework by example of the development of the file:fakedata module.

=== BUILDING TESTS ===
To run a test you first have to build the scenario and campaign files. In the scenario files you define which hosts will run which clients to send which files. As an example we will build a test that sends a single file from host1 to host2, both accessable over ssh, using the swift client. We will use one scenario file to define the file object, one file to define the hosts and one to define the clients and put it all together. The separation into multiple files is just an example: you could just as well use one file or a different separation.

One thing we will not explicitly do here, but what you should do when writing your own scenarios (and what I did when writing this), is referring to the documentation. The base entry point is the README file, which documents the parameters to all the generic objects. Apart from that you should always open the file of every module you use: module specific documentation is placed at the beginning of the module file. Finding these files is simple: as an example module file:local is located in ControlScripts/modules/file/local . This is actually the very reason the module is called file:local. This is, by the way, also the way that is used throughout the framework to refer to specific files.

Once you got the hang of the basic syntax the REFERENCE becomes very useful. I found going through it every time I wrote a test was actually the fastest way not to miss anything. The REFERENCE, of course, only contains the documentation for the modules delivered with the framework.

= file: TestSpecs/files/my_file =

```
[file:local]
name=myfile
path=/home/me/someNiceFileToTransfer
rootHash=01234567890123456789012345678901234567890
```

This creates a single file object named 'myfile'. It points to the local file given by path=. Since we'll be transferring this file using swift it is useful to also give the rootHash. Of course the root hash here is bogus ;).

= file: TestSpecs/hosts/my_hosts =

```
[host:ssh]
name=my_seeder
hostname=myseederhost.foo.bar

[host:ssh]
name=my_leecher
hostname=myleecherhost.foo.bar
user=my_alter_ego
```

This creates two host objects named 'my_seeder' and 'my_leecher'. They instruct the framework to use SSH to connect to the hosts under the given hostnames (this can also be an IP). In case of the leecher a different username than the logged in user is to be

used.

= file: TestSpecs/scenarios/my_scenario =

```
    [client:swift]
    name=seedingswift
    location=git://github.com/gritzko/swift.git
    source=git
    builder=make
    remoteClient=yes
    listenPort=15000
    wait=300

    [client:swift]
    name=leechingswift
    location=/home/me/prebuilt_swift_dir
    tracker=myseederhost.foo.bar:15000

    [execution]
    host=my_seeder
    file=myfile
    client=seedingswift
    seeder=yes

    [execution]
    host=my_leecher
    file=myfile
    client=leechingswift

    [processor:gnuplot]
    script=TestSpecs/processors/simple_log_gnuplot
    [processor:savehostname]

    [viewer:htmlcollection]
```

This first creates two client objects named 'seedingswift' and 'leechingswift'. The seedingswift client is instructed to have its source pulled using git (source=git) from the given repository (location=). It is to be built remotely (remoteClient=yes) using make (builder=make). Two swift specific parameters for the seedingswift client are given to instruct the client to listen on port 15000 and to wait for 300 seconds before terminating. The leechingswift client uses a locally prebuilt binary swift located in /home/me/prebuilt_swift_dir/. This client will be uploaded to the leeching host and executed. It is instructed to use myseederhost.foo.bar:15000 as its tracker.

Then the file proceeds with declaring two executions. Executions are the combination of host, client and file. The first execution instructs the framework to run the seedingswift client on the my_seeder host to transfer myfile and it tells the framework that that host will be a seeder (seeder=yes). The latter is important to do correct: only seeding executions will have the actual files needed to seed uploaded, non-seeding executions will only upload the meta data. The second execution runs the leechingswift client on the my_leecher host to transfer myfile again.

The last lines instruct the framework to run two postprocessors: gnuplot and savehostname. The former runs gnuplot on the gathered data with a supplied script, the simple_log_gnuplot script in this case, and the latter just saves the hostname as given above in single files. The output of both of these will be used by the htmlcollection viewer which will be run in the end. That viewer will generate an HTML overview of what has been going on.

= file: TestSpecs/my_campaign =

```
    [scenario]
    name=scenario1
```

```
    file=TestSpecs/files/my_file
    file=TestSpecs/hosts/my_hosts
    file=TestSpecs/scenarios/my_scenario
    timelimit=60

    [scenario]
    name=scenario2
    file=TestSpecs/hosts/my_hosts
    file=TestSpecs/files/my_file
    file=TestSpecs/scenarios/my_scenario
```

This is the campaign file. The campaign file is the complete description of the campaign, using indirections into the scenario files. It can't contain other objects than scenario objects and just instructs the framework which files to concatenate in order to create a full scenario file. It also gives the scenarios a name and optionally a time limit (in seconds). Note that the order of the file parameters is important: the files are simply concatenated in the order they are given and if an object is declared before it is used, the framework will simply complain. For example, if we would specify the my_file scenario file after the my_scenario scenario file, the framework will complain after parsing the first execution: it can't find file object myfile.

=== RUNNING TESTS ===
Now that your very interesting and elaborate test suite has been built, it is time to run it. The most easy way is:

```
    ./ControlScripts/run_campaign.py TestSpecs/my_campaign
```

This will run the scenarios in your campaign file. You can instruct the framework to check your campaign instead, without actually running everything:

```
    ./ControlScripts/run_campaign.py --check TestSpecs/my_campaign
```

Note that the syntax and sanity checks will be run during the actual run as well: a check run simply stops before any uploading and executing is done. When developing campaigns it is advisable to do a check run first, for example to establish whether your hosts are reachable without user interaction.

Several more options are available, mainly for debugging. Just run

```
    ./ControlScripts/run_campaign.py
```

without any other options to get a list of them.

And that's all there is to it. Just run it.

= Access to hosts =
One important note on access to hosts: this needs to be done without user interaction! This goes for everything in the framework, but accessing hosts is the most important example. Usually you will access some hosts over SSH. Make sure you can access those hosts without having to type anything! Create a key for your own identity and use ssh-agent to make sure you don't need to enter the passwords for your private keys. (You do have passwords on your private keys, right?)

A typical session for me goes like this:

```
    ssh-agent bash
    ssh-add
        [Type password to private key]
    ./ControlScript/run_campaign.py TestSpecs/my_campaign
    exit
```

The host:ssh module will check whether your hosts are reachable, but you can do so by hand yourself:

```
    ssh yourhost "date"
```
This should connect to the host, print the date, and fall back to your local prompt. If anything happens in between, such as extra output or user interaction, the framework will not work. Of course, you should add those parameters you also give to the framework, such as a different username or extra parameters.

=== REVIEWING RESULTS ===
After your tests have run, or failed, you should always review some results. The results can by default be found in the Results/ directory. Say you have just ran the above campaign my_campaign, and it was 17:00:00 on the 24th of November 2011. The results will then be in Results/my_campaign-2011.11.24-17.00.00/. In this directory you will first find err.log. Always review this: it is extra output from the scenarios. This file is especially important when something failed (the output of the framework will direct you here, as well).

Apart from the err.log file there is the scenarios directory which holds one directory for each scenario. Inside each scenario's directory are all the logs and results of that scenario. Firstly there is the scenarioFile file, which is the concatenation of scenario files used to initialize the scenario. This is useful for debugging and also automatically documents the setup of your tests. Note that when line numbers are mentioned in error lines, they always refer to this file.

The executions directory contains one directory for each execution, numbered exec_0, exec_1, etc. Inside these you will find the logs and parsedLogs directories, which contains the raw logs from the clients and the interpreted logs after a parser has been run on them (for using other parsers than the default ones: consult the full documentation). You can of course use these logs to do your own extended analyses.

Next to the executions directory are the processed and views directories, which respectively contain post-processed data, such as graphs or formatted logs, and views, such as the HTML overview.

When everything from your my_campaign campaign went well, you should usually first check the actual output. The htmlcollection view was defined, which takes together all processed data and puts it into an HTML page. To view this, you could run:
```
    firefox
    Results/my_campaign-2011.11.24-17.00.00/scenarios/scenario1/views/collection.html
```

=== EXTENDING THE FRAMEWORK ===
The framework is built with extensions in mind. There are several categories of extensions you can make:
- host modules
- file modules
- client modules
- parser modules
- processor modules
- viewer modules
- tc modules
- builder modules
- source modules
- workload modules

Those are quite some extension points. This HOWTO doesn't even use all of them and no effort will be made to discuss each extension in detail. For a particular extension's details, please refer to the README and other documentation.

The general process of creating a new module is this:
1) Read up on the API the module should implement;
2) Copy the skeleton file to your own module;
3) Read your new module (which is just the skeleton) and read up on any mentioned APIs you can use, as well as the global API;
4) Write your implementation in the skeleton that is currently your module, be sure to document and check all places where it says TODO;
5) Thoroughly test your implementation and adjust as needed.

That looks a lot like generic software development and in fact it is. But due to the use of the skeleton a lot of the hard labor is taken out of it. Every module is also based on a generic parent class that does most of the heavy lifting and administration. This leaves just the particular details for your module to be filled in. For example, if you would like to add a new client (probably the most commonly made extension) you need to define the layout on disk of your client (so it can be moved/uploaded), tell the framework how to run it and instruct how to retrieve the logs. And that's it. For a simple client with only one binary and logging on stderr this would take a total of 5 lines of code. Most effort would probably be in reading through all the comments in the skeleton implementation.

As an example of this process the development of file:fakedata is documented below.

= 1) Read up on the API the module should implement =
As a first step, let's run the doxygen tool to get our documentation.

```
cd Docs/
doxygen doxy
```

With the documentation in place, we'll take a look at the API of the file object.

```
firefox html/index.html
```

click on Classes
click on core::file:file

Read through the class' methods to get an idea of what functionality the file:fakedata class will have to offer.

Based on everything we can read here one could build an implementation. But let's make life easy on ourselves.

= 2) Copy the skeleton file to your own module =

```
cp ControlScripts/modules/file/_skeleton_.py ControlScripts/modules/file/fakedata.py
```

Check.

= 3) Read your new module and read up on any mentioned APIs and the global API =
There's a number of TODOs in the skeleton file and a lot of comments. Most comments give examples on possible implementations, as well. In fact the examples show you the complete implementation of an empty file.

With each method defined there are descriptions of how to implement that method. It's a good idea to also have a look at the types of the arguments passed to the methods we will be implementing. Going over all methods one could note we really only deal with host objects in this module. So let's look up the host class in our firefox: *click on Classes* and *click on core::host::host*. Read through the methods to know what services a host can provide. Interesting bits are:

- getTestDir
- sendCommand
- sendFile
- sendFiles

Also important is the static Campaign class. *Click on classes* and *click on core::campaign::Campaign* to read the documentation on that. Since we need a utility provided with the framework for file:fakedata, the testEnvDir property might be useful.

Now that we've read up on our supporting code and have an idea of what to do, we can get to implementing the module.

= 4) Write the implementation of your module =
When implementing a module there are two important points to take into account, next to

building your complete implementation:
    - Go over all places where it says 'TODO' (search for it)
    - Document what you're doing and how your module works
The TODOs are there to make sure you touch all points you should, either because they need some administrative touches, or you should carefully consider whether to implement and/or extend the function. The documentation is obviously needed to make sure others can use your module. The most important documentation comes at the top of your class: there it should say what the module does, how to use it, and other important things about it.

A full contextual diff will be placed at the end of this file, so the exact implementation won't be discussed here, just a number of specific ways of getting there. As such, the documentation and administrative changes can be reviewed in the diff.

sendToSeedingHost is where the really interesting stuff happens. Note that the binary parameter tells us to use an already existing binary on the remote host (and which binary), so we should check whether it is set before trying to compile the binary remotely. Uploading and compiling the files involves some interaction with the rest of the framework and for that reason its development is detailed below. The following thoughts are relevant:
    - The fakedata utility is in Utils/fakedata/ and consists of all .cpp and .h files there;
    - On the remote host the source should have its own (temporary) directory;
    - There are many compilers out there and we can't easily take all of them into account;
    - Errors might occur and we should handle those.

The first thought touches on finding those files locally. In the Campaign object the testEnvDir property tells us where the testing environment is located. From there we can get to the fakedata utility: "{0}/Utils/fakedata/".format(Campaign.testEnvDir) should be the directory holding the files.

The second thought has to do with making sure we don't overwrite other files and at the same time don't pollute the remote host with our stuff. A remote temporary directory would be ideal for that, as long as it is removed again when we stop. We'll create a remote temporary directory using the mktemp command. As for the base path we can give to mktemp: is there a good way to place this temporary directory? In fact you'll find there is: whenever a host is initialized a temporary directory is made available on it where temporary files for the testing framework can be placed. host.getTestDir() host.getPersistentTestDir() will tell you about this. The question is which to use. Will these files be needed after cleanup? Certainly not, only log files and similar output is needed after cleanup and this utility can be thrown away again. So host.getTestDir() is the right function to call.

Many compilers are available and we could write some very complex code trying to find out which compiler is supported, etc, etc. We might as well go for autoconf for that. Or we could just choose one. g++ is often available and for if it isn't: just let the user specify a manually compiled binary. This is a tradeoff between usability and complexity. In this case the complexity will grow far too much if we'd try and support many compilers. Those hosts lacking the g++ compiler are covered by the binary parameter.

The last thought, the occurence of errors, becomes more important with the choice for just one compiler. It's also a though that fall into two parts: finding problems and acting on them. Any problems that could occur in this case are during calls to remote commands. Luckily the host.sendCommand() function, which we'll use for running commands remotely, will return the output of the command. As such it is possible to just include some simple bash to always output, say, "OK" when everything went fine. Catch the output in a variable, check that the last line says "OK", and problems can be found. What to do, then, when a problem occurs? Raising an exception usually does it, they get logged and kill the current scenario.

Having thought about these things we can write most of the code for compiling the utility remotely. For running it one important question still arises: where to store the

file? A convention that is used throughout the framework is that each module claims its own directory in a temporary directory's substructure: module_type/module_subtype/ . Or in this case: self.getFileDir(host). Wait, where did that come from? It's documented in the file parent object and will just return the complete path to the directory specific for your file module on the passed host. With this information it becomes a matter of filling in the blanks and just writing the code. See the diff for the results.

= 5) Test and adjust =
Testing your module should be done using your usual software testing techniques: cover all code, test corner cases, special parameters, etc. Make sure it works. It is usually easiest to write a small campaign in which to test a module you develop. I have a few around, for example, in which I can just plug a new client and have that client run on a few machines to send some files around. It is this one I also changed to use file:fakedata in order to test that.

When first developed the file:fakedata was not implemented correctly. Some typos, some small mistakes, the usual. The reason you test. More interestingly it turned out that no files were transmitted at all when using file:fakedata, even though the files were created as intended. An error in the design was found: torrents (which were used for its testing) name specific files, but file:fakedata chooses its own name. This led to the file being generated, but never being recognized by the seeder. The filename parameter was introduced because of this.

= Diff =
Below is the full contextual diff from file:_skeleton_ to file:fakedata. Note that the current version of file:fakedata is quite different, since it supports multiple files.

Note that this is actually the ported python version instead of the original bash version (for which this, now adapted, HOWTO was originally written).

```
*** ControlScripts/modules/file/_skeleton_.py   2012-03-09 13:27:03.546043891 +0100
--- ControlScripts/modules/file/fakedata.py     2012-03-09 13:27:03.572055429 +0100
***************
*** 1,59 ****
! # These imports are needed to access the parsing functions (which you're likely to use in parameter parsing),
! # the Campaign data object and the file parent class.
! from core.parsing import *
  from core.campaign import Campaign
  import core.file

! # NOTE: The last import above (import core.file) is different from usual. This is done to prevent trouble with python's
! # builtin file type. The import
! #   from core.file import file
! # works perfectly, but hides the normal file type. The tradeoff is between a bit more typing (core.file.file instead of file)
! # and possible errors with regard to file and file (??).
!
! # You can define anything you like in the scope of your own module: the only thing that will be imported from it
! # is the actual object you're creating, which, incidentally, must be named equal to the module it is in. For example:
! # suppose you copy this file to modules/file/empty.py then the name of your class would be empty.

  def parseError( msg ):
      """
      A simple helper function to make parsing a lot of parameters a bit nicer.
      """
      raise Exception( "Parse error for file object on line {0}: {1}".format(
      Campaign.currentLineNumber, msg ) )
```

```
! # TODO: Change the name of the class. See the remark above about the names of the
module and the class. Example:
! #
! #   class empty(core.file.file):
! class _skeleton_(core.file.file):
      """
!     A skeleton implementation of a file subclass.

!     Please use this file as a basis for your subclasses but DO NOT actually
instantiate it.
!     It will fail.
!
!     Look at the TODO in this file to know where you come in.
      """
!     # TODO: Update the description above. Example:
!     #
!     #   """
!     #   An empty file object.
!     #
!     #   To be used just to create an empty file.
!     #
!     #   Extra parameters:
!     #   - filename  The name of the file to be created.
!     #   """

      def __init__(self, scenario):
          """
          Initialization of a generic file object.

          @param  scenario        The ScenarioRunner object this client object is part
          of.
          """
          core.file.file.__init__(self, scenario)
-         # TODO: Your initialization, if any (not likely). Oh, and remove the next
line.
-         raise Exception( "DO NOT instantiate the skeleton implementation" )

      def parseSetting(self, key, value):
          """
          Parse a single setting for this object.

--- 1,44 ----
! from core.parsing import isPositiveInt
  from core.campaign import Campaign
  import core.file

! import os

  def parseError( msg ):
      """
      A simple helper function to make parsing a lot of parameters a bit nicer.
      """
      raise Exception( "Parse error for file object on line {0}: {1}".format(
      Campaign.currentLineNumber, msg ) )

! # The list of files needed for the fakedata utility
! fakedataGeneratorFiles = ['compat.h', 'fakedata.h', 'fakedata.cpp', 'genfakedata.cpp']
!
! class fakedata(core.file.file):
      """
!     A file implementation for generated, fake data.
!
!     This module uses Utils/fakedata to generate the data for the files.
```

```
!     Extra parameters:
!     - size      A positive integer, divisible by 4096, that denotes the size of the
generated file in bytes. Required.
!     - binary    The path of the remote binary to use. This might be needed when g++
does not work on one of the hosts
!                 this file is used on. Optional, defaults to "" which will have the
binary compiled on the fly.
!     - filename  The name of the file that will be created. Optional, defaults to
"fakedata".
      """
!
!     size = None         # The size of the file in bytes
!     binary = None       # Path to the remote binary to use
!     filename = None     # The filename the resulting file should have

      def __init__(self, scenario):
          """
          Initialization of a generic file object.

          @param  scenario        The ScenarioRunner object this client object is part
          of.
          """
          core.file.file.__init__(self, scenario)

      def parseSetting(self, key, value):
          """
          Parse a single setting for this object.

***************
*** 68,90 ****
          generic settings parsed and to have any unknown settings raise an Exception.

          @param  key     The name of the parameter, i.e. the key from the key=value
          pair.
          @param  value   The value of the parameter, i.e. the value from the key=value
          pair.
          """
!         # TODO: Parse your settings. Example:
!         #
!         #   if key == 'filename':
!         #       if self.filename:
!         #           parseError( "Really? Two names? ... No." )
!         #       self.filename = value
!         #   else:
!         #       core.file.file.parseSetting(self, key, value)
!         #
!         # Do not forget that last case!
!         #
!         # The following implementation assumes you have no parameters specific to your
file:
!         core.file.file.parseSetting(self, key, value)

      def checkSettings(self):
          """
          Check the sanity of the settings in this object.

--- 53,80 ----
          generic settings parsed and to have any unknown settings raise an Exception.

          @param  key     The name of the parameter, i.e. the key from the key=value
          pair.
          @param  value   The value of the parameter, i.e. the value from the key=value
```

```
            pair.
            """
!           if key == 'size':
!               if not isPositiveInt( value, True ):
!                   parseError( "The size must be a positive, non-zero integer" )
!               if self.size:
!                   parseError( "Size already set: {0}".format(self.size) )
!               self.size = int(value)
!           elif key == 'binary':
!               if self.binary:
!                   parseError( "The path to the fakedata binary has already been set:
{0}".format( self.binary ) )
!               self.binary = value
!           elif key == 'filename' or key == 'fileName':
!               if key == 'fileName':
!                   Campaign.logger.log( "Warning: the parameter fileName to file:fakedata
has been deprecated. Use filename instead." )
!               if self.filename:
!                   parseError( "The filename has already been set: {0}".format(
self.filename ) )
!               self.filename = value
!           else:
!               core.file.file.parseSetting(self, key, value)

        def checkSettings(self):
            """
            Check the sanity of the settings in this object.

***************
*** 92,105 ****
            Any defaults may be set here as well.

            An Exception is raised in the case of insanity.
            """
            core.file.file.checkSettings(self)
!           # TODO: Check your settings. Example:
!           #
!           #   if not self.filename:
!           #       raise Exception( "A dummy file still needs a filename, dummy." )

        def sendToHost(self, host):
            """
            Send any required file to the host.

--- 82,102 ----
            Any defaults may be set here as well.

            An Exception is raised in the case of insanity.
            """
            core.file.file.checkSettings(self)
!
!           if not self.size:
!               raise Exception( "The size parameter to file {0} is not optional".format(
self.name ) )
!           if not self.filename:
!               self.filename = 'fakedata'
!           if not self.binary:
!               if not os.path.exists( os.path.join( Campaign.testEnvDir, 'Utils',
'fakedata' ) ):
!                   raise Exception( "The Utils/fakedata directory is required to build a
fakedata file" )
!               for f in fakedataGeneratorFiles:
!                   if not os.path.exists( os.path.join( Campaign.testEnvDir, 'Utils',
```

```
'fakedata', f ) ):
!                   raise Exception( "A file seems to be missing from Utils/fakedata:
{0} is required to build the fakedata utility.".format( f ) )

      def sendToHost(self, host):
          """
          Send any required file to the host.

***************
*** 115,128 ****
          change the name overriding that method is enough.

          @param  host        The host to which to send the files.
          """
          core.file.file.sendToHost(self, host)
-         # TODO: Send any extra files here. These are the files that are required by
all executions, whether they're seeding or leeching.
-         # Seeding specific files are to be sent in sendToSeedingHost(...).
-         #
-         # Just having the default implementation send the meta file is usually enough.

      def sendToSeedingHost(self, host):
          """
          Send any files required for seeding hosts.

--- 112,121 ----
***************
*** 133,146 ****
          The default implementation does nothing.

          @param  host        The host to which to send the files.
          """
          core.file.file.sendToSeedingHost(self, host)
!         # TODO: Send the actual files to a seeding host. sendToHost(...) has already
been called.
!         # Note that self.getFileDir(...) is not guaranteed to exist yet. Example:
!         #
!         #   host.sendCommand( 'mkdir -p "{0}/files/"; touch "{0}/files/{1}"'.format(
self.getFileDir(host), self.filename ) )

      def getFile(self, host):
          """
          Returns the path to the files on the remote seeding host.

--- 126,160 ----
          The default implementation does nothing.

          @param  host        The host to which to send the files.
          """
          core.file.file.sendToSeedingHost(self, host)
!
!         res = host.sendCommand( 'mkdir -p "{0}/files"'.format( self.getFileDir(host) )
)
!
!         binaryCommand = None
!         if not self.binary:
!             remoteBaseDir = '{0}/fakedata-source'.format( self.getFileDir(host) )
!             host.sendCommand( 'mkdir -p "{0}"'.format( remoteBaseDir ) )
!             for f in fakedataGeneratorFiles:
!                 host.sendFile( os.path.join( Campaign.testEnvDir, 'Utils', 'fakedata',
f ), '{0}/{1}'.format( remoteBaseDir, f ), True )
!             res = host.sendCommand( '( cd "{0}"; g++ *.cpp -o genfakedata && echo &&
echo "OK" )'.format( remoteBaseDir ) )
```

```
!             if len(res) < 2:
!                 raise Exception( "Too short a response when trying to build
genfakedata for file {0} in directory {1} on host {2}: {3}".format( self.name,
remoteBaseDir, host.name, res ) )
!             if res[-2:] != "OK":
!                 raise Exception( "Could not build genfakedata for file {0} in
directory {1} on host {2}. Reponse: {3}".format( self.name, remoteBaseDir, host.name,
res ) )
!             binaryCommand = '{0}/genfakedata'.format( remoteBaseDir )
!         else:
!             res = host.sendCommand( '[ -e "{0}" -a -x "{0}" ] && echo "Y" || echo "N"'
)
!             if res != 'Y':
!                 raise Exception( "Binary {0} for file {1} does not exist on host
{2}".format( self.binary, self.name, host.name ) )
!             binaryCommand = self.binary
!         res = host.sendCommand( '"{0}" "{1}/files/{2}" {3} && echo && echo
"OK"'.format( binaryCommand, self.getFileDir(host), self.filename, self.size ) )
!         if len(res) < 2:
!             raise Exception( "Too short a response when trying to generate the fake
data file {0} on host {1}: {2}".format( self.name, host.name, res ) )
!         if res[-2:] != "OK":
!             raise Exception( "Could not generate fake data file {0} on host {1}:
{2}".format( self.name, host.name, res ) )

      def getFile(self, host):
          """
          Returns the path to the files on the remote seeding host.

***************
*** 155,174 ****

          @param  host        The host on which to find the file(s).

          @return The path to the (root of) the file(s) on the remote host, or None if
          they are not (yet) available.
          """
!         # Note that this is the new name of getName(...), which made no sense in
naming
!         #
!         # TODO: Send the path to the file uploaded to a seeding host. Example:
!         #
!         #   "{0}/files/{1}".format( self.getFileDir(host), self.filename )
!         #
!         # This implementation assumes you don't really have files, which is unlikely
but possible:
!         return None
!
!     # TODO: More methods exist, but they are pretty standard and you're unlikely to
want to change them. Look at core.file for more details.

      @staticmethod
      def APIVersion():
-         # TODO: Make sure this is correct. You don't want to run the risk of running
against the wrong API version
          return "2.0.0"
--- 169,178 ----

          @param  host        The host on which to find the file(s).

          @return The path to the (root of) the file(s) on the remote host, or None if
          they are not (yet) available.
          """
```

```
!         return "{0}/files/{1}".format( self.getFileDir(host), self.filename )

      @staticmethod
      def APIVersion():
          return "2.0.0"
```

Appendix D

Incremental Improvements to the Evaluation Framework

Table D.1 gives an overview of the improvements made to the evaluation framework during the project. The columns show four different versions of the framework. The rows contain the requirements for the framework and the most important feature sets.

The table shows a constant improvement over time in most fields, indicating the constantly increasing needs that has to be met by framework. The most recent version has not been included, since the benchmarking of each version is a time consuming process. The most important changes have been under the hood or extensions to already available features (such as executions containing multiple files). The impact on the table can be summarized as a slightly more complex core, mainly due to more lines of code, but also more and better documentation due to some undocumented parts having been documented and certain steps having been changed to be more logical. Furthermore the documentation in general has improved.

		Bash		Python	
		1.0.0	1.0.5	2.0.0	2.1.0
Requirements	Simplicity	(0) Technical documentation	(+) Technical documentation HOWTO	(+) Technical documentation HOWTO	(+) Technical documentation HOWTO & reference
	Maintainability	Simple core, 9.3k / 44%	Complex core, 14.0k / 41%	Simple core, 12.6k / 55%	Complex core, 14.8k / 55%
	Extendable	163	167	87	88
	Host support	- Local does not function well	0 Local does not function well Supports DAS4	+ Supports DAS4	+ Supports DAS4
	Network conditions	+ netem support	+ netem support	+ netem support	+ netem support
Feature sets	Source locations	Local directory Remote directory GIT repository Subversion repository	Local directory Remote directory GIT repository Subversion repository	Local directory Remote directory GIT repository	Local directory Remote directory GIT repository
	Builders	None make scons	None make scons	None make scons	None make scons
	Data files	Local file	Local file No data Generated fake data	Local file No data Generated fake data	Local file No data Generated fake data Remote file
	Clients	lib*swift*	lib*swift* μTorrent opentracker lighttpd / Aria 2	lib*swift* μTorrent opentracker lighttpd / Aria 2	lib*swift* μTorrent opentracker lighttpd / Aria 2 libtorrent
	Parsers	lib*swift*	lib*swift* lighttpd aria2 μTorrent	lib*swift* lighttpd aria2 μTorrent	lib*swift* lighttpd aria2 μTorrent libtorrent CPU logging
	Hosts	Local host (flaky) SSH	Local host (flaky) SSH DAS4	Local host SSH DAS4	Local host SSH DAS4
	Post-processors	GNU plot	GNU plot	GNU plot	GNU plot
	Viewers	HTML overview	HTML overview	HTML overview	HTML overview
	Workloads				Manual timeout Linear workload Poisson workload
	Debug	Logfile	Logfile Partial stacktraces on errors	Logfile Complete stacktraces on errors/request Full connection logging	Logfile Complete stacktraces on errors/request Full connection logging
	Other				Optimized relative starting time of peers External CPU and memory profiling

Table D.1: Overview of the evolution of the P2P Testing Framework.

Simplicity. All versions have simple dependencies and require the user to build a configuration file before they can run a test. What remains is the documentation, which will be used as the sole characteristic to judge the Simplicity. The following scale is used, depending on the available documentation:

1. (--) No documentation; one has to understand everything before starting;
2. (-) Some, but bad documentation; one will have to look up things in code and figure out details before being able to start;
3. (0) Complete technical documentation; one can get started without reading code or figuring out things but all the documentation needs reading and is not written towards the goal of just getting started;
4. (+) Complete technical documentation and extra guidance, such as Howto's or a full reference; one can get started without figuring out things on their own and besides the complete documentation there are guides to get started quickly;
5. (++) Complete technical documentation, clear extra guidances and readily usable examples; everything is there: full documentation, guides and the really fast copy-paste-and-run examples to get started.

Maintainability. Maintainability is hard to measure, but the original requirement gives some guidance. "The framework should be clean and understandable", which means the structures and techniques used should be easy to understand. In this regard, more code is less easy to understand. The complexity of the core has been noted to be "simple" or "complex", which is both subjective and relative; one could equaly well argue that the core in any version is simple or complex, but compared to each other some versions of the framework are certainly more complex than others. The size of the codebase is a more objective indication of complexity and is given in lines of code (LOC). Tests and external packages were not included in this measurement.

"Documentation of the code is an important factor for this requirement" gives another guide to measuring Maintainability. Although the quality of documentation is very hard to measure care has been taken to document at least each function and each argument. An indication of how well documented the code is can be given by the ratio of code to documentation. All files of each version were copied and stripped of all documentation; counting the lines of code in this stripped version yields the number of lines of real code. Dividing this by the size of the complete codebase and multiplying by 100% gives the percentage of real code. Presented here is the inverse: the percentage of documentation. The following parts of documentation were stripped and are, hence, included in the presented percentage:

1. Comment lines;

2. Docstrings;

3. Whitespace between comments that were obviously added for the purpose of commenting;

4. Methods that could be left out without affecting operation, i.e. overriding methods that just pass on the call to their parent, since they are basically just there to document their existence in the superclass;

5. Skeleton files, since they are basically a form a documentation on how to implement a particular subclass;

6. The declaration of object fields that are also initialized in __init__ in Python, since declaring them has no function in Python and is hence purely documentary.

The table indicates the core of the newest Python version to be complex. This is due to the number of features in the core: the Python code of the core is much cleaner and easier to comprehend, but the latest core simply has more features than the bash versions could have supported. This can also be seen from the core of the last bash version and the core of the first Python version: the featureset of these cores is about the same, but the bash core is indicated to be complex while the Python core is indicated to be simple.

Extendable. Extending the framework should be easy. This means that an extension requires only a small effort to firstly be implemented and secondly work well. The amount of extension points or the documentation of those will not be taken into account here, although better documentation generally lowers the effort to create an extension.

Measuring the amount of effort for a first implementation is done by creating a minimal implementation of the lib*swift* client module, supporting four simple parameters, for each version with the exact same features based on the (documentary) skeleton modules. All documentation apart from error messages and functional whitelines have been stripped from these versions. The length in lines of code (LOC) of each is presented as a relative benchmark of amount of effort required for a first implementation of a module.

The amount of effort to get a module to work well cannot be measured, but good debugging facilities really help lower the effort by finding and pinpointing bugs quickly. The feature list lower on the table includes a list of debugging facilities, giving a hint as to how easy a bug can be hunted down. It should be noted here that stack traces work very well under Python while they are unreliable at best under bash.

Host support. The requirement stated that both local and remote hosts had to be supported, and preferrably more. A simple scale was derived from this description:

1. (-) Baseline not met: either local or remote hosts over SSH do not function (well);
2. (0) Baseline: both local and remote hosts over SSH function, but other support is included;
3. (+) Baseline is met and extra remote hosts are supported.

A (--) has been left out since that would basically mean the framework is not worthy of being called "functioning" and (++) has been left out since the difference with (+) would be arbitrary.

Network conditions. The requirement states that support for non-ideal network conditions is needed. This can be done in several ways and are categorized in increasing order of usefulness:

1. (--) None; no network conditions can be controlled
2. (-) Bandwidth limiting by the client itself; relies on the client to do the work and possibly influences the test results in unexpected ways
3. (0) External bandwidth limiting without other network conditions; allows minimal control over the network conditions but does not directly influence the client
4. (+) External bandwidth limiting and other network conditions; allows full control over the network conditions, but with restrictions such as the need to have root access to the remote host
5. (++) External bandwidth limiting and other network conditiong without restrictions; full control without any questions asked

It has been assumed that a client will not be able to go beyond bandwidth limiting as far as traffic control goes.

Single command. This requirement is absent from the table, since it is basically included in Simplicity.

Features. The table also holds all the feature sets of the different versions. Apart from improving in its core the testing framework also kept growing in the number of features it supports. Below is a small description of what each feature comprises. Note that, even when the same supported feature is present in each version, it is rare for them not to have improved in between versions.

1. Source locations: the source code of a client can be automatically retrieved and built by the testing framework. These are the locations the framework can retrieve the source from;

2. Builders: the source code of a client can be automatically built by the testing framework, but building is never trivial. These are the supported methods for building the source. None is mentioned explicitly to make clear the source code is not required: the testing framework can handle binaries just as well;

3. Data files: these are the data files the testing framework can use to give to the clients to transfer for a test;

4. Clients: the clients that are supported. Note that libtorrent is supported using a special-purpose wrapper program, only;

5. Parsers: while a client is running it usually generates a log in a client-specific format, the parsers are used to interpret these logs and rewrite them to more standardized formats. The dummy parsers are not mentioned;

6. Hosts: the types of hosts the testing framework can use to run experiments on;

7. Post-processors: after all data has been gathered and parsed any post-processors can be run to process them further. Post-processors that simply save some data about the execution are not mentioned;

8. Viewers: after all data has been processed viewers can make an overview of it all;

9. Workloads: control the timing of when executions are started on the remote hosts relative to the beginning of the experiment;

10. Debug: facilities provided for debugging;

11. Other: other features that are not mentioned under the above categories.

Not mentioned in the table are automated .torrent generation, automated Merkle root hash calculation and .torrent file mangling to support dynamically created trackers using the opentracker client. The .torrent creation has been available in all versions, where the other two were available in all but the first bash implementation. All these automated tasks were severely restricted in their operation and performance under the bash implementation.