Software Performance Engineering in Complex Distributed Systems

Laurens F. D. Versluis
L.F.D.Versluis@student.tudelft.nl

Software Performance Engineering in Complex Distributed Systems

Master's Thesis in Computer Science

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

Laurens F. D. Versluis

19th August 2016

Author
Laurens F. D. Versluis

Title
Software Performance Engineering in Complex Distributed Systems

MSc presentation
Dijkstrazaal, HB09.150
EEMCS, Delft
15:00-16:00, August 29, 2016

Graduation Committee

dr. ir. J. A. Pouwelse	Delft University of Technology
dr. ir. M. Zuniga	Delft University of Technology
dr. ir. A. Bozzon	Delft University of Technology

Abstract

Performance is a make-or-break quality for software. When making changes it is essential to ensure no *performance regression* has occurred i.e. the program performs more slowly or consumes more resources than previous versions. Tribler is the result of ten years scientific research in complex distributed systems. Over the course of years Tribler's performance has fallen below acceptable user experience levels, mainly because there is a lack of *software performance engineering*.

In this work, we lay the foundations for a regression testing systems that allows developers to continuously monitor the metrics that are having the most impact on the performance of Tribler. Applying this system gave us a deep insight in the greatest bottleneck of the performance of Tribler: synchronous, blocking disk operations. Resolution of this bottleneck includes a major refactoring effort of the message synchronization system Dispersy. Implementation of a novel, non-blocking disk operation framework allowed us to increase the throughput of Tribler's API by up to 150% and reduce its response times by up to 57.5%.

Preface

This thesis presents the work I have conducted in the last ten months. During this period, I have had the pleasure of meeting new people who made my time much more enjoyable. I would like to thank some people who contributed to this thesis. Firstly, I would like to thank Johan Pouwelse for offering me this thesis and providing feedback on my work. Secondly, I would like to thank Elric Milon who shared his knowledge of Tribler, Python, Git and Emacs with me as well as maintaining and updating our infrastructure from which everyone profited. Thirdly, I am grateful to Pim Otte for reviewing my work and providing feedback. I would also like to thank both Martijn and Hans, who I have spent the most time with in the MSc lab and for occasionally helping me when struggling with Linux. Next, I would like to thank Ernst, Niels, Paul and everyone else of the Tribler team for providing feedback and suggestions. Last but not least, I would like to thank my friends and family for their support, motivation and encouragement throughout my thesis.

Laurens Freydis Dene Versluis

Delft, The Netherlands
19th August 2016

Contents

Chapter 1

Introduction

Performance is a make-or-break quality for software. In today's society we expect manufacturers to progress in their expertise and their products to advance. We take for granted that new cars have a larger range, are more powerful, less pollute and more safe with each iteration of design. In contrast, software does not has this image; software updates breaking functionalities, adding severe performance issues or increasing the complexity of the user interface are common. The Facebook mobile Android application is a prime example of this, having more than one billion installs and being in development for years by a complete development department still receives updates which causes performance issues for users, see Figure 1.1. Just as with their cars users expect software to advance or at least not become *worse* in terms of performance.

Interestingly, both industry and academia agree performance plays an important role in software development. Smith et al. have shown that the cost of a software product is determined more by how well it achieves its objectives for quality attributes such as performance than by its functionality [1]. For instance, an increase of 500 milliseconds latency in Google's search results could cause 20% traffic loss [2]. The deterioration of performance introduced by changes is often referred to as *performance regression*. Performance regression can lead to several undesired consequences such as damaged customer relationships as the software does not meet its required performance. Huang et al. provide the example of an e-commerce website that saw an increase of 2000% in their page loading times because of an update to the underlying database engine [3]. These situations can lead to lost revenue and possibly missed market windows. Other consequences of performance failures may express themselves in lost productivity for users, increased costs, failures on deployment or even abandonment of projects [4, 5].

Performance can be divided in two dimensions: *responsiveness* and *scalability*. Responsiveness is the ability of a system to meet the requirements for response time or throughput. The response time can be measured by how fast a system can respond to an event, where throughput can be measured by how many events can be processed in a set amount of time. Scalability is the ability of a system to meet

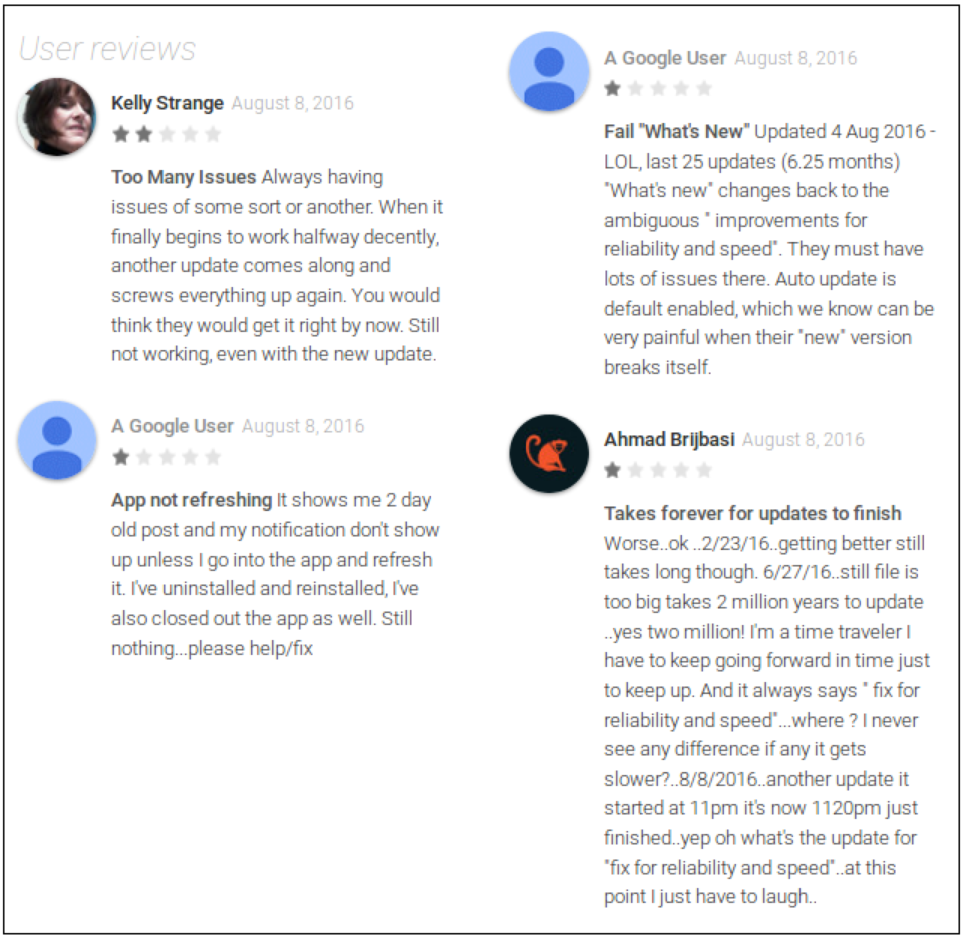


Figure 1.1: Reviews showing the Facebook Android application introducing issues with updates.

the required response time or throughput when faced with a growing demand of its software functions.
To inspect if one of these two dimensions has decreased when changes are made, performance regression testing can be applied. Using this method, the system is tested for performance regression under various loads [4]. Traditional software development focusses on correctness, causing regression testing to be deferred to a late stage in the development cycle, if applied at all. To illustrate the varying amounts of regression testing appliances, Huang et al. mention the performance testing intervals of MySQL, Linux and Chrome. These projects apply performance tests every release, every week and every four revisions, respectively. Once performance regression is detected, developers have to spend extra efforts determining what causes said regression, especially when a lot of changes have been applied to the code base since the last measurement [3]. Performance tests should be executed as frequently as possible, ideally per change made by developers. However,

as some tests take hours or even weeks, this approach is not always feasible.
Software performance engineering (SPE) is the discipline concerned with constructing software systems that meet performance objectives [6]. It prescribes principles for creating responsive software, methods to obtain performance specifications and offers guidelines for the types of evaluations to be conducted at each development stage. SPE features two general approaches [4] where the first approach is purely measurement based. This characterizes itself by performing actions late in the development cycle such as applying regression tests, diagnosis and tuning, when the system can be run and measured in real-time. The second approach features a model-based style. Using this approach, performance models are created at the early stages of development, influencing the architecture and design of the system to meet performance requirements.

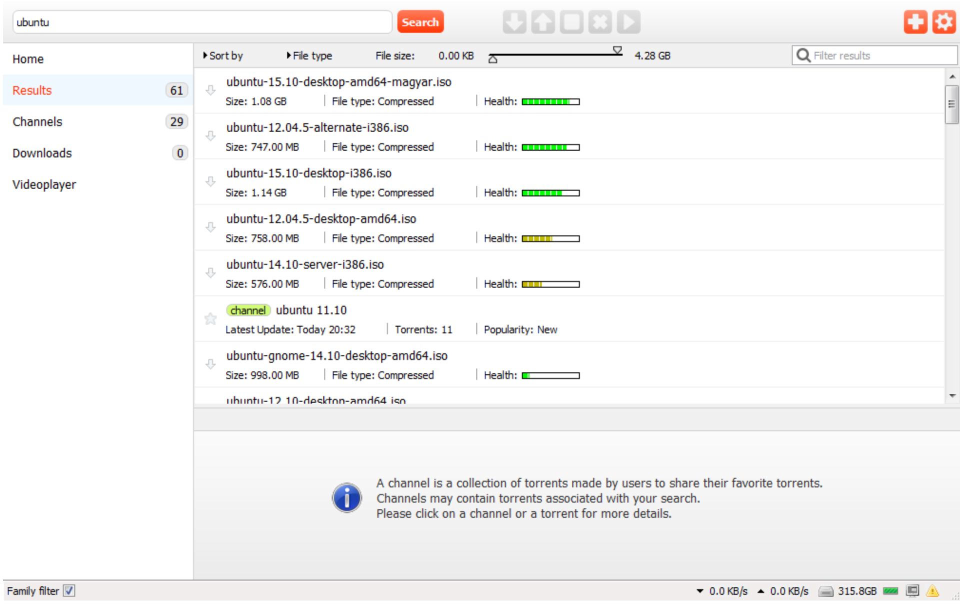


Figure 1.2: Screenshot of Tribler v6.5.2.

Tribler is a BitTorrent client with a focus on secure, private and decentralized communication. It is the result of ten years of scientific research in the field of decentralized systems, online cooperation and other disruptive technologies such as onion routing, blockchain and cryptocurrency. Over 100 scientific publications have Tribler in their foundation. Tribler is completely open-source and can be downloaded from the organisation's website[1]. Tribler's interface is visible in Figure 1.2.
Over the course of more than a decade Tribler has gained a tremendous amount of attention in both media and academia. It has been downloaded approximately 1.8 million times [7] and has more than two thousand monthly active users. This makes Tribler one of the research projects that allow researchers to run experimental code

[1] `https://www.tribler.org`

‘in the wild’ on a large scale.
One of Tribler’s unique features is that it allows users to discover and exchange data in a complete decentralized way. This introduces additional challenges compared to centralized solutions.
Using a centralized structure, heavy computations can be offloaded to servers which are often more powerful and more responsive than a consumer grade computer. Once the result has been computed, it can be communicated back to the user at only the cost of the communication. It has been demonstrated that for devices with a finite power supply such as smartphones, this technique can be applied to save energy or to boost performance [8, 9].
Decentralized systems such as Tribler do not have such servers present that can be offloaded to, restricting the area of computation to the device itself. It is thus essential that the software performs well on heterogeneous devices, possibly facing challenging network conditions.
To enhance privacy and anonymity, support for anonymous downloads was introduced in 2014 by R. Plak [10] and R. Tanaskoski [11]. In 2015, the support for anonymous seeding of torrents using Tor-like hidden services was added by R. Ruigrok [12]. A trade-off has to be made by Tribler between the desired performance and the level of anonymity provided, as any additional layer of privacy comes with an increased number of cryptographic operations.
Not only anonymity impacts the overall performance of Tribler: architectural flaws introduced in the past have led to a decrease in performance today. This manifests itself in users frequently reporting high system load, a non-responsive Graphical User Interface and low download speed. Furthermore, Tribler is plagued with a high number of disk operations which has been known since 2013 [13].
The focus of this thesis is to improve Tribler’s performance by making use of software performance engineering techniques in the late stages of the development cycle with a particular focus on the performance of disk operations and software regression testing.
The rest of this thesis is structured as follows. Chapter 2 provides the problem description and the research questions this thesis attempts to answer. Chapter 3 explains the Python threading model and why we observe performance issues with Tribler. Chapter 4 presents the design and implementation of a new database manager that will solve Tribler’s disk operation issues. Chapter 5 presents how we incorporate software performance engineering into Tribler’s development cycle by adding a software regression testing system. Chapter 6 presents experimental results showing Tribler’s performance has improved and validates the software regression testing system introduced. Finally, Chapter 7 concludes this thesis and provides future work.

Chapter 2

Problem description

Tribler's goal is to offer a YouTube-like experience with similar performance and ease of use. All Tribler's features are implemented in a completely decentralized manner, not relying on any centralized component.
Numerous initiatives exist around these goals of re-decentralisation and performance. However, none of them gathered any significant usage compared to the social media usage levels. For instance, YouTube features one billion unique monthly users [14] and there are 1.8 billion monthly active Facebook users [15].
The problem is that the performance, usability, and features offered by decentralised alternatives are inferior when compared to the experience offered by central solutions. Creating academically pure self-organising systems such as Tribler has proven to be notoriously difficult. For example, the extensive list of 194 projects which all aim to create an alternative Internet experience using decentralisations shows the amount of years spent and lines of code produced [16]. Most of these projects are abandoned and few of them have actual real-world usage.

- [ ] *Tribler anonymous downloads are fast*
  Only 330KByte/sec is unacceptable for our video streaming use-case. Currently our GCM cypto or other bottleneck is slowing things down. #1882
- [ ] *User experience similar to Youtube*
  Toward a Youtube-like experience within Tribler. Use the thumbnails pre-view of videos as the key navigation and content selection tool. Integrate our recommendation engine to the GUI, like Youtube. #1846. **@devos50**
- [ ] *Anonymous Video streaming, similar experience to Youtube on-demand service*
  Fast and robust downloads from hidden seeds. Users may need to wait a minute, but it simply works. Anon swarming: #1885 **@lfdversluis**

Figure 2.1: Three of the six uncompleted Tribler roadmap items.

The Tribler project created a roadmap – available on its GitHub repository – to offer the same service, features, user experience, and performance as the YouTube video-on-demand service. However, especially the poor performance of Tribler is hampering wide-spread adoption and usage. Figure 2.1 shows three of the six main uncompleted roadmap items.

2.1 Key Performance Optimizations

Tribler can be seen as a large and complex distributed system. Metrics from Open-HUB show that Tribler, along with its components, features more than 169 thousand lines of code, received contributions from 111 unique contributors and took approximately 44 years of effort [17].

In large and complex systems there are likely to be many performance issue present, often referred to as *bottlenecks*. J. M. Juran's Pareto principle admonishes that one should "Concentrate on the vital few, not the trivial many" [18]. This principle is also known as the 80/20 rule. Concretely, this means that resolving the vital bottlenecks yields the best diminishing returns, even for large systems such as Tribler. After careful scrutiny it was decided that the most vital bottleneck to address, within the context of a nine month thesis, is Tribler's database I/O.

2.2 Addressing Blocking I/O

The problem we address within this thesis is the underlying reason for poor performance and unacceptable user experience. Measurements dating back from 2013 indicate that Tribler's performance is I/O-bound [13]. Especially with slow hard disks, but also with fast SSD storage the main performance bottleneck seems to be around database access. With our focus on the fundamental issue we believe we can make a significant step forward in making decentralized technology able to compete with centralized solutions on large-scale usage.

All information within Tribler is stored in a database for persistence and ease of use. Information about the network i.e. peers, messages and authentication is stored in a separate database managed by the Distributed Permission System (Dispersy). Dispersy is an elastic database system written in the Python programming language and uses SQLite as its underlying database engine. It lies at the heart of Tribler, providing the means to discover peers and content in a decentralized way while offering security and anonymity.

Dispersy is fully decentralized with the exception of bootstrap servers. It can run on systems with a large number of nodes, without any sever architecture needed [20, 21]. All nodes perform the same algorithmic procedures and tasks and do not differentiate between any node i.e. all nodes are equal.

Furthermore, Dispersy provides one-to-one and one-to-many data dissemination mechanisms to forward data to nodes. Eventually, all data will reach all nodes in the network, overcoming challenging network conditions. The current overview of the database structure is presented in Figure 2.2 (Johan Pouwelse, 2013).

These databases are becoming a key performance bottleneck [13]. Back in May 2013 measurements indicated that Tribler read and wrote 660 Megabytes per hour

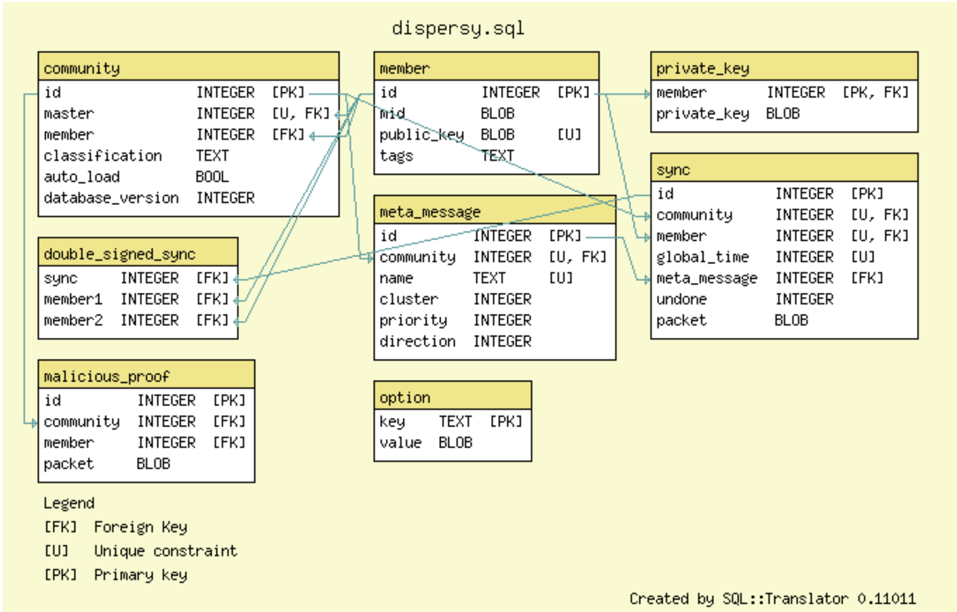


Figure 2.2: The database schema of Dispersy, (source: Johan Pouwelse, 2013) [19].

to and from disk. The next measurement in April 2014 showed this number was somewhat reduced to 623. In May 2014 efforts were made to reduce this enormous amount of I/O; by batching database statements the number dropped to 538 megabytes per hour.
So far this metric has only been measured sporadically by hand, running Tribler for an arbitrarily amount of time and check the amount of I/O by using htop[1]. htop produces an overview similar to Figure 2.3 (Johan Pouwelse, 2014). Measurements to observe to which extent Dispersy is responsible for these numbers were never conducted, however it is strongly suspected by the Tribler developers that Dispersy is responsible for most of it. Since 2014 no work or measurements have been done related to this issue.

2.2.1 Blocking I/O

One of the main causes of Tribler's performance issues can be explained by the blocking behaviour of I/O. Currently, Dispersy is deeply embedded into Tribler, running on the same (main) thread Tribler is running on. Tribler, just like Dispersy, is written in the Python programming language. In Python, a thread performing an I/O operation will block, causing all operations on the thread to suspend. This means whenever Tribler or Dispersy performs I/O all functions of Tribler and Dispersy halt. With the enormous amount of I/O Tribler is performing, this forms a

[1] `http://hisham.hm/htop/`

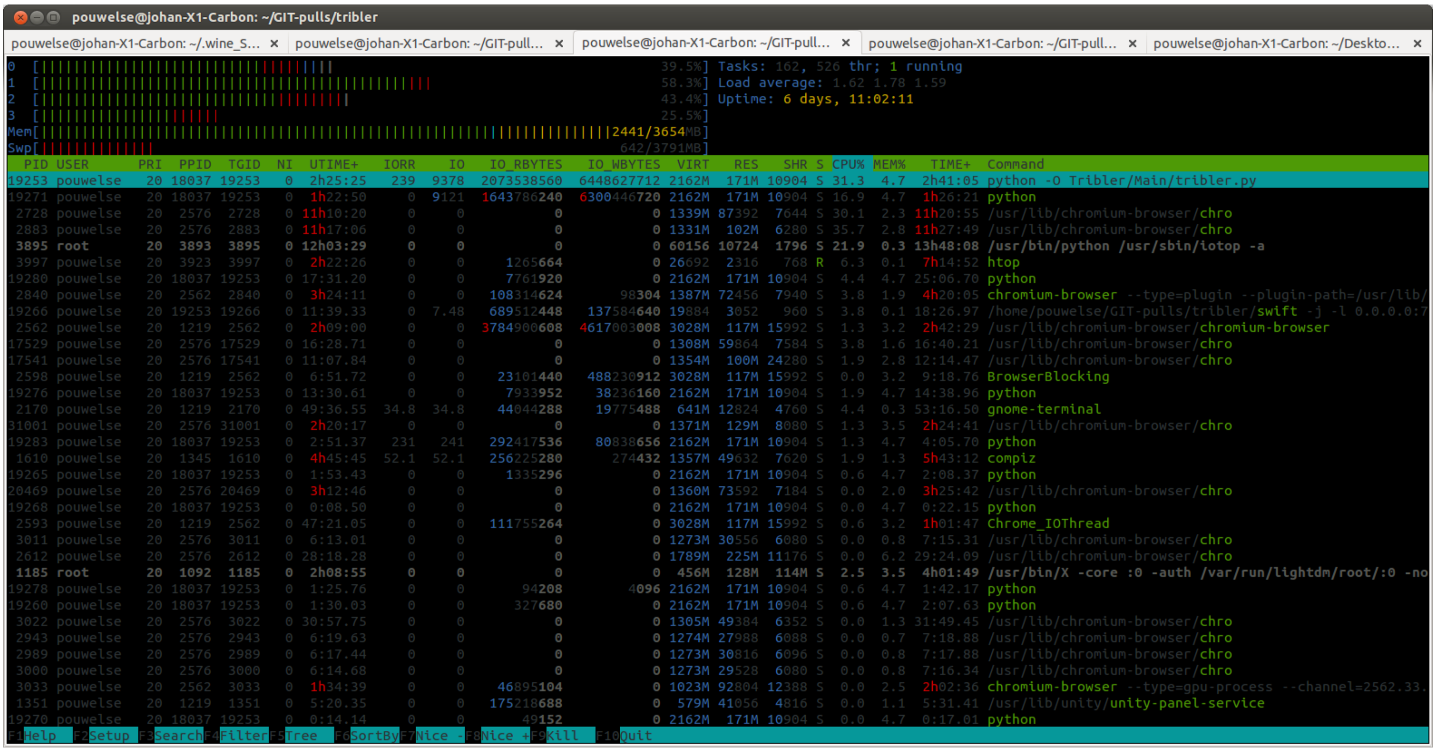


Figure 2.3: A screenshot of htop showing Tribler's I/O, (source: Johan Pouwelse, 2014 [13]).

huge limiting factor on the responsiveness and therefore performance of Tribler.
When a thread suspends, other threads can take over and perform operations, yet besides the main thread there are only two additional threads in Tribler: the Graphical User Interface (GUI) thread and the Dispersy endpoint thread. As the name implies, the GUI is running on the GUI thread as the framework Tribler currently uses requires this. Ironically, the Dispersy endpoint thread was introduced because of the blocking I/O behaviour. Under heavy load, Dispersy drops packets because it cannot keep up. Processing packets is done on the main thread and as this thread frequently blocks, the buffers overflow causing packet loss. These two threads do not saturate the available processing time offered by the main thread blocking, leading to wasting valuable CPU cycles.
Furthermore, Tribler has seen several changes to its code base including the addition of the MultiChain: Tribler's own Blockchain-like structure [22]. This feature heavily relies on its database to store blocks and other information about the user and other peers. Moreover, the MultiChain makes use of its own database rather than Dispersy's. Norberhuis points out: 'The information is stored in two places within Tribler and this could be eliminated. It would reduce the disk footprint and the amount of read/write transactions as only one database would have to be maintained. The I/O ineractions[sic] are a problem according to Tribler maintainers.'

[22], yet numbers on how much I/O the MultiChain generates are not presented. This makes it hard to estimate Tribler's current I/O rates.
What's more, a feature called 'credit mining' is currently in development that will also interact with the database of Tribler. There are no metrics on the current situation of Tribler and it's hard if not impossible to estimate the impact of any addition to come. Currently, there is a lack of insight in these metrics, or a lack of *software performance engineering*, causing the exact extent of the problem to be unknown.

2.3 A Lack of Software Performance Engineering

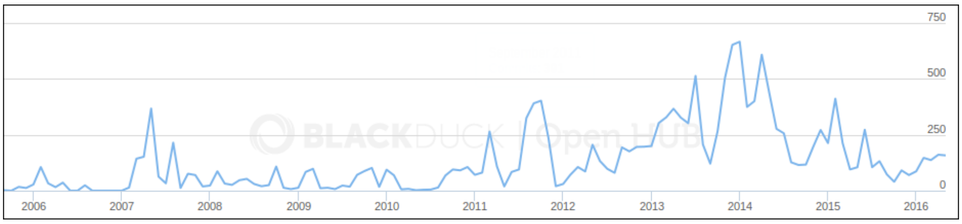


Figure 2.4: The distribution of commits on Tribler (source: OpenHUB, 2016 [17]).

Nowadays, programs are evolving at a rapid speed and Tribler is no exception. Since 2005 Tribler is under continuous development, over seventeen thousands commits have been made, spanning more than a decade [17]. The distribution of these commits can be seen in Figure 2.4. Code commits to fix bugs, refactor code, enhance or add functionality are pushed to the code repository of Tribler at a frequent rate. It is important that software performance engineering is a part of this process: performance should not get compromised by changes, if any it should improve. Naturally trade-offs can be made performance wise, but it should be done with a clear understanding of the consequences.
For the past three years, attempts have been made to monitor the performance, yet with little success. The first attempt was to create probes using Systemtap[2]. System tap is a tool for 'instrumenting the Linux kernel for analyzing performance and functional problems', (Jacob et al. 2008) [23]. While some success was reported, this system is no longer being maintained nor functional. After this, code was added to the Dispersy code base to track and log if a function was running longer than a fixed amount of time. While this implementation does provide some insight, its workings are crude and only covers some of the functions present. For instance, it cannot handle asynchronous constructions. Tribler did not have any observation system integrated, leaving the development team in the dark regarding its performance.
It is apparent that there is a lack of software performance engineering in the development cycle of Tribler. Performance has never been one of the priorities in Tri-

[2]`https://sourceware.org/systemtap/`

bler's lifetime: only 6% of all tickets on GitHub are (indirectly) related to performance based on their content. To ensure performance will no longer degrade, realistic benchmarks need to be developed which Tribler can be tested with. Changes can then be compared against the current code base, tracking important performance statistics such as the amount of I/O, run time of functions, throughput and responsiveness. These benchmarks can then be integrated into a regression testing system which can be integrated in our Jenkins continuous integration system. Using Jenkins, performance regression tests can run on every proposed change and at predetermined moments, allowing for a continuous updated overview of Tribler's performance metrics.

2.4 Realistic Benchmarks

Directly related to the performance problem is the benchmark problem. In order to improve user performance we require making assumptions about realistic use cases. Each user has different usage patterns, hardware and network conditions, all affecting performance. Creating one of more several benchmarks testing several scenario's is required to accurately tune the system for real world usage. At the same time, a benchmark cannot consume too much time. Benchmarking is by nature time consuming [3], however running long regression tests per commit will severely strain the development speed.
Therefore, it is important to create a reference benchmark which has a close resemblance to real world usage without consuming too much time.

2.5 Objective and Research Questions

The objective of this thesis is to improve the performance and responsiveness of Tribler and to introduce a regression testing system. The verification of the performance regression testing system is done by focussing on removing Tribler's biggest bottleneck present: blocking database I/O. By resolving this bottleneck, important metrics tracked by this regression testing system should show positive changes, indicating improvement.
The research presented in this thesis was carried out in cooperation with the Tribler team. The Tribler team consists of both staff members of the Technical University of Delft as well as Bachelor and Master students. Based on the objectives of Tribler, this thesis aims to answer the main research question formulated below.

Main Research Question: How can we improve Tribler's performance, responsiveness and throughput?

To answer this main research question, we have defined three research questions below. Each of these research questions will be justified as why they contribute to the main research question.

Research Question 1: Can a system such as Tribler benefit from asynchrony?

To improve performance and responsiveness, parts of Tribler can be rewritten to become asynchronous. By performing tasks asynchronously the performance and responsiveness of a program can improve. However, an asynchronous approach can have its drawbacks. One of these drawbacks is that it requires a different mindset for the programmers as the whole call chain and structure of a program becomes different. Identifying these drawbacks and deciding if the benefits outweigh the costs is necessary to prevent the current state from worsening.

Research Question 2: How do we resolve Tribler's blocking database I/O problem?

As database I/O is currently the main bottleneck, we need to resolve it. Tribler already has a framework integrated that is especially designed to handle I/O in a non-blocking way. We require an approach that prevents us from reinventing the wheel while still solve the bottleneck at hand in an adequate manner. Additionally, we need to make sure the order of database operations does not change as this may lead to inconsistencies.
Careful scrutiny is required to look at available solutions and their pros and cons. A decision can then be made on the best course of action.

Research Question 3: How do we incorporate software performance engineering into Tribler's development process to gain insight into performance statistics?

Currently, not a single developer has insight into how well Tribler performs and what impact changes have on Tribler in its current state. To be able to conclude performance changes do not negatively impact the performance of Tribler, we can apply software performance engineering. Software performance engineering focuses on introducing performance regression tests and benchmarks into the development cycle. By making use of these regression tests, Tribler developers finally get insight into vital metrics which is desperately needed.

2.6 Main Contributions

The main contributions of this thesis are as follows. First, we elaborate on the subject of multitasking and parallelization in the context of the Python programming language and provide arguments where asynchronous programming is preferred over synchronous programming, using Tribler as a case study. We then resolve the vital I/O bottleneck currently present in Tribler using asynchrony and a multi-threaded approach. This is done by introducing the Storm database framework into Tribler and creating a database manager with an asynchronous, non-blocking yet serialized interface. Next, we introduce software performance engineering into Tribler's development process by adding a regression testing system that benchmarks different versions of the same code base to gain insight into changes in performance metrics such as disk I/O. Finally, experimental results and measurements will be provided to confirm the main goal of this thesis i.e. improving responsiveness, performance and throughput.

Chapter 3

Python's Threading Model

Before we can address Tribler's blocking I/O problem, we first need to better understand Python's threading model and how the Python interpreter behaves. By looking in depth how the operating system (OS) and the Python interpreter behave together, we can look into how multiple threads can be beneficial to Tribler. Additionally we take a look at asynchrony in Python to see if we can apply this in Tribler to gain performance by improving the responsiveness. This allows us to answer the second research question 'Can a system such as Tribler benefit from asynchrony?' and gives us a direction of design and implementation for our blocking I/O problem.

3.1 Threads in Python

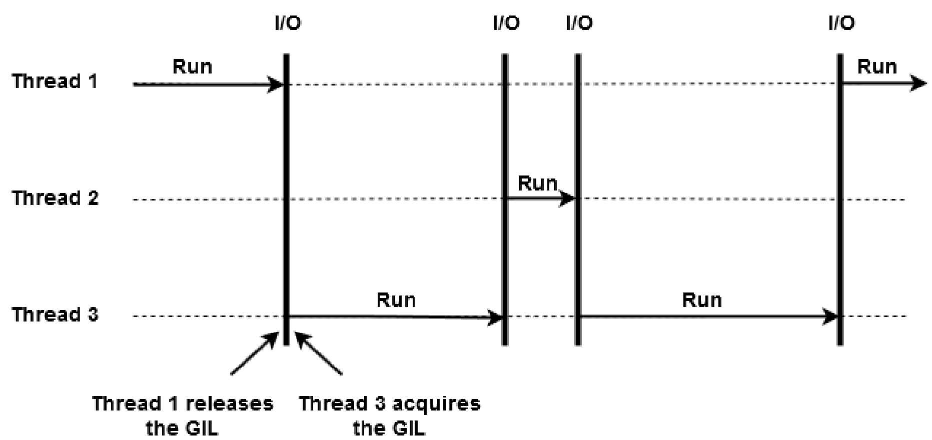


Figure 3.1: A schematic view of how threads release the GIL when performing IO in Python.

Python threads are normal OS threads, either POSIX (pthread) or Windows threads

[24, 25]. Python does not feature a thread scheduler; threads are fully managed by the operating system that hosts them. To control thread execution Python features a construct called the Global Interpreter Lock (GIL). This GIL ensures that only one thread can run in the Python interpreter at any time, i.e. a thread needs to hold the GIL in order to execute. This means there cannot be any parallel execution. Once a thread is done executing or needs to block it releases the GIL. This gives way for cooperative multitasking as other threads that are ready to execute can try to acquire the GIL, visualized in Figure 3.1.
To make sure threads running a Central Processing Unit (CPU) bound tasks do not hold the GIL indefinitely a simple check mechanism is built-in that 'checks' every thread once per 100 ticks. Ticks are loosely mapped to interpreter instructions and do not define a time unit. Listing 3.1 contains two code samples that only take one tick each, but require a different amount of time to compute.

```
a = 5 * 5 # A fast one tick instruction

numbers = xrange(100000000)
b = -1 in numbers # A slow one tick instruction
```

Listing 3.1: Two code samples that each take one tick yet require a different amount to compute.

When a check is run the following four steps are executed:

1. The thread that holds the GIL resets its tick counter.
2. If the current thread is the main thread, it runs the signal handlers.
3. The thread releases the GIL.
4. The thread tries to reacquire the GIL.

Note that a thread may immediately reacquire the GIL after releasing it. Since every thread has this check, CPU-bound threads will engage in cooperative multitasking.

3.2 Multi-threaded Programming Performance

Because threads cannot run in parallel, this changes the performance one may expect from a multi-threaded program. David Beazly presented his findings in his Python Concurrency Workshop (2009) [25]. By running a trivial CPU-bound function using two threads on a dual-core MacBook, processing time increased from 24.6 to 45.5 seconds; an increase of 185%. Disabling one of his CPU cores yielded a run time of 38.0 seconds, an increase of 154%.
To confirm this behaviour still exists in the latest version of Python2, we rerun Beazly's experiment. The setup used in this experiment can be found in Table 3.1. Running it sequentially gives us a run time of 8.17 seconds, whereas running it

Table 3.1: Specifications of the setup used during the rerun of David Beazly's experiment.

Component	**Specifications**
Operating System	Ubuntu 16.04 LTS
Python version	2.7.12
CPU	Intel Core i5-2410M
HDD	Samsung 850 EVO 250GB
RAM	8 GB DDR3 1600MHz

using two threads provides us with a run time of 14.49 seconds; an increase of 177% which is in range of Beazley's observations.
To investigate this behaviour, Beazley studied the underlying C code to inspect why he was observing these performance results.
Whenever a (Python) thread releases the GIL, it sends out a signal. The OS then propagates this signal to other threads which then can attempt to acquire the GIL. The time or *lag* between thread switching and execution may be significant depending on the OS, according Beazley. Most operating systems make use of a priority system for threads: the thread with the highest priority will be scheduled by the OS. Often, CPU-bound threads have a low priority and I/O-bound threads a high priority. If a signal is send to a low priority thread and all CPUs are currently busy processing other, high priority threads, it won't be run until one of the CPUs becomes available.
As it turns out, the GIL signalling is the source of the performance loss. Whenever the periodic 'check' runs, the following happens:

- First the Python interpreter locks a mutex.
- Next, it signals on a condition variable/semaphore where another thread is *always* awaiting execution.
- Because of this waiting thread, additional pthread processing and system calls are generated to deliver the signal.

Beazley provided rough measurements on the number of system calls generated, summarized in Table 3.2. From these numbers Beazley concludes that the amount of additional calls generated is significant and the main reason for the performance loss. He then dived deeper into these numbers to find a cause.
By recording a real-time trace of all GIL related operations i.e. acquisition, release, etc., Beazley was able to reconstruct the key problem. When running multiple CPU-bound threads on multiple cores, all of them will be scheduled *simultaneously*. The threads then proceed to battle over GIL acquisition. Whenever a thread releases the GIL because of the 100 tick 'check', it immediately tries to reacquire it. Another thread will also try to acquire the GIL upon this signal, but

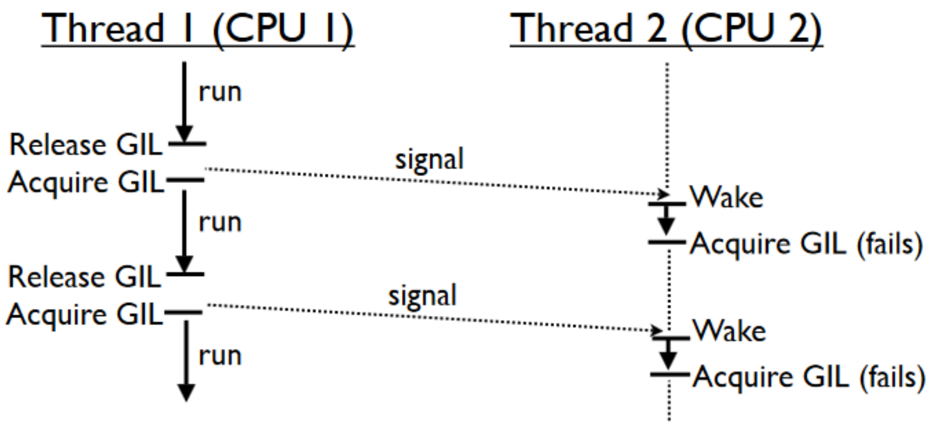


Figure 3.2: A schematic view of two threads battling for GIL acquisition (source: David Beazley, 2009).

Table 3.2: A summary of the measurements from David Beazley [25].

Threads	Cores	Unix system calls	Mach system calls
1	1	736	117
2	1	1149	3.3M
2	2	1149	9.5M

as this signal arrives with a delay it will most likely fail. This processes is visualized by Figure 3.2 (David Beazley, 2009). Beazley argues that here two competing goals are clashing. On the one hand Python wants to only run one thread at a time where on the other hand the OS wants to schedule as many processes/threads to take advantage of its multiple core architecture. This clash raises a lot of overhead, which results in a severe performance penalty.
Even running threads on one core result in these GIL acquisition battles. I/O-bound threads which are high priority may fail to acquire the GIL when a CPU-bound thread is busy, degrading the response time of the I/O thread.

3.2.1 A new GIL

To solve the excessive amount of GIL signalling, a new GIL has been introduced in Python 3.2 [24]. Instead of having the 100 ticks 'check', there is now a global variable. A thread will continue running until this variable is set to one (1), indicating another thread requests to acquire the GIL at which the running thread must release the GIL. This means whenever only one thread is running, the variable will never be set (there is no competing thread) and no signalling takes place. Whenever another thread is present, it will be in a suspended state as it does not hold the GIL, and starts a five millisecond timeout. From here two things can hap-

pen: the running thread releases the GIL voluntarily within the timeout (e.g. an I/O operation is performed) at which the other thread can immediately acquire it, or the timeout expires. If the timeout expires, the other thread will set the global variable to one and enters another timeout, awaiting a signal that the GIL has been released. The running thread will release the GIL, sends out a signal that it has done so and starts a wait period. When the other thread acquires the GIL it will send out an acknowledgement to the thread that just released the GIL and starts its computations. Finally, the former running thread will now enter a timeout upon receiving the acknowledgement so that it may reacquire the GIL again.
To observe if the new GIL offers better performance, Beazley ran the same experiment using Python 3.2 and the results look promising. Running the code sequentially, using two threads and using four threads on a quad core MacBook resulted in run times of 11.53, 11.93 and 12.32 seconds respectively. Unfortunately, there are caveats when performing I/O operations using the new GIL. Whenever an I/O operation does not block, the thread still releases the GIL which requires the thread to initiate the timeout again to reacquire the GIL. Meanwhile CPU bound threads will also attempt to acquire the GIL since it was released, causing stalls. Beazley argues that some more work is required on the GIL to get rid of this behaviour, yet expresses that even with the GIL present threads can still deliver excellent performance and programmers still should use threads when appropriate.
Unfortunately, Tribler currently runs on Python 2.7 and a considerable amount of work is required to migrate to Python3. However, as the new GIL does look promising with respect to blocking I/O it can be noted down as an item for future work.

3.3 Attempts to Parallelize Python

Parallelize Python by removing the GIL and other means to speed up Python are topics that regularly return in the Python mailing list and at PyCon [26]. Throughout the years many attempts have been made to alter Python or remove the GIL to fully benefit from multiple CPUs. While many attempts are software related, some alternative implementations have been applied to hardware [27]. To date, no one has succeeded in removing the GIL and meet the (hard) requirements for replacement [26].
One of the most well-known alternative implementations of Python is PyPy. It makes use of a tracing Just-in-Time compiler to produce optimized code [28]. By doing so, PyPy offers increased speed, reduced memory usage and support for stackless mode while providing a high compatibility with existing Python code [29]. PyPy's geometric average on run time is 7.6 times faster than CPython (normal Python) [30]. While it has many popular libraries ported to be used with PyPy, it still lacks some common used packages. Moreover, most of these libraries are not available on the official packaging repositories of Ubuntu and/or Debian, rendering PyPy unusable for the Tribler project as publishing Tribler on said repositories

requires the dependencies to be available on them as well.
Two other popular implementations are JPython and IronPython. Both these projects have removed the GIL and can fully exploit multiprocessor systems [26].
JPython is a Python interpreter implemented in Java. It can be integrated in Java applications and allows Python applications to be compiled into Java classes. Using JPython, Python – after compiled to Java bytecode – will run in the Jython virtual machine, giving full access to all Java APIs and classes [31].
IronPython does basically the same as JPython, compiling the source to in-memory bytecode and runs it on the Dynamic Language Runtime [32]. It allows developers to run Python using the .NET framework.
To illustrate the attempts to remove the GIL are still ongoing, Larry Hastings presented ‘The Gilectomy’ at PyCon 2016. He showed that removing the GIL is fairly easy, but has a huge negative impact on CPython’s performance, cache misses being the main reason. Additionally, Hastings names some methods that may make The Gilectomy a viable alternative to CPython.

Libraries that introduce parallelism or asynchrony to gain performance are also available in large numbers. In particular, many projects exist that attempt to make I/O asynchronous [33]. Often, these leverage the multiprocessing package in the standard library of Python.
Decorated Concurrency (DECO) uses the multiprocessing package of Python to parallelize functions using ‘concurrent’ and ‘synchronized’ decorators [34]. Different processes have their own Python interpreter which in turn has its own GIL, allowing for parallel processing. Using the ‘concurrent’ decorator, a function will be wrapped and executed on a new process. Similarly, the ‘synchronized’ decorator inserts synchronization events to automatically refactor assignments of the results of ‘concurrent’ function calls to happen during synchronization events. However, this approach is only viable for heavy loads that generally can run on their own as constructing a new process and communicating between different processes generate overhead. The authors mention that a function should at least have an execution time of one millisecond for this method to be beneficial.

3.4 Asynchronous Programming

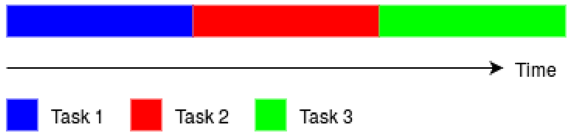


Figure 3.3: Synchronous processing of tasks

Another method to possibly improve Tribler’s performance is to make use of asynchronous programming. Traditionally, synchronous programs execute tasks se-

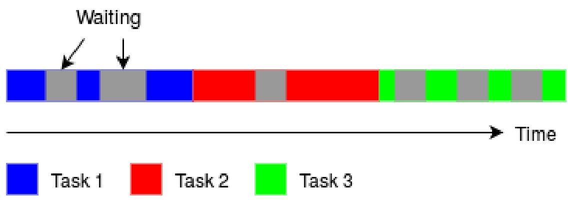
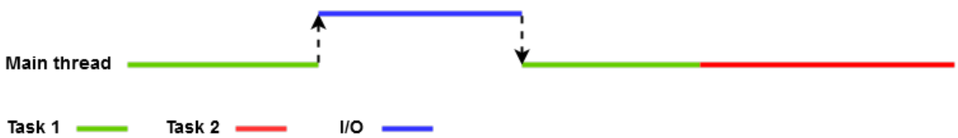
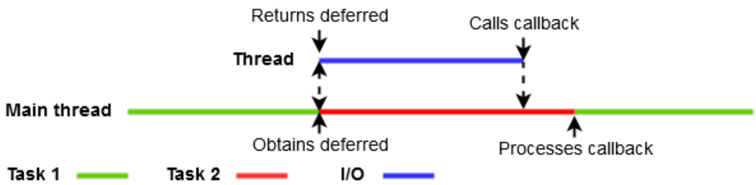


Figure 3.4: Tasks will be waiting when disk or network has to catch up.

quentially as visualized in Figure3.3. When these functions perform an I/O operation, they are waiting for the disk or network to catch up, visualized in Figure 3.4. In turn, the thread these functions run on will block. This means the main thread of a program will block if the I/O call happens on that thread, see Figure 3.5a5. A synchronous program that performs I/O operations regularly will therefore spend much of its time blocked.



(a) A schematic overview of an I/O operation done by the main thread (synchronous).



(b) A schematic overview of an I/O operation executing on a separate thread, retuning a deferred (asynchronous).

Figure 3.5: A schematic overview of a how an I/O call is handled in a synchronous versus an asynchronous manner.

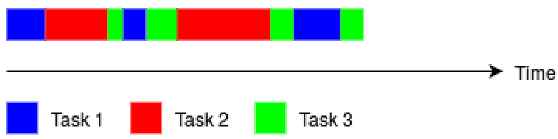


Figure 3.6: Tasks will be executed interleaved when made asynchronous.

In asynchronous programming, tasks are split into multiple chunks and executed interleaved, see Figure 3.6. The fundamental idea behind this approach is that when faced with a task that would normally block waiting for I/O, it will instead execute some other task that can still make progress, saving valuable time. By leveraging

this trait, an asynchronous system can outperform a synchronous program considerably.
Support for asynchrony in Python 2.7 is poor so the use of a framework is required. Tribler makes use of Twisted, an event-driven networking engine written in Python which allows for event-driven and asynchronous programming [35]. In particular, Twisted is built around event-driven, asynchronous I/O [36].
When programming asynchronously in Twisted, the function that is being called (callee) returns a deferred. A deferred is a placeholder for the actual value that the callee eventually will return once its done computing.
To ensure I/O operations do not block the main thread, the I/O can be moved to a separate thread, returning a deferred. By attaching a callback and an errback to this deferred, the caller can handle the case of a successful I/O operation and failure respectively. While this sepearate thread is blocked, the main thread continues executing other scheduled tasks. Once the I/O operation on the separate thread is done, this thread will invoke the callback of the deferred. In case the I/O operation resulted in an error, the thread will invoke the errback. Both the invocations of the callback and errback are scheduled on Twisted's event loop. This event loop can be seen as a queue of tasks which have to be processed by Twisted. The event loop will then schedule the continuation of the task that issued the I/O operation and proceeds executing the current task if not done yet, see Figure 3.5b.

3.4.1 Drawbacks

The main drawback of asynchronous programming is that it makes the structure and execution of a program more complex. As any task can run while another task is blocking, one must be careful that the current task does not modify information the blocking task is dependent on. Especially when working with databases one needs to make sure that the order of database queries is not different.
A second point of attention is the overhead generated when creating multiple threads. As pointed out by the previous sections, offloading work to multiple threads may actually have a negative impact on performance.
Finally, asynchronous programming requires a different mindset as the software is now event-based and makes use of callbacks. This mindset may require new developers joining the Tribler team to study this paradigm, however we do not believe this is a significant drawback.

3.4.2 Initial Measurements

To observe if Tribler can benefit from asynchrony, an experiment has been conducted which replicates Tribler's workload. An overview of the system this experiment was conducted on is available in Table 3.3. As Tribler functions as both a client and a server, CPU, I/O and network tasks are executed interchangeably. To mimic these workloads, the experiment has been set up as follows.

Table 3.3: Specifications of the setup used in the initial CPU/IO/Network experiment.

Component	**Specifications**
Operating System	Ubuntu 15.10
Python version	2.7.11
Processor	Intel Core i7-2630QM
Storage	Samsung 850 EVO 500GB
Memory	8 GB DDR3 1600MHz

Table 3.4: Overview of which workload is processed asynchronously per run configuration.

Configuration	**I/O Asynchronous**	**CPU Asynchronous**	**Network Asynchronous**
0	no	no	no
1	yes	no	no
2	no	no	yes
3	yes	no	yes
4	no	yes	no
5	yes	yes	no
6	no	yes	yes
7	yes	yes	yes

First, six client-server pairs are constructed where each server runs on a separate port. Each client-server pair has its own SQLite database (the same database engine as Tribler and Dispersy) containing two tables: one for the server to fetch data from and one for the client to insert data into. The table for the server to read from is filled with data beforehand.

Next, the client performs ten requests to the server, representing network traffic. Upon receiving the request, the server performs a database query, generating I/O. Unique queries are performed to avoid caching or other optimization techniques employed by the database engine that could influence the results. Once the results have been fetched from the database, the server will perform several calculations on the data and compress the data using zlib[1], creating a CPU-based load. After that, the server sends the compressed data to the client upon which it decompresses and parses said data, generating additional CPU load. Finally, the client will insert the data into the database generating additional I/O.

By measuring the time it takes for all client-serve pairs to complete the requests, we can measure the impact when processing the three different workloads (CPU, I/O and network) synchronously versus asynchronously. To perform these task asynchronously, the Twisted framework has been used at it currently embedded into Tribler. This framework features utilities to offload work to a thread-pool,

[1] `http://www.zlib.net/`

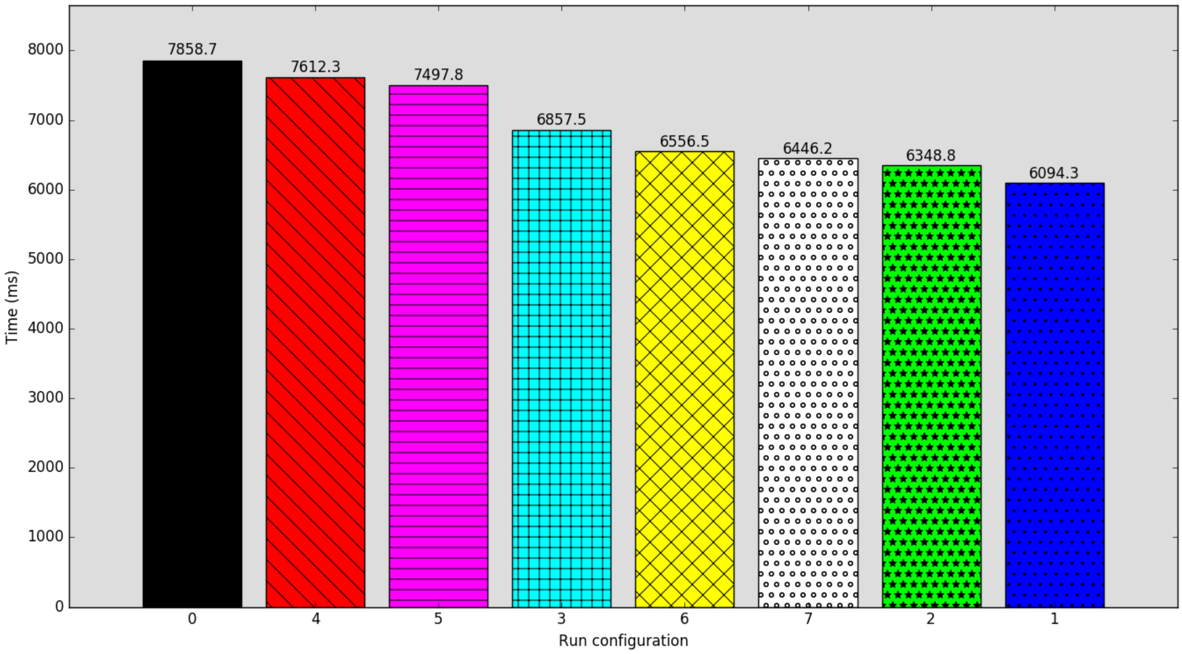


Figure 3.7: Results of the initial experiment.

running the task on a separate thread and taking care of the communication to and from this thread-pool.

In total there are eight possible configurations, defined in Table 3.4. The result of this experiment is visible in Figure 3.7. From this figure we conclude that offloading tasks to separate threads asynchronously yields superior performance over the synchronous case. Solely offloading I/O offers the best result: the time required to perform all requests decreased by 22.5%. As a Solid State Drive (SSD) was used in this experiment, it is possible that the performance gain when using a hard disk with moving parts is bigger as read and write operations are slower on such disks. Offloading all components using asynchronous programming results in a decrease of 18.0% in run time, 4.5% less compared to solely offloading I/O. This can be explained by the fact that offloading all operations to multiple threads will cause more overhead.
From these results we conclude that Tribler can benefit from asynchronous programming especially since I/O is the primary bottleneck.

3.5 Summary

In this chapter we analysed the Python threading model in detail. We explained why parallelism is not possible using standard Python and presented numerous initiatives that attempt to change this behaviour.
We investigated the performance behaviour of multi-threaded programs in Python and explained why multiple threads can cause performance regression. Additionally, we have shown and repeated experiments conducted by David Beazley to confirm that this performance behaviour is still present in Python 2.7.
We shortly elaborated on the the new GIL introduced in Python 3.2, which could be a point of future work to improve Tribler's performance. Next, we presented the use case of asynchrony in the context of I/O in Python and elaborated on its drawbacks. Our initial experiment shows that we can gain performance when using asynchrony, answering the first research question presented in Section 2.5.

Chapter 4

Design & Implementation

After the initial results and discussion of asynchrony in Python, it is evident that asynchronous I/O is beneficial to Tribler's performance and capable of solving our blocking database I/O problem. Since almost all of Tribler's I/O operations are database operations, it is the key point to focus on.
Tribler has Twisted integrated for years, yet Dispersy has not seen any integration despite the decision to do so in 2014 [37]. To ensure a good foundation to build upon without reinventing the wheel, it is key to search for a framework that supports both SQLite (Dispersy's and Tribler's current database system) and asynchronous I/O using Twisted. Using this framework, we create a database manager with a completely asynchronous, non-blocking yet serialized interface using test driven development. Integrating this database manager requires extensive effort, modifying 32% of all files, 23% of all functions and spanning more than six thousand lines of code.

4.1 A New Database Framework

With the recent addition of the MultiChain there are three distinct database files with three distinct database managers in the Tribler code base. None these database managers are fully documented or tested. A proper solution is to replace these three database managers with a new manager featuring an asynchronous and non-

Table 4.1: An overview which features each of the four frameworks support.

	Twistar	**Storm**	**Axiom**	**Alchimia**
Available in the Debian & Ubuntu repositories	✗	✓	✓	✗
Allows 'raw' queries	✓	✓	✓	✓
Allows an ORM approach	✓	✓	✓	✗
Framework is mature	✓	✓	✓	✗

blocking interface. This will result in less code to maintain, all logic in one place and easier to cover with proper unit tests and documentation, yielding increased stability and speed, improved maintainability and enhances the productivity of developers.

After careful scrutiny, four database frameworks that offer integration with Twisted and SQLite were selected: Axiom, Storm, Alchemia and Twistar. Next, they were compared on the possibility to use it as an object-relational mapper (ORM), the possibility to query the database using 'raw' queries, its maturity and the availability in the official repositories of Ubuntu and Debian which is a must as Tribler is published on the official repositories as well. The results of this comparison can be found in Table 4.1.

From this table it is clear that Twister and Alchimia are not good fits; neither of them are available in the official repositories of Ubuntu and Debian. After comparing Axiom and Storm in better detail the final decision led us to choose Storm. The Storm database framework has been developed by Canonical and is featured in several other products such as Launchpad [38], showing its real world usage and maturity. The Storm website features a rich tutorial and documentation section, superior to that of Axiom, where new developers joining Tribler will benefit from. Additionally, all table creation and updates must explicitly be handled by the developer which is Tribler's and Dispersy's current approach. As we favour this enforcement over automatically generated tables, Storm was chosen as the foundation of the new database manager: 'StormDBManager'.

4.2 Designing StormDBManager

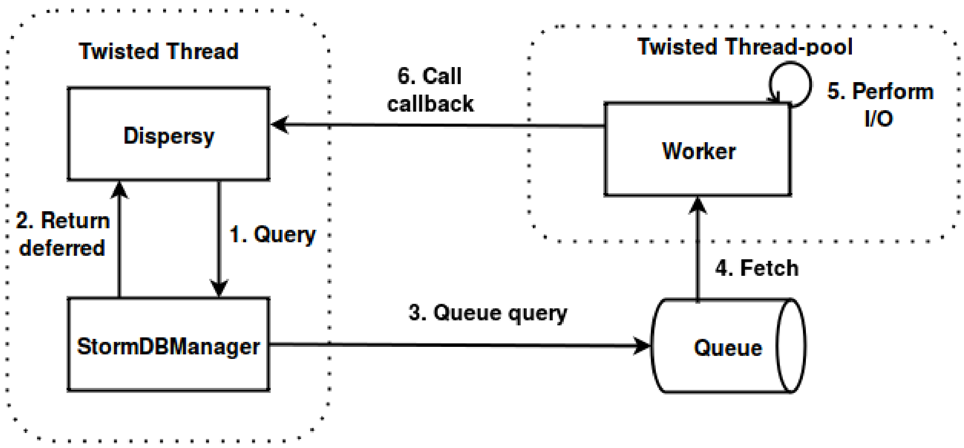


Figure 4.1: An overview of the queueing mechanism of StormDBManager.

StormDBManager features a complete asynchronous, non-blocking yet serialized interface to handle database access. Because Storm also features ORM support,

this database manager can be the foundation for an ORM based approach in the future.
Since multi-threaded support is severely limited using SQLite, we decided to leverage the Twisted thread-pool to allocate a thread for a longer period of time to run a worker on. This worker will be managed by the StormDBManager. Using this approach, all database operations happen on the same thread but outside the Twisted main thread, guaranteeing I/O does not block it. The system works as follows, visualized in Figure 4.1. First, a Dispersy function calls the StormDBManager to run a query (1). The StormDBManager generates a deferred and returns this to the caller (2). Next, the StormDBManager queues a tuple of four elements (3):

1. The function to be called, e.g. execute or fetchone.
2. The arguments to be passed to the function e.g. the query.
3. The keyword arguments to be passed to the function.
4. A deferred to handle the response in an asynchronous way.

By using a thread-safe queue, all calls are scheduled in the same order as required, ensuring serialized behaviour. The worker running on the thread waits blocking for new items to come, preventing the thread from dying. Once a tuple is available it fetches it from the queue (4). It then executes the function (5) and calls the deferred's callback with the result (6). After that, the worker proceeds to wait blocking for a new item, or executes the next tuple if present.

4.2.1 Test Driven Development

To implement StormDBManager, we have used test driven development. In test driven development, the programmer first writes tests which outline the basic functionality of the object to be created, forcing the developer to think about its workings [39]. Next, the developer implements the functions one by one, running the tests after each implemented function to observe if more tests pass. In our case we have implemented eleven tests, containing 23 assertions targeting all functions StormDBManager will offer. After StormDBManager was fully implemented, the tests covered all functions and branches with a total coverage level of 80%. Using this approached turned out to be fruitful: adjusting the implementation of StormDBManager to be more consistent introduced bugs which were captured immediately, preventing long debug sessions.

4.3 Implementing StormDBManager

As the new StormDBManager will start retuning deferreds, functions of Dispersy need to be able to coop with this new paradigm. Every caller of this function will need to be transitively updated as well to handle the deferreds being returned.

```python
def foo(x):
  y = x + 1
  z = bar(y)
  self.variable = z
```

Listing (4.1) Foo synchronous

```python
@inlineCallbacks
def foo(x):
  y = x + 1
  z = yield bar(y)
  self.variable = z
```

Listing (4.2) Foo asynchronous

Example of the same function synchronous and asynchronous.

To keep the amount of changes to a minimum we have made extensive use of the 'inlineCallbacks'[1] decorator. The inlineCallbacks decorator allows programmers to write asynchronous code in a synchronized manner. To illustrate this in an example, consider the two code samples of the same function called 'foo' in Listings 4.1 and 4.2. The left listing shows the synchronous version of foo calling a function 'bar' which for example performs a database query.
After refactoring bar to make use of the 'StormDBManager' it will become asynchronous, returning a deferred. To handle this, we need to update foo to cope with this. Listing 4.2 shows the refactored version of foo; it is decorated with the inlineCallbacks decorator and has now a 'yield' statement in front of the bar function call. Twisted automatically waits for bar's deferred to fire and then continues with the execution.
Consequentially, because of the inlineCallbacks decorator, foo is now an asynchronous function as well, returning a deferred whenever called. As a result all functions that call foo needs to be updated transitively as well.
In total there are 129 function calls to Dispersy's database (excluding tests). By implementing StormDBManager in Dispersy, all functions that host one or more of these 129 database calls needed to be refactored transitively as with the example provided above. After this was done, 414 out of 1784 were modified residing in 52 different files. In total these modifications spanned 4605 lines of additions and 2003 of deletion.
Naturally Tribler also required modifications; in total 106 files required modifications spanning 3572 additions and 1242 deletions to the code base. Finally, an experiment framework called Gumby (see Section 5.1) required minor modifications. In total these modifications resulted in more than eleven thousand modified lines of code applied over a period of six months.

4.3.1 Testing StormDBManager in Dispersy

Unfortunately, unlike the implementation of StormDBManager, its integration into Dispersy is a lot harder to test. As every caller of a function needs to be updated transitively when it becomes asynchronous, a small change can turn into a huge

[1] http://twistedmatrix.com/documents/current/api/twisted.internet.defer.inlineCallbacks.html

refactoring effort. At the same time the unit tests present in Dispersy do not cover the code base very well. This resulted in several bugs which were often hard to find. To counter this situation, some additional tests are written once an uncovered execution path was discovered. Unfortunately, these tests are not enough to cover all non-covered code. Bringing Dispersy to a decent coverage level will require extensive effort and time and is therefore noted as future work.
To test this new version of Dispersy in combination with Tribler, we have made use of the so-called ‘allchannel’ experiment. In this experiment thousand instances of Tribler are created on the DAS5[2] supercomputer which will connect to each other and start synchronizing data. Because of these actions, most functionalities of Dispersy will be invoked, causing most, if not all bugs not captured by the tests to surface. Next, from the data outputted by this experiment we confirm no errors arose and that all data was correctly sent and received by all nodes.

4.4 Summary

In this chapter we have answered the second research question presented in Section 2.5: ‘How do we resolve Tribler’s blocking database I/O problem?’. We have introduced the Storm database framework and the database manager built with said framework: StorDBManager. A major refactoring has been done on Dispersy’s code base to integrate StormDBManager, resulting in all I/O to become asynchronous and non-blocking. To validate no bugs are introduced by this major refactoring, we have made use of the tests present and by running the ‘allchannel’ experiment. Furthermore, because of this intensive refactoring it became apparent that work is needed to improve the test coverage situation in Dispersy. Improving this situation will require extensive effort and is noted as future work.

[2] `http://www.cs.vu.nl/das5/`

Chapter 5

Introducing Software Performance Engineering

In this chapter we look into the topic of introducing software performance engineering (SPE) into Tribler by adding a performance regression testing system into the Tribler development cycle. By looking at the tools already available and prior work, we come up with a system that is both extensible and has minimal impact in Tribler's current architecture. By comparing two version of Tribler, one with the asynchronous I/O rework done presented in Section 4.3 and Tribler's current code base, we show that this performance regression testing system is an asset to Tribler from which many future developers can benefit. Moreover, this performance regression testing system should force developers to keep an eye on performance when making code changes, gradually improving Tribler performance over time.

5.1 Introduction to Gumby

Gumby is an experiment runner framework for Dispersy and Tribler. It is being used by Tribler developers to run experiments on local computers as well as remote servers such as the DAS5 supercomputer. Gumby takes care of most of the steps required in order to run an experiment, including:

- Clearing the output directory at the start of an experiment.
- Syncing workspace directories with remote nodes.
- Run setup scripts concurrently to set up the experiment on all nodes.
- Spawn trackers to monitor the experiment and abort in case errors occur.
- Start both local and remote process instances in parallel at the exact same time.
- Fetch all the data from the output directories residing on remote servers.

- Run post-experiment scripts to generate items such as graphs and tables.

To run an experiment one can specify configurations and scenario files to be executed.
Configuration files specify which experiment to run and define all the settings needed in order to run this experiment using Gumby. These settings often include paths to data, variables such as how many nodes to create and how long the experiment has to run. Once a configuration file has been created, it can be passed to Gumby which takes care of running the experiment.
Scenario files allow for a carefully timed execution of functions. In a scenario file, each line specifies which nodes executes which function at what time. This is extremely useful in order to repeat the exact same procedure many times.
The combination of configuration and scenario files enable a setting in which a developer can run an experiment many times with the exact same execution order and timing. This enables us to perform regression tests using Gumby which we will elaborate on next.

5.2 Performance Regression using Gumby

By using configuration files and scenario files, we can create experiments which run Tribler and perform certain actions such as downloading content. These experiments can in turn be used to compare different versions of Tribler which allows us to test for performance regression. To allow for such comparisons, we decided to extend Gumby by adding experiments or *benchmarks* and additional data processing mechanisms. This procedure works as follows.
First, we run the current version of Tribler or Dispersy i.e. the current code base using a certain benchmark. Next, we run the exact same benchmark on a modified version of Tribler or Dispersy. Gumby will ensure that all functions will be called at the exact same moments, ensuring the execution of both runs is identical. While the experiment is running, Gumby tracks statistics such as memory usage or response times and writes these to a specified output directory. At the end of each experiment, all (relevant) data is fetched from the nodes and combined into a dataset. Using post-experiments scripts we can then compare the two obtained datasets and check if there are changes in e.g. resource usage or metrics.

5.3 Representative Benchmarks

One of the challenges of regression testing using benchmarks is that there should be benchmarks present which feature representative, real-world usage scenarios [40]. While it is intriguing to put a system to the limit of its capabilities and review if it performs better with made modifications, eventually real users should also profit from said modifications.

To create representative benchmarks, both general use cases as well as user-traces can be used. The latter has once been employed by Meulpoler et al. using BarterCast [41]. Using BarterCast, traces were recorded of users to get insight into their usage of Tribler. These traces can be converted into realistic benchmarks. However, as this is outside the scope of this thesis we have devised three benchmarks which feature a more generic yet realistic use-case of Tribler.

5.3.1 Benchmark 1

The first benchmark that we have devised consists of a user having just installed the Tribler client and is about to join the network. This use case is especially important for user satisfaction as first impressions add to this impression [40].

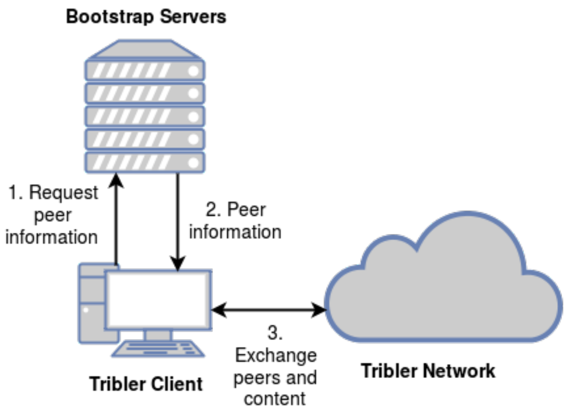


Figure 5.1: The bootstrap procedure of Tribler.

When a new client joins the network it has no knowledge of this network yet besides the location of bootstrap servers which are hard-coded in the Tribler client. To gather information about the network and to start discovering peers and content, the client connects to the bootstrap servers, requesting the locations of peers in the network to exchange data with. After the bootstrap client provides this information, the client will connect to these peers and start exchanging information about content and peers. This process is visualized in Figure 5.1. Over time, this new peer will learn of all content in the network.

By running a Tribler client with no prior knowledge idle for a fixed amount of time, measurements on Tribler's resource usage such as CPU and I/O rates can be made. As Dispersy facilitates Tribler's peer and content discovery, all data will be written to a SQLite database which generates I/O. Meanwhile incoming messages are processed which creates CPU load.

To create this benchmark, an old experiment that runs Tribler idle has been fixed. This experiment was introduced by a Tribler developer to run Tribler for various amounts of time, ranging from 1 hour to 1 week, at set times. Unfortunately it was not updated when changes were made to Tribler. By updating this experiment, it is now functional again and can be run using the latest version of Tribler and Dispersy.

5.3.2 Benchmark 2

The second benchmark represents a user downloading content by opening a magnet link or torrent file using Tribler, closing it as soon as it's finished downloading. This benchmarks represents the free-riding problem: users downloading content without uploading it back to the network i.e. seeding [42]. To mimic this behaviour, we start Tribler and immediately start downloading a file using a magnet link, as if someone opened a magnet link from a third party website. By periodically checking the download status, Tribler can be shutdown when the download is complete.

Using this benchmark, statistics such as CPU usage and download rates can be measured between different versions. This provides insight into Tribler's core functionality: decentralized access to distributed content.

To implement this, the code of the first benchmark was used as a foundation, adding the download, download progress check and shutdown procedure logic to it.

5.3.3 Benchmark 3

In this third benchmark we stress test Tribler by performing requests to the newly introduced application programming interface (API). This API is work that is being done in parallel to this thesis by other members of the Tribler development team where the graphical user interface (GUI) is decoupled from Tribler's core logic. This GUI will instead run in a separate process, communicating through a socket with Tribler's core, allowing Tribler to run headless i.e. without a GUI.

Somewhat related to the first benchmark, the response times of this API are equally important. Hassenzahl et al. show that web designers have 50 milliseconds to make a good first impression [43]. Just like most websites, Tribler's user interface will receive data from an API, rendering response times of this API an important factor. Especially when the GUI has just been opened on a fresh install, various requests will be send to the API requesting channels, torrents, upload and download rates, the amount of free space on the machine, etc. Meanwhile Tribler's core will also be busy connecting to the bootstrap servers as previously discussed. To still deliver good response times, Tribler's core should be as responsive as possible to process incoming requests to this API as fast as possible.

To measure the responsiveness of this API, and indirectly that of Tribler, we have created a benchmark which can perform a varying amount of requests per second to Tribler's API. To perform these requests, we have used Apache JMeter[1]. Apache JMeter was designed to 'simulate a heavy load on a server, group of servers, network or object to test its strength or to analyse overall performance under different

[1] `http://jmeter.apache.org/`

load types.' (jmeter.apache.org, 2016). JMeter tracks the average, maximal en minimal response times and calculates the standard deviation. Additionally, it tracks how many responses are received per second and what the throughput of the API is.

By running this benchmark on two different versions of Tribler, we can observe if the are changes in any of the mentioned statistics captured by JMeter. Since the responsiveness is directly related to the performance of a program, this will provide a good indication.

5.4 Continuous Regression Testing using Jenkins

To include software performance engineering in Tribler's development cycle, it was decided to include regression testing in our Jenkins continuous integration system. Using Jenkins, one can create jobs to run specific tasks. An examples of such a task is to run all unit tests on the current version of Tribler. Additionally, Jenkins can automatically run jobs on defined events such as commits pushed to a pull request or a new pull request being opened on the GitHub repository.

At the start of this thesis, Jenkins only ran the unit tests on a Linux server. To be able to run tests on all major platforms Tribler supports, three additional servers were set up running Windows 32-bit, Windows 64-bit and OS X Yosemite. After these were deployed, a code coverage and pylint checker job were added which track how much of the code has been covered by the tests and if the code adheres the PEP8[2] code style, respectively. When either the code coverage drops or the pylint check fails, the build is marked as failed, safeguarding deterioration of the code base.

To further improve Tribler's development process, we have created a performance regression job that runs the third benchmark using five requests per second. To merge this job into the development cycle, we have prepared it to be added to the 'GH_Tribler_PR' MultiJob. This 'GH_Tribler_PR' MultiJob runs whenever a user pushes a commit to an open pull request, forces a rerun or opens a new pull request on the Tribler GitHub repository. Whenever this performance regression job fails, indicating performance regression has occurred, the 'GH_Tribler_PR' is marked as failed. Upon failure, the author who proposes the changes must then investigate and adjust the code accordingly.

To be able to compare old and new code, we have created a separate job that runs the current code base of Tribler daily using the same experiment. Using this job, we can command Jenkins to fetch the data from this job so we can create a comparison graph and table depicting changes between the output of each job. Once this job is added to the MultiJob, whenever a pull requests gets merged into the code base, this job will automatically run again to ensure comparisons happen against the most recent code.

[2] `https://www.python.org/dev/peps/pep-0008/`

To make sure the build server can process all jobs in parallel without running out of resources, each server has a specific amount of slots defined. In Jenkins, every job that is running takes up a slot on a build server. To avoid allocating too many slots at once, the 'GH_Tribler_PR' MultiJob consists of three phases:

1. The first phase is the testing phase. In this phase tests are run on the four build servers and in parallel the code style is being checked. Because the current code base is under heavy maintenance by other members of the Tribler team, the next phase is always entered to obtain coverage information.

2. The second phase is the code coverage phase. After the tests have run, all code coverage statistics are merged together to obtain the full coverage report. Next, the full report is compared against the full report of the current code base. If the code coverage dropped with this version, the build is marked as failed, otherwise the third and final phase is entered.

3. The third phase is the experimental phase. In this phase an experiment called 'Allchannel+ChannelCommunity_short' is run which checks if Dispersy can still synchronize data properly.

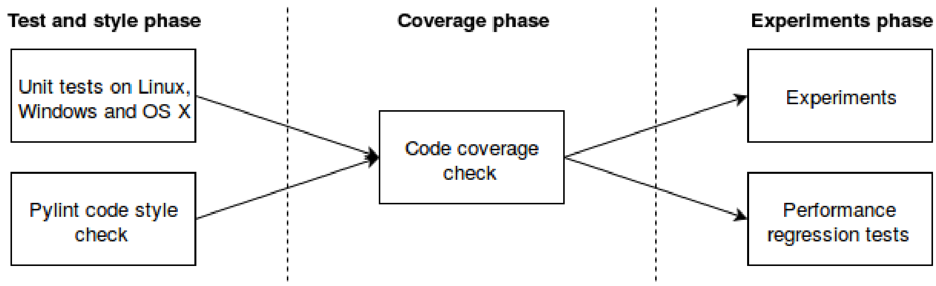


Figure 5.2: The three stages of testing

As our benchmark can be categorized as experiment, we decided to add our job to the third phase. Figure 5.2 visualizes the three phases of the 'GH_Tribler_PR' MultiJob.

5.4.1 Multiple Platforms

As each of these operating systems could influence performance [44], it is evident regression testing should be performed on these operating systems. Currently, the build servers are used to package Tribler for official releases and to run unit tests on. Soon a build server running the Android operating system will be added as there is an official Android Tribler app in development.
Using Jenkins, we can schedule benchmarks automatically on all available servers, visualized in Figure 5.3. By doing so, we can observe if Tribler performs the same on all operating systems. In case a specific operation system does not match the

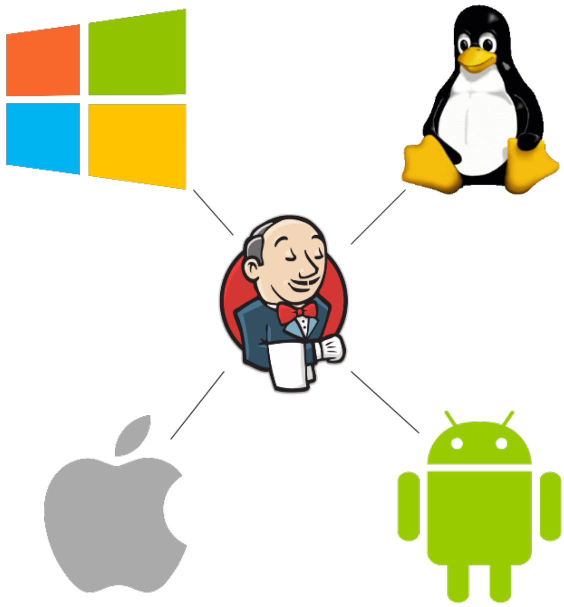


Figure 5.3: Jenkins running on four major operating systems.

signature of the others, developers can look into the inner workers of this operating system to better understand the situation. As this item was not feasible to implement due to time constraints, it is noted down as future work.

5.5 Summary

In this chapter we have introduced Gumby and elaborated on how we extended Gumby to create a regression testing system. Furthermore, we elaborated on the need of representative benchmarks and provided three benchmarks which represent possible real-world use cases of Tribler. Next, we depicted the current testing infrastructure and elaborated on how to integrate this regression testing system into our continuous integration system Jenkins, answering the third research question 'How do we incorporate software performance engineering into Tribler's development process to gain insight into performance statistics?'. Finally, we have marked the option to employ this regression testing system on all platforms Tribler supports as future work.

Chapter 6

Experimental results

Table 6.1: Specifications of the virtual private server used for experiments.

Component	**Specifications**
Operating System	Ubuntu 15.10
Python version	2.7.10
Processor	Intel Xeon E5-2450 v2 @2.50GHz
Storage	500 GB SSD Raid0
Memory	8 GB 1000 MHz

To evaluate Tribler's current situation and to validate our implementations are working correctly, experiments have been conducted. In this chapter we elaborate on these experiments and discuss the results. Section 6.1 and 6.2 present experiments done on past and the current version of Tribler. Section 6.3 provides experimental results on comparing the current version of Dispersy with our work. Section 6.4 shows that the output of the regression testing system can be quickly assessed by developers to observe changes in performance.
All experiments with the exception of one have been conduction on a virtual private server whose specifications can be found in Table 6.1. This virtual private server is connected to a 100Mbit Internet connection.

6.1 Tribler's I/O Throughout the Years

As explained in Section 2.2, Tribler's I/O has been a problem for years. To observe if and how the amount of I/O has changed over time, we have performed a quick assessment on four different versions of Tribler using iotop[1]. The versions and their release dates are shown in Table 6.2.
As these measurements were never performed systematically, it is vital to perform these measurement now to observe if any changes have occurred between releases

[1]`http://guichaz.free.fr/iotop/`

and document them. This will provide valuable insight in Tribler's behaviour and the extent of the problem. Moreover it will show us if the amount of I/O is being reduced since the creation of the ticket on GitHub.

Table 6.2: The four versions of Tribler and their release dates.

Tribler version	**Release date**
6.3.5	2014-11-06
6.4.3	2015-01-21
6.5.2	2016-05-13
6.6.0-exp1	2016-07-26

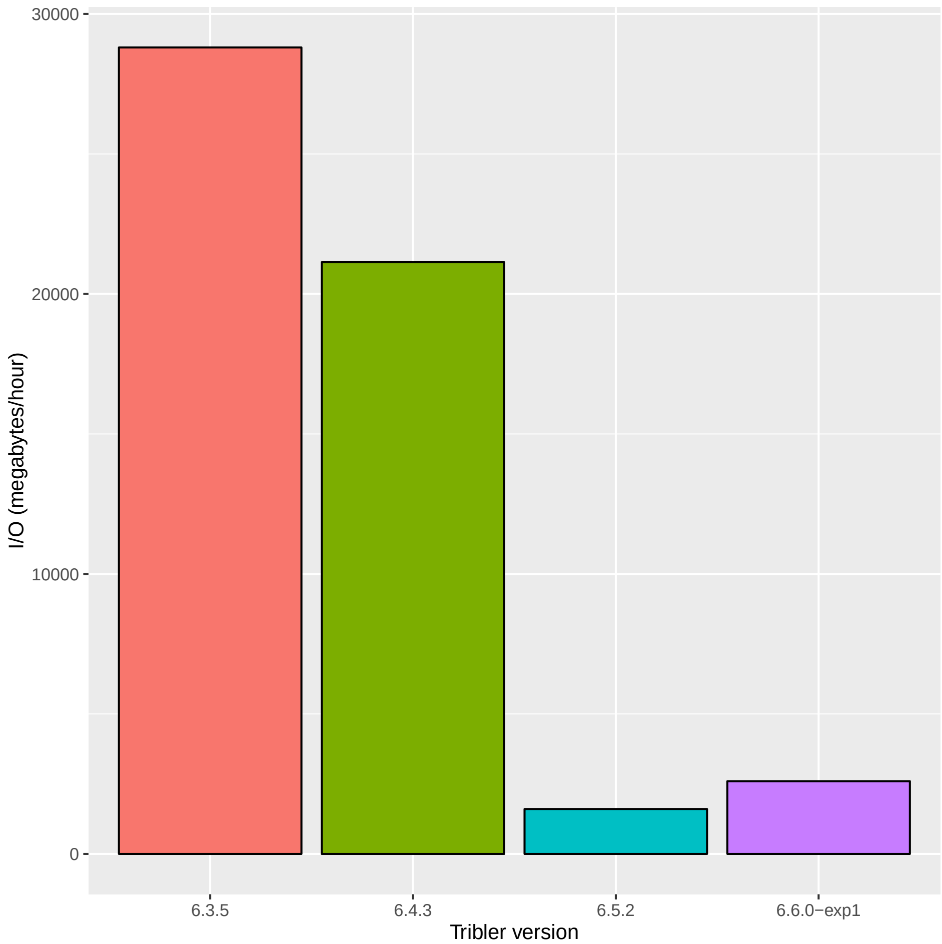


Figure 6.1: The amount of I/O per version of Tribler.

Each version of Tribler will run for one hour idle, using a clean state directory i.e. no prior knowledge of the network and its contents. A state directory is a directory where Tribler stores all configuration files, databases, information about the

Table 6.3: The packets collected for each version by Tribler when running idle for one hour using a clean state directory.

Packet type	Tribler version	Amount
Dispersy-identity: a message containing the public key of another peer for identification purposes.	6.3.5	20,026
	6.4.3	38,268
	6.5.2	46,890
	6.6.0-exp1	41,480
Dispersy-undo-own: a message that allows a member to undo his/her own messages and flags them as undone.	6.3.5	40,594
	6.4.3	45,095
	6.5.2	47,409
	6.6.0-exp1	44,279
Votecast: a message indicating a vote has been given by a user on a channel, indicating it is spam or not.	6.3.5	44,990
	6.4.3	90,339
	6.5.2	116,557
	6.6.0-exp1	96,509

network and other data such as checkpoints. During the idle run, Tribler will start discovering peers and content such as channels and torrents, storing the obtained data in the database which gets flushed to disk. Additionally, peers will start requesting data from this Tribler instance such as search and peer exchange requests, causing Tribler to read data from its database. These read and write operations will be monitored by iotop and the amounts automatically accumulated. After one hour, the amount of read and write I/O is noted down and Tribler is shutdown.

The results of this experiment are visible in Figure 6.1. From this figure we observe that Tribler's I/O has been reduced significantly in version 6.5.2. This decrease was the result of batching multiple database queries and periodically flush them to disk, an effective optimization technique also applied in other work [45, 46]. Furthermore we observe the amount of I/O is increasing again in the latest 6.6.0-exp1 release possibly due to the MultiChain feature added.

Peculiarly, the numbers observed in this experiment are significantly higher than the numbers reported in the original ticket. We believe there are two reasons that contribute to these numbers. The first reason being to the 100 Mbit connection of the experiment machine, providing excellent connectability conditions. The second reason is the fact that this instance began with a clean state directory while running idle. This provides ideal conditions for Tribler to spend most of its resources on discovering peers and content.

To confirm that the decrease in I/O is not due to a decrease in packets received, we have tracked the amount of packets Tribler managed to synchronize while running one hour idle using a clean state directory for each version. The results are visible in Table 6.3. From this table we conclude that while version 6.3.5 and 6.4.3 perform much more I/O, the amount of packets obtained is significantly less compared to version 6.5.2. From these numbers we this conclude that the cost of having high

I/O rates is not justified by the benefits as they perform even worse. Interestingly, the numbers of the latest 6.6.0-exp1 version are similar to that of 6.4.3, while we cannot explain this fully, we believe it may be caused by external network conditions or the MultiChain feature.
In conclusion, we believe that the I/O rate of Tribler does not show a correlation with the amount of packets synchronized. As the I/O rate of Tribler rises again in the latest version, possibly because of the MultiChain feature requiring additional computations, it shows the urgency of the I/O to become asynchronous and non-blocking.

6.2 Identifying Hotspots in Tribler's Database Usage

Table 6.4: The breakdown of Tribler's database queries when running Tribler idle for one hour using an empty state directory.

Query	**Amount of calls**	**Total time (s)**	**Max**	**Average**	**Min**
fetchall	91661	278.319	0.23420	0.00304	0.00000
execute	218818	16.142	0.21455	0.00007	0.00000
executemany	22558	10.414	0.06059	0.00046	0.00000
update	6647	22.542	0.22119	0.00339	0.00227
insert_or_ignore	18511	50.580	0.16437	0.00273	0.00000
fetchone	87904	273.999	0.21972	0.00312	0.00196

Table 6.5: The breakdown of Dispersy's database queries when running Tribler idle for one hour using an empty state directory.

Query	**Amount of calls**	**Total time (s)**	**Max**	**Average**	**Min**
commit	69	2.758	0.21891	0.03998	0.00000
execute	741633	29.300	0.14197	0.00004	0.00000
executemany	6781	0.309	0.01387	0.00005	0.00001
executescript	3	0.000	0.00001	0.00001	0.00001

To observe the impact of each query separately and identify hotspots, an infrastructure was created that automatically provides insight into the most expensive, most executed, and longest duration queries. With more than one million queries per hour, Tribler developers desperately need such a tool. Developers can use this information to directly dive into the relevant source code and test possible improvements. To get insight into Tribler's current query statistics, we have run two versions of Tribler for one hour using an empty state directly and a filled state directory. The filled state directory in our experiment contains meta information on 100,000 torrents and roughly 1200 channels on which ten have been subscribed. Such a filled state directory is quite common for regular Tribler users as they dis-

Table 6.6: The breakdown of Tribler's database queries when running Tribler idle for one hour using a filled state directory.

Query	Amount of calls	Total time (s)	Max	Average	Min
fetchall	36815	318.498	0.46183	0.00865	0.00000
execute	119773	190.340	1.64548	0.00159	0.00000
executemany	20691	2.953	0.02570	0.00014	0.00001
update	8962	26.787	0.02191	0.00299	0.00221
insert_or_ignore	692	1.692	0.00483	0.00244	0.00178
fetchone	55090	171.909	0.96909	0.00312	0.00000

Table 6.7: The breakdown of Dispersy's database queries when running Tribler idle for one hour using a filled state directory.

Query	Amount of calls	Total time (s)	Max	Average	Min
commit	68	24.513	1.65048	0.36048	0.00000
execute	49117	149.413	0.55947	0.00304	0.00000
executemany	164	0.004	0.00009	0.00002	0.00002

cover the network and its contents. By subscribing to ten channels, Tribler will attempt to discover all content within these channels.

For each of the two runs, we have created breakdowns depicting the amount of queries issued on Tribler's and Dispersy's database. The breakdowns for Tribler's database using an empty respectively filled state directory can be found in Table 6.4 and Table 6.6. In Table 6.5 and Table 6.7 we show the breakdown of Dispersy's database using an empty and filled state directory, respectively.

From these breakdowns we observe that Tribler's database I/O has an enormous impact on Tribler's performance, in contrast what was assumed by Tribler developers (see Section 2.2). In the experiment with an empty state directory, Dispersy generated 32.368 seconds worth of I/O using 748,486 queries in total. Tribler's database was queried 446,099 times, generating 651.997 seconds of I/O, a *twenty fold increase* compared to Dispersy. In total 684,365 out of 3600 seconds are spend on I/O, meaning 19% of the time the main thread is blocked.

The situation worsens when Tribler starts filling its state directory. Using the filled state directory, Dispersy generated 173.929 seconds worth of I/O using 49,349 queries in total. Tribler's database was queried 242,023 times, generating 712.180 seconds of I/O. The amount of queries is significantly lower as Dispersy now has discovered most of the network, resulting in less new peers to be added. Additionally, less new content is discovered, reducing the amounts of operations on Tribler's database. In general, we observe that the queries are becoming more time consuming, which can be explained by the database having more data to search through. In total, 886,109 seconds out of 3600 seconds are spend on I/O, meaning 24.6% of the time the main thread is blocked.

To gain insight into which functions issue the most expensive queries, we have created a breakdown of the callers for each database function. This breakdown lists the five callers that issued queries consuming the most time. By providing the name, file path and line number in this breakdown, developers can immediately jump to the right point in the code base, increasing the developer's productivity. These caller breakdowns for Tribler's database when running Tribler with an empty and filled state directory are provided Table 6.8 and Table 6.9, respectively. Table 6.10 and Table 6.11 show the respective breakdowns for Dispersy's database using an empty and filled state directory.

Table 6.8: The five most expensive query issuers per database function called on Tribler's database when running Tribler idle for one hour using an empty state directory.

DB. function	Duration (ms)	Caller	Caller location
fetchall	234	getRecentAndRandomTorrents	...B/SqliteCacheDBHandler.py line 1846
	208	getPeerIDS	...B/SqliteCacheDBHandler.py line 101
	205	getPeerIDS	...B/SqliteCacheDBHandler.py line 101
	195	getPeerIDS	...B/SqliteCacheDBHandler.py line 101
	183	getPeerIDS	...B/SqliteCacheDBHandler.py line 101
execute	214	commit_now	.../CacheDB/sqlitecachedb.py line 228
	198	commit_now	.../CacheDB/sqlitecachedb.py line 228
	193	commit_now	.../CacheDB/sqlitecachedb.py line 228
	182	commit_now	.../CacheDB/sqlitecachedb.py line 228
	165	commit_now	.../CacheDB/sqlitecachedb.py line 228
executemany	60	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	55	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	52	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	50	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	49	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
update	221	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	23	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	22	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	21	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	21	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
insert_or_ignore	164	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	19	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	18	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	18	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	17	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
fetchone	219	getChannelIdFromDispersyCID	...B/SqliteCacheDBHandler.py line 1367
	172	getOne	.../CacheDB/sqlitecachedb.py line 490
	157	getOne	.../CacheDB/sqlitecachedb.py line 490
	141	getChannelIdFromDispersyCID	...B/SqliteCacheDBHandler.py line 1367
	123	getChannelIdFromDispersyCID	...B/SqliteCacheDBHandler.py line 1367

In addition to creating a breakdown of the most expensive queries, we also investigated which lines contribute the most to the total amount of time spent on I/O. By creating this breakdown we are able to identify queries that cause a lot of I/O, yet may not show up in the breakdowns above because of their low run time. For each of the four runs mentioned above, we compute the top five most I/O time consuming lines and calculate their contribution in percentages, see Table 6.12.
From this table we observe that the top five lines are responsible for at least 59.9%

Table 6.9: The five most expensive query issuers per database function called on Tribler's database when running Tribler idle for one hour using a filled state directory.

DB. function	Duration (ms)	Caller	Caller location
fetchall	461	_flush_to_database	...B/SqliteCacheDBHandler.py line 1206
	437	_flush_to_database	...B/SqliteCacheDBHandler.py line 1206
	436	_flush_to_database	...B/SqliteCacheDBHandler.py line 1206
	436	_flush_to_database	...B/SqliteCacheDBHandler.py line 1206
	436	_flush_to_database	...B/SqliteCacheDBHandler.py line 1206
execute	1645	commit_now	.../CacheDB/sqlitecachedb.py line 228
	1588	commit_now	.../CacheDB/sqlitecachedb.py line 228
	1465	commit_now	.../CacheDB/sqlitecachedb.py line 228
	1434	commit_now	.../CacheDB/sqlitecachedb.py line 228
	1188	commit_now	.../CacheDB/sqlitecachedb.py line 228
executemany	25	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	18	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	18	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	17	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
	15	on_remove_votes_from_dispersy	...B/SqliteCacheDBHandler.py line 1188
update	21	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	19	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	17	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	14	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
	12	addExternalTorrentNoDef	...B/SqliteCacheDBHandler.py line 338
insert_or_ignore	4	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	4	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	3	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	3	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
	3	addOrGetPeerID	...B/SqliteCacheDBHandler.py line 116
fetchone	969	hasTorrents	...B/SqliteCacheDBHandler.py line 1516
	212	getOne	.../CacheDB/sqlitecachedb.py line 490
	82	getOne	.../CacheDB/sqlitecachedb.py line 490
	79	hasTorrents	...B/SqliteCacheDBHandler.py line 1516
	73	getOne	.../CacheDB/sqlitecachedb.py line 490

of all database I/O time for any of the four runs. Focusing on reducing these queries by applying query optimization may reduce the run times significantly [47].
From the results provided above we conclude what was long a hunch: Tribler is I/O bound and in desperate need of a non-blocking database I/O solution. By creating an infrastructure which provides a breakdown of query execution statistics, we have solved the Tribler team latent need: insight into Tribler's database usage. Using this infrastructure, developers can now focus on the queries which require optimization to adhere to the 80/20 rule: targeting the vital few, not the trivial many.

Table 6.10: The five most expensive query issuers per database function called on Dispersy's database when running Tribler idle for one hour using an empty state directory.

DB. function	Duration (ms)	Caller	Caller location
commit	218	_flush_database	...bler/dispersy/dispersy.py line 2089
	204	_flush_database	...bler/dispersy/dispersy.py line 2089
	198	_flush_database	...bler/dispersy/dispersy.py line 2089
	186	_flush_database	...bler/dispersy/dispersy.py line 2089
	169	_flush_database	...bler/dispersy/dispersy.py line 2089
execute	141	get_member	...ler/dispersy/community.py line 1860
	16	check_undo	...ler/dispersy/community.py line 3370
	13	initialize	...ler/dispersy/community.py line 388
	12	get_member	...bler/dispersy/dispersy.py line 492
	11	get_member	...bler/dispersy/dispersy.py line 492
executemany	13	on_undo	...ler/dispersy/community.py line 3476
	9	initialize	...ler/dispersy/community.py line 374
	3	initialize	...ler/dispersy/community.py line 374
	1	initialize	...ler/dispersy/community.py line 374
	1	initialize	...ler/dispersy/community.py line 374
executescript	0	check_database	...persy/dispersydatabase.py line 83
	0	check_database	...ty/multichain/database.py line 297
	0	check_database	...ty/multichain/database.py line 298

Table 6.11: The five most expensive query issuers per database function called on Dispersy's database when running Tribler idle for one hour using a filled state directory.

DB. function	Duration (ms)	Caller	Caller location
commit	1650	_flush_database	...bler/dispersy/dispersy.py line 2089
	1592	_flush_database	...bler/dispersy/dispersy.py line 2089
	1470	_flush_database	...bler/dispersy/dispersy.py line 2089
	1440	_flush_database	...bler/dispersy/dispersy.py line 2089
	1194	_flush_database	...bler/dispersy/dispersy.py line 2089
execute	559	_select_and_fix	...ler/dispersy/community.py line 888
	548	_select_and_fix	...ler/dispersy/community.py line 888
	545	_select_and_fix	...ler/dispersy/community.py line 888
	542	_select_and_fix	...ler/dispersy/community.py line 888
	527	_select_and_fix	...ler/dispersy/community.py line 888
executemany	0	on_undo	...ler/dispersy/community.py line 3476
	0	initialize	...ler/dispersy/community.py line 374
	0	initialize	...ler/dispersy/community.py line 374
	0	initialize	...ler/dispersy/community.py line 374
	0	initialize	...ler/dispersy/community.py line 374

Table 6.12: The top five lines contributing the most to the I/O time for each of the four runs.

Tribler's database, empty state directory	
Line	**Contribution**
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 101	26.1%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 1367	24.4%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 262	8.0%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 116	7.8%
Tribler/Core/CacheDB/sqlitecachedb.py line 490	7.0%
Tribler's database, filled state directory	
Line	**Contribution**
Tribler/Core/CacheDB/sqlitecachedb.py line 335	22.9%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 1831	10.6%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 1820	10.1%
Tribler/Core/CacheDB/SqliteCacheDBHandler.py line 262	8.3%
Tribler/Core/CacheDB/sqlitecachedb.py line 490	8.0%
Dispersy's database, empty state directory	
Line	**Contribution**
Tribler/dispersy/dispersy.py line 1550	18.7%
Tribler/dispersy/dispersy.py line 888	13.9%
Tribler/dispersy/dispersy.py line 978	11.9%
Tribler/dispersy/dispersy.py line 492	9.6%
Tribler/dispersy/community.py line 3370	8.9%
Dispersy's database, filled state directory	
Line	**Contribution**
Tribler/dispersy/community.py line 888	49.8%
Tribler/dispersy/community.py line 885	29.7%
Tribler/dispersy/dispersy.py line 2089	13.6%
Tribler/dispersy/community.py line 914	5.0%
Tribler/dispersy/database.py line 109	0.5%

6.3 Measuring Performance Improvements

To measure the performance gain of Tribler using our work, we have conducted two sets of experiments where two instances of Tribler, one with the current synchronous, blocking Dispersy implementation and one with our asynchronous, non-blocking version of Dispersy, are compared.
In the first experiment we have ran Tribler idle for one hour while querying the Twisted event loop every 100 milliseconds for delayed calls. By doing so we can observe if certain tasks have been delayed past their set time of execution i.e. measure the latency in the system.
In the second experiment we have stress-tested Tribler's new API. By requesting data from an endpoint at several rates, we can observe the response times, variance in these response times and throughput of Tribler.

6.3.1 Measuring the Latency of Tribler

Table 6.13: Specifications of the setup used in the latency experiment.

Component	**Specification**
Operating System	Ubuntu 16.04 LTS
Python version	2.7.12
Processor	Intel Core i5-2410M
Storage	Samsung 850 EVO 250GB
Memory	8 GB DDR3 1600MHz

In this experiment we have compared two versions of Tribler, one with Dispersy having blocking, synchronous I/O and one with Dispersy running StormDBManager and thus having non-blocking, asynchronous I/O. This experiment was run on a machine whose specifications can be found in Table 6.13. Each instance of Tribler was run one hour idle where every 100 milliseconds the event loop of Twisted was queried for delayed calls. By observing if scheduled calls are past their set time of execution, we can measure the amount of delay or *latency* in the system. Latency occurs when the twisted reactor thread is blocked or busy with a task that takes a relative long time to complete, causing other tasks to become delayed. Latency is therefore directly related to the responsiveness of a program. The lower the latency, the more responsive a system is.
In theory, making functions asynchronous slices them into smaller 'chunks' which can be executed interleaved, creating a more responsive system as e.g. user actions will be executed in between (background) operations.
In this experiment the hypothesis is that the latency of the asynchronous version will be lower than its synchronous counterpart. Since Tribler is running idle it has more resources i.e. CPU time available to tend to Dispersy which is running in the background, which most likely will cause the difference to be smaller than when

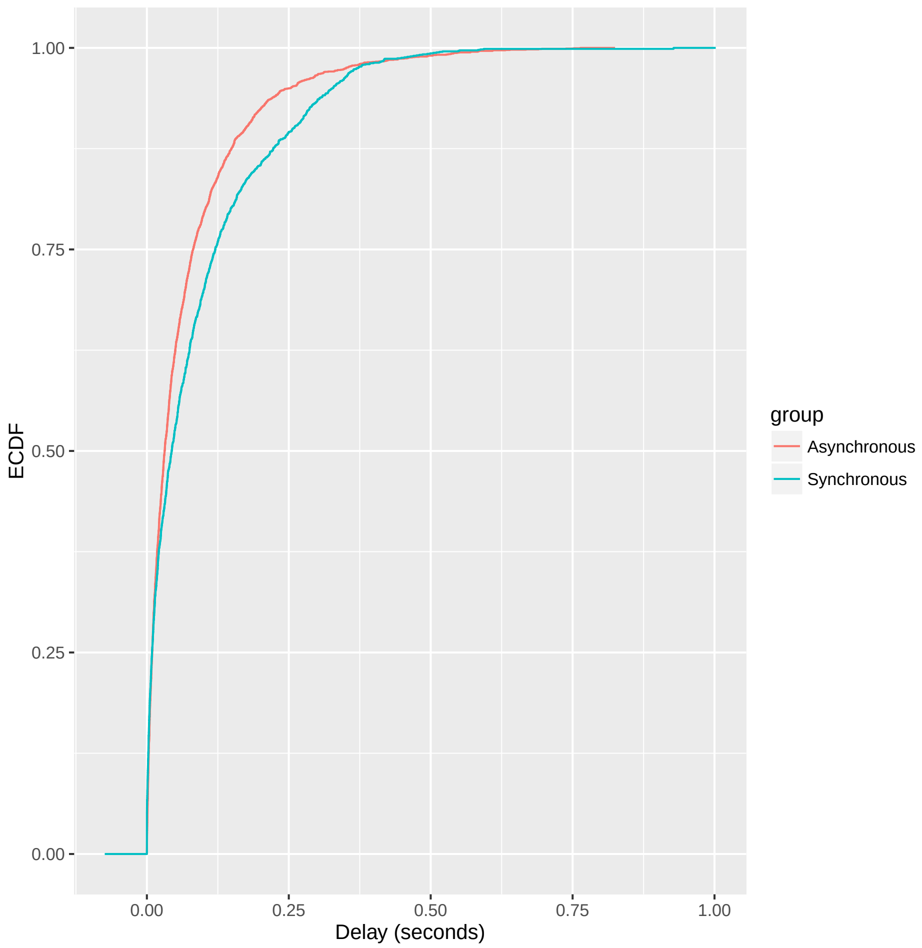


Figure 6.2: The ECDF plot showing the distribution of delay (or latency) of functions in Tribler when running idle for one hour using Dispersy with blocking, synchronous I/O and running Dispersy with non-blocking, asynchronous I/O.

Tribler is experiencing additional load. However, we believe the difference on average may still be significant enough to prefer the asynchronous implementation.

After running the experiment, we have created an empirical cumulative distribution function (ECDF) plot of the delayed calls, visible in Figure 6.2. From this figure we observe that the asynchronous version performs better than the synchronous version. On average, the synchronous version has a latency of 87 milliseconds, where the asynchronous version has a latency of 66 milliseconds, a reduction of 24%. Furthermore, the synchronous case has more outliers, some even touching the one second mark. This confirms the hypothesis: both on average and in maxima the latency of the asynchronous version are lower.

6.3.2 Measuring the Responsiveness of Tribler

To measure the responsiveness of Tribler while under load, we have stress tested the API of Tribler using the procedure described in Section 5.3.3.
In this experiment we use the filled state directory mentioned in Section 6.2. By querying Tribler's channel API endpoint for all discovered channels, all channels in the database will be fetched and returned. As this is Tribler's heaviest endpoint in terms of computation, it's the best way to put Tribler under load. In total the experiment will be run six times, querying the channel endpoint exactly 1000 times per run using 1, 2, 5, 10, 15 and 20 requests per second respectively. By tracking the response times and the throughput (T) the API can offer, we can measure the gain in responsiveness and thus in performance of Tribler. We expect that the asynchronous version will outperform the synchronous version significantly in both response times and throughput.

Table 6.14: The results of the six experiments runs with and without asynchronous, non-blocking I/O in Dispersy.

Req./s	Async.	Avg (ms)	Min	Max	Std. Dev.	T (KB/s)	Resp./s
1	no	315	53	4774	615.90	564.19	0.9
2	no	237	52	4524	475.31	1049.31	1.7
5	no	143	52	2865	259.64	2399.57	3.9
10	no	135	54	3338	259.37	3827.02	6.2
15	no	133	51	4678	382.57	4521.46	7.3
20	no	109	51	3400	239.23	5599.75	9.1
1	yes	134	57	2576	189.89	612.60	1.0
2	yes	113	56	1224	162.62	1197.90	1.9
5	yes	67	56	846	38.50	3058.27	5.0
10	yes	89	52	1138	92.58	5435.71	8.8
15	yes	88	52	963	82.23	6799.35	11.0
20	yes	74	52	1051	57.37	8264.01	13.4

The results of the experiment can be found in Table 6.14. From this table we observe that asynchronous version has a significant less amount of response time, both on average and in maxima. The reduction in response times (on average) ranges between 32.1% and 57.5%. As the responsiveness of a program can be directly linked to its performance (see Section 5.3.3), this looks very promising. Interesting to note here is that the average response times of the synchronous version go down when the amount of requests per second goes up. We believe this may be due to Twisted caching responses.
Another indication that the system has become more responsive is the the standard deviation. For every run the standard deviation of the asynchronous version is significantly less than its synchronous counterpart, indicating the response times are more stable. This can be explained by the slicing of tasks because of asynchrony;

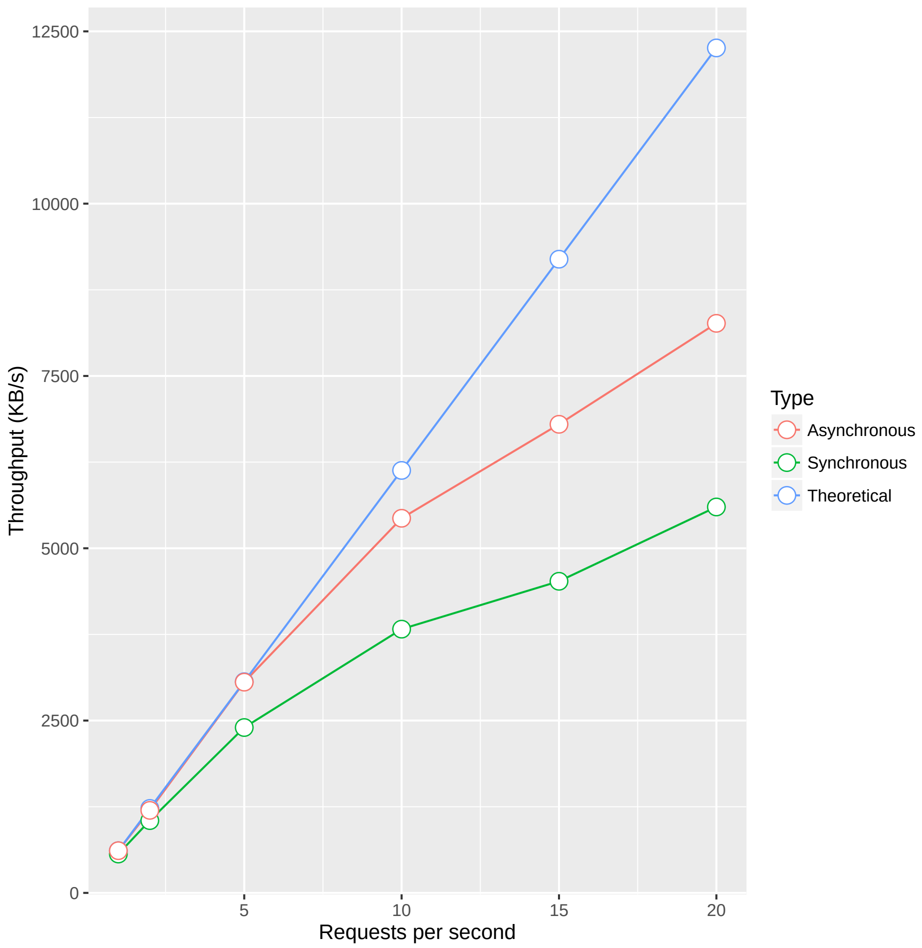


Figure 6.3: The throughput theoretically and of Tribler running Dispersy with asynchronous, non-blocking and synchronous, blocking I/O

as tasks are more interleaved, smaller tasks such as a request will be processed in between bigger tasks, yielding a higher and more stable responsivity.

A third promising statistic is the throughput. As the response is 613 kilobytes (KB) in size, the theoretical maximum throughput will be 613, 1226, 3065, 6130, 9195 and 12260 KB/s for 1, 2, 5, 10, 15 and 20 requests per second, respectively. If we plot the theoretical, asynchronous and the synchronous throughput we obtain Figure 6.3. As we can see the throughput of the asynchronous case lies close to the theoretical maximum until around the ten requests per second. At this point Tribler starts to show signs of being overloaded, which is also visible in Table 6.14 when looking at the amount of responses received per second.

If we look at the responses per second for fifteen and twenty requests per second, we observe that the gap between requests and responses grows percentage wise. The question that arises here is why Tribler can't provide 13.4 responses per second

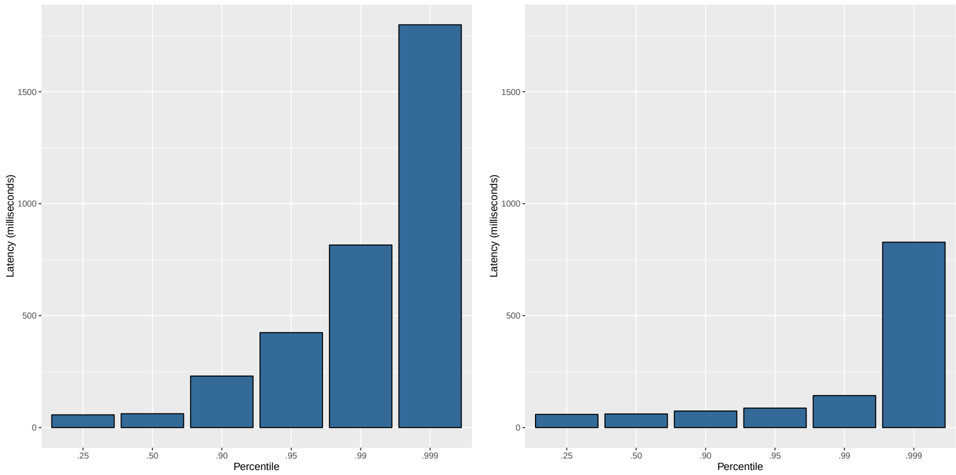
with fifteen requests per second as in the case with twenty. Again we believe the answer lies in the Twisted framework.
As more tasks are scheduled on the event loop of Twisted, it will process each of them fairly where priority is given to the most delayed task. Since there are now more requests pending, it will spend more computation power on the requests. Even though this means that more responses per second can be provided, percentage wise the amount of replies per request drops: for fifteen requests this percentage is 73% where for twenty requests per second this percentage is 67%.
All in all, this experiment demonstrates that the asynchronous system has superior performance over the synchronous case, increasing the throughput by up to 150% and reducing response times by up to 57.5%.

6.3.3 Longtail Latencies

Another important measurement that can be made is at which percentiles of requests high latencies occur. These high latencies often occur at high percentiles, referred to as *longtail latencies*. If a longtail latency is caused by an I/O operation, it may be an indication that the system blocks for that duration. As Hsu explains 'Longtail latencies affect members every day and improving the response times of systems even at the 99th percentile is critical to the member's experience.', (Hsu, 2015) [48]. Longtail latencies are often orders of magnitude larger than the median latency. Moreover, they frequently go unnoticed when measuring the average or taking the median. Hsu explains that longtail latencies impact users quite regularly even when the chance of one occurring is low. If a longtail latency occurs with a probability ρ and a user performs n requests, the probability that a user will observe a longtail latency is

$$1 - (1 - \rho)^n \tag{6.1}$$

If we take $\rho = 0.01$ and $n = 20$, the probability of a user observing a longtail latency response will be 18.2%, a significant chance. As the new API introduced in Tribler will be queried frequently, it is apparent that longtail latencies are also of importance to Tribler.
To observe if the longtail latency problem exists in Tribler and if it has been reduced by our work, we have inspected the data generated by the experiment run conducted in Section 6.4. The results are visible in Figure 6.4. From these two figures we observe that the existing code has a lot of long tail latencies, starting at the 90th percentile. Our work starts showing long tail latencies from the 99.9th percentile, which is a huge improvement over the existing code. However, even though we have reduced the 99.9th percentile from 1799 milliseconds in the existing code to 828 milliseconds in our work, it still indicates that work can be done to reduce this number. As Hsu indicates, finding the root cause of these long tail percentiles can be hard and due to time constraints it is not feasible for this work to resolve. It is therefore marked as future work.



(a) The latencies per percentile when running Dispersy with existing code.

(b) The latencies per percentile when running Dispersy with our work.

Figure 6.4: The response times per percentile when running Dispersy with the existing code and our work.

6.4 Validating the Performance Regression Testing System

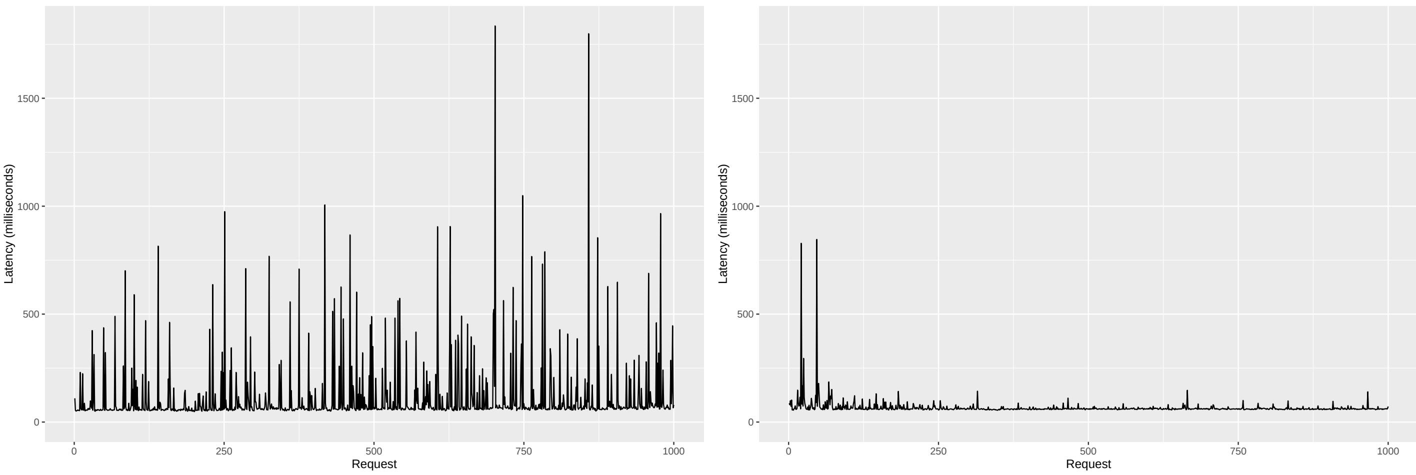


Figure 6.5: The comparison graph showing the response times of Tribler's API. Left: Tribler running Dispersy with synchronous, blocking I/O, right: Tribler running Dispersy with asynchronous, non-blocking I/O.

To supply developers with a quick overview of the changes in the metrics tracked by Gumby, we have created a regression testing system as explained in Chapter 5.

To validate this overview provides an instant, clear overview without going in-depth, we run the experiment described in Section 6.3.2 again with five requests per second. This experiment captures the response times, variance in said response times, throughput and amount of responses received. From the results, our regression testing system creates a side-by-side comparison graph and a table with an overview of the differences in the data obtained.
Figure 6.5 shows the comparison graph generated. From this figure, it is clear that the left hand side – showing the existing code – has higher response times than the right side (our work). It is immediately clear that the proposed changes have a positive impact on the responsiveness of the API.

Metric	Current code base	Modified code base	Change (%)
Avg. response time	143	67	-53,1%
Max. response time	2865	846	-70.5%
Min. response time	52	56	7.7%
Standard deviation	259.64	38.50	-82.5%
Throughput	2399.57	3058.27	27.5%
Responses per sec	3.9	5.0	28.2%

Figure 6.6: The generated table depicting changes between the current and modified code base.

To look at the data generated by the benchmark in more detail, developers can look at the table generated by the regression testing system, an example of this table is visible in Figure 6.6. This breakdown presents the average, minimum, maximum and standard deviation in response times as well as the throughput and amount of responses per second achieved.
From these figures we observe that the performance regression testing system provides a clear overview for developers to get a quick overview of the changes.

6.5 Summary

From the results provided in this chapter we can conclude that Tribler is I/O bound. By implementing a non-blocking database solution we have reduced Tribler's API response times by up to 57.5%, improved the throughput by up to 150% and reduced the long tail latencies considerably. Furthermore we have finally gained insight into Tribler's database usage. By providing a breakdown per database function and on the level of source code lines, we can now accurately pinpoint which lines are responsible for the most I/O time and which database queries might need optimization. Finally, we have shown that our regression testing system performs well and once integrated into Tribler's development cycle, will provide detailed metrics and insights, allowing developer to pinpoint bottlenecks faster and more accurate.

Unfortunately, due to time constraints we only managed to make Dispersy's database I/O asynchronous. However, the foundation for Tribler's database I/O to become asynchronous has been created and the insights provided are good starting points for future work.

All in all we believe we have contributed to the goal of decentralized systems such as Tribler to become as performant as centralized solutions.

Chapter 7

Conclusion and Future Work

This thesis aims to contribute to the goal of re-decentralisation of systems such as Tribler to become as performant and as accessible as centralized solutions such as YouTube. We have addressed Tribler's blocking database I/O, its main performance bottleneck, by integrating the Storm database framework into a new database manager: 'StormDBManager'. StormDBManager features a complete asynchronous, non-blocking interface for database access while still maintaining a serialized query execution strategy. Furthermore we have provided deep insight into Tribler's and Dispersy's database usage, pinpointing functions that could be reviewed for query optimization. By making Dispersy's database I/O asynchronous and non-blocking, we have improved Tribler's API throughput by up to 150%, reduced its response times by up to 57.5% and moved its longtail latencies to the 99.9th percentile up from the 90th percentile.

Additionally, we have created a regression testing system and prepared it to be integrated in our Jenkins continuous integration system to adopt software performance engineering in the development cycle, further maturing the project. We have verified both the regression testing system and the resolving of the bottlenecks by providing experimental results. We believe that with this performance boost and software performance engineering focus, we have contributed to Tribler's further years of research and strengthened Tribler's chances on becoming a decentralized alternative for YouTube-like streaming.

While we believe we made a significant step forward in both performance and software performance engineering, there are items left for future work.

Tribler's database I/O time is significantly larger than Dispersy's. It is clear that Tribler can become more performant if this I/O is made asynchronous and non-blocking using StormDBManager as well. In terms of performance left to gain this will be the most low-hanging fruit.

As discussed in Section 5.4.1, different platforms and operating systems may influence the performance of a program, it is useful to run regression tests on all platforms Tribler supports. Deploying the regression testing system on all platforms will ensure no regression occurs on one of them.

To remove the ‘raw queries’ from all code bases, an object-relational mapping approach can be applied. This will reduce the complexity of the system as all data will be contained in objects which are generally easier to modify and read from. This will require extensive refactoring of Tribler and Dispersy’s code bases.

While we have managed to move the longtail latencies to the 99.9th percentile and reduce the size of these latencies, there is still room for improvement. Removing them completely by buffering or other means can lead to further improvements which could be investigated.

To improve performance further, moving Tribler from Python2 to Python3 is another possibility. As we have seen, the Global Interpreter Lock in Python 3.2 and onwards has been updated to handle I/O-bound threads better. This item will also require extensive effort as certain functions have changed or may no longer exist in this new version.

To increase the speed of processing database queries, it can be investigated if we can use SQLite’s multi-threaded support to process more queries. While the SQLite developers themselves admit SQLite has minimal multithreaded support, it can still be investigated to what extent we can leverage its support.

Improving the quality of Dispersy’s tests is another item that was mentioned in this thesis. It was found that passing all tests in Dispersy does not provide any guarantees beyond a basic level of correctness. Improving and adding tests is required to increase the stability, maintainability and correctness of Dispersy.

A final item that we would like to highlight as future work is running benchmarks in a closed environment, disconnected from the Internet. When running Tribler and Dispersy, it will connect to the Internet which may influence performance metrics such as I/O rates and amount of packets obtained. Creating a closed environment with local peers will ensure that only those peers can communicate to one another. These peers can then be instrumented to behave in a predetermined manner. This will increase both the reliability and accuracy of the measurements made.

Bibliography

[1] C. U. Smith and L. G. Williams, "Software performance engineering," in *UML for Real*, pp. 343–365, Springer, 2003.

[2] M. Mayer, "In search of a better, faster, stronger web," in *Velocity Conference*, 2009.

[3] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 60–71, ACM, 2014.

[4] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering, 2007. FOSE'07*, pp. 171–187, IEEE, 2007.

[5] L. G. Williams and C. U. Smith, "Performance evaluation of software architectures," in *Proceedings of the 1st international workshop on Software and performance*, pp. 164–177, ACM, 1998.

[6] C. U. Smith and L. G. Williams, "Best practices for software performance engineering," in *Int. CMG Conference*, pp. 83–92, 2003.

[7] GitHub, "Releases," 2016.

[8] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?," *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[9] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *International Conference on Mobile Computing, Applications, and Services*, pp. 59–79, Springer, 2010.

[10] R. Plak, *Anonymous Internet: Anonymizing peer-to-peer traffic using applied cryptography*. PhD thesis, TU Delft, Delft University of Technology, 2014.

[11] R. J. Tanaskoski, *Anonymous HD video streaming*. PhD thesis, TU Delft, Delft University of Technology, 2014.

[12] R. Ruigrok, *BitTorrent file sharing using Tor-like hidden services*. PhD thesis, TU Delft, Delft University of Technology, 2015.

[13] J. Pouwelse, “Reduce io activity from 623mbyte/hour,” 2014.

[14] A. Mainka, S. Hartmann, W. G. Stock, and I. Peters, “Government and social media: A case study of 31 informational world cities,” in *2014 47th Hawaii International Conference on System Sciences*, pp. 1715–1724, IEEE, 2014.

[15] J. Sharma, “Strategies to overcome dark side of social media for organizational sustainability,” *International Journal of Virtual Communities and Social Networking (IJVCSN)*, vol. 8, no. 1, pp. 42–52, 2016.

[16] redecentralize.org, “Alternative internet,” 2016.

[17] OpenHUB, “Tribler,” 2016.

[18] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy, “Finding and removing performance bottlenecks in large systems,” in *European Conference on Object-Oriented Programming*, pp. 172–196, Springer, 2004.

[19] J. Pouwelse, “Documentation: Tribler, dispersy and libswift developers portal,” 2013.

[20] Tribler, “Dispersy,” 2016.

[21] N. Zeilemaker, B. Schoon, and J. Pouwelse, “Dispersy bundle synchronization,” *TU Delft, Parallel and Distributed Systems*, 2013.

[22] S. D. Norberhuis, *MultiChain: A cybercurrency for cooperation*. PhD thesis, TU Delft, Delft University of Technology, 2015.

[23] B. Jacob, P. Larson, B. Leitao, and S. Da Silva, “Systemtap: instrumenting the linux kernel for analyzing performance and functional problems,” *IBM Redbook*, 2008.

[24] D. Beazley, “Understanding the python gil,” in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[25] D. Beazley, “Inside the python gil (slides),” in *Python Concurrency Workshop*, 2009.

[26] Python, “Global interpreter lock,” 2015.

[27] F. Tabba, “Adding concurrency in python using a commercial processor’s hardware transactional memory support,” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 5, pp. 12–19, 2010.

[28] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, “Tracing the meta-level: Pypy’s tracing jit compiler,” in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25, ACM, 2009.

[29] PyPy, “Pypy,” 2016.

[30] Pypy, “Speed center,” 2016.

[31] Jython, “Why jython,” 2016.

[32] Python, “Ironpython,” 2014.

[33] asyncio.org, “Python async io resources,” 2016.

[34] A. Sherman and P. Den Hartog, “Deco: Polishing python parallel programming,” 2016.

[35] K. Kinder, “Event-driven programming with twisted and python,” *Linux journal*, vol. 2005, no. 131, p. 6, 2005.

[36] G. Lefkowitz and I. Shtull-Trauring, “Network programming for the rest of us.,” in *USENIX Annual Technical Conference, FREENIX Track*, pp. 77–89, 2003.

[37] J. Pouwelse, “Consider a new networking/threading framework: Twisted, gevent or keep our own,” 2013.

[38] Canonical, “Storm,” 2011.

[39] D. S. Janzen and H. Saiedian, “Test-driven development: Concepts, taxonomy, and future direction,” *Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[40] X. Ferré, N. Juristo, H. Windl, and L. Constantine, “Usability basics for software developers,” *IEEE software*, vol. 18, no. 1, p. 22, 2001.

[41] M. Meulpolder, J. A. Pouwelse, D. H. Epema, and H. J. Sips, “Bartercast: A practical approach to prevent lazy freeriding in p2p networks,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, IEEE, 2009.

[42] E. Adar and B. A. Huberman, “Free riding on gnutella,” *First monday*, vol. 5, no. 10, 2000.

[43] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown, “Attention web designers: You have 50 milliseconds to make a good first impression!,” *Behaviour & information technology*, vol. 25, no. 2, pp. 115–126, 2006.

[44] A. Gupta, A. Tucker, and S. Urushibara, “The impact of operating system scheduling policies and synchronization methods of performance of parallel applications,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 19, pp. 120–132, ACM, 1991.

[45] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block i/o: Rethinking traditional storage primitives," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 301–311, IEEE, 2011.

[46] O. C. Lin, A. T. Kwee, and F. S. Tsai, "Database optimization for novelty detection," in *Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference on*, pp. 1–5, IEEE, 2009.

[47] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)*, vol. 16, no. 2, pp. 111–152, 1984.

[48] R. Hsu, "Who moved my 99th percentile latency?," 2015.