Identifying and Managing Technical Debt in Complex Distributed Systems

by

M.A. de Vos

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 30, 2016 at 3:00 PM.

Student number:	4135121	
Project duration:	November 1, 2015 – August 30, 2016	
Thesis committee:	Prof. dr. ir. J.P. Pouwelse,	TU Delft, supervisor
	Dr. Ir. A. van Deursen,	TU Delft
	Dr. Ir. C. Hauff,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

The term *technical debt* has been used to described the increased cost of changing or maintaining a system due to expedient shortcuts taken during development, possibly due to budget or time constraints. The term has gained significant attention in the agile and academic community and several models have been proposed to keep track of and solve technical debt.

Tribler, a platform to share and discover content in a complete decentralized way, has accumulated a tremendous amount of technical debt over the last ten years of scientific research in the area of peer-to-peer networking. We will focus on identifying technical debt and the question on how to manage and prevent it. This thesis will describe various disruptive improvements to the architecture, user interface and testing framework of Tribler. With the deletion of 12.581 lines, the modification of 765 lines and addition of 12.429 lines, we show that various important software metrics are improved.

Our performance experiments will demonstrate that the performance of Tribler has not significantly degraded due to the invasive modifications.

verbeteren

Preface

Preface…

M.A. de Vos
Delft, January 2013

Contents

1 Introduction

The resources, budget and time frame of software engineering projects are often constrained. This requires software engineers to analyse trade-offs that have to be made in order to meet deadlines and budgets. Making decisions that are beneficial on the short term, might lead to significantly increased maintenance costs in the long run. The phenomenon of favouring short-term development goals over longer term requirements is often referred to as *technical debt*. While technical debt might not have implicit consequences on the user experience, it dramatically impacts quality and maintainability of the software.

The term technical debt was first introduced by Ward Cunningham as writing ”not quite right” code in order to ship a new product or feature to market faster[12]. Since then, the term has gained progressively more attention in the software engineering research and the agile community. Effective management of such debt is considered critical to achieve and maintain a high level of software quality. In 2007, Steve McConnell created the technical debt taxonomy where he refined and expanded the definition[1]. He points out that some kind of engineering practices are not considered technical debt, such as deferred features, incomplete work that is not shipped and other features where one does not have to ’pay’ debt for. Martin Fowler considers technical debt more as a metaphor to use when communicating with non-technical people and introduced the technical debt quadrant in 2009[3]. According to his work, technical debt can be categorized in distinct types, separating issues arising from recklessness from those decisions that are made strategically. Figure 1.1 presents this distinction in more detail.

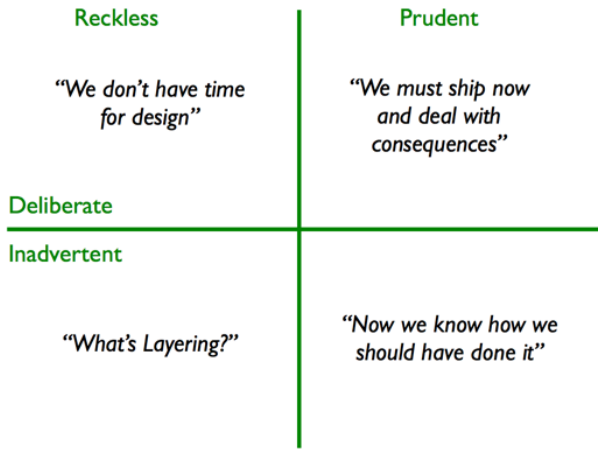


Figure 1.1: The technical debt quadrant, as proposed by Martin Fowler.

Technical debt has several interesting properties, explored and defined in the work of Brown et al[10]. Whether the debt is visible or not is an important factor during software engineering as significant problems can arise when the debt is not clearly visible or documented. The value of the technical debt is

the economic difference between the system as it is and the system in an ideal state for the assumed environment. The technical debt is relative to a given or assumed environment. The phenomena has an origin which can be introduced by strategic decisions or accumulated in a more unintentional way, either due to recklessness or lack of knowledge. Finally, we can consider the impact of the technical debt: for instance, what are the required changes we have to perform in order to pay off the debt?

Technical debt can be both invisible and visible to end users[20]. Examples of invisible technical debt includes code smells, coding style violations, low internal quality and high code complexity. Visible technical debt is expressed in bugs that are affecting users but can also be identified by user-unfriendly, cluttered graphical user interfaces. Decisions to extend and evolve the graphical user interface with new visual elements, can lead to a high amount of technical debt and a poor user experience.

The term itself is borrowed from the finance domain[16]. There is however one important distinction between financial debt and technical debt. When working with financial debt, the costs that the debtor has to pay is usually clear. This is not always the case with technical debt: there might be some situations where no technical debt incurred. For instance, if it is known for a part of the system to never be updated or maintained in the future, time can be saved by not updating the related documentation. Software engineers have to carefully consider what technical debt they wish to incur and when this debt will be paid off.

There are several causes that contribute to the amount of accumulated technical debt during the software development process[21]. While notable to a lesser extent in research-oriented software engineering, time pressure can cause developers to think reckless about their architecture. Uncertainty in decision making during an early stage of development might lead to higher technical debt. Finally, in an agile environment, software requirements might change more often, causing the underlying architecture and code base to change. Not properly managing such changes can lead to significant technical debt.

Technical debt often becomes a noticeable problem in large systems, having a large number of contributors. Tribler is an example of such a system. The software is the result of 10 ongoing years of scientific research in the field of decentralized network technology and has incurred a serious amount of technical debt, both visible and invisible for users. Tribler is the combination of four disruptive techniques in one large code base: BitTorrent, allowing users to download files in a decentralized matter, Tor, providing anonymity and strong encryption, Bitcoin, providing a way to introduce the notion of trustworthiness inside the network and Wikipedia, providing collaborative editing of content. This is depicted in Figure 1.2.

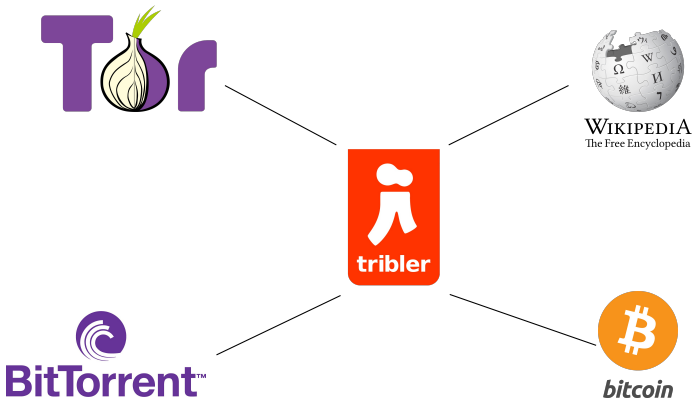


Figure 1.2: The four disruptive technologies as integrated in Tribler.

Anonymity by using a Tor-like protocol has been added In 2014 by the work of R. Plak[24] and J.H. Tanaskoski[29]. In 2015, the protocol has been extended to support for anonymous seeding of torrents[26]. The graphical user interface of Tribler is shown in figure 1.3.

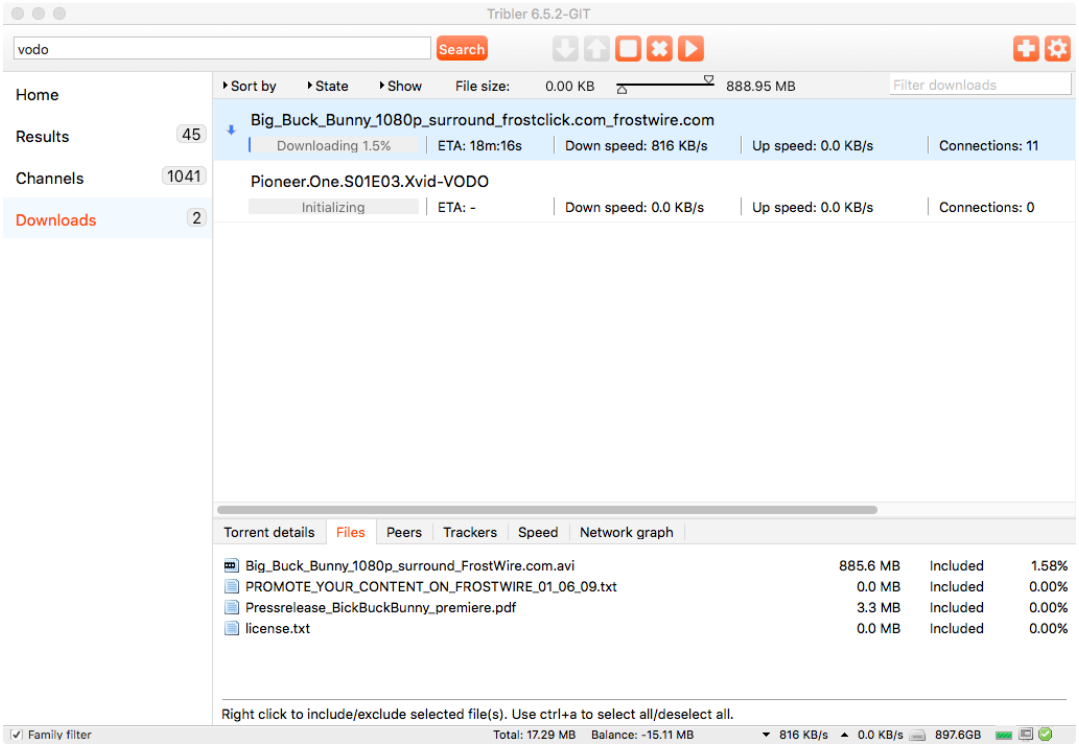


Figure 1.3: The graphical user interface of Tribler v6.5.2.

The focus of this thesis will be tracking and managing technical debt in Tribler. The following research question can be formulated:

How can we track and manage technical debt within Tribler?

This question can be divided into some sub questions:

zijn deze nog cor

1. What tools should be used to identity technical debt within Tribler?
2. What kind of technical debt should be prioritized for fixing?
3. What are the adequate requirements in the software development process to make the right decisions about incurring technical debt in the future?

The rest of this document is outlined as follows: in Chapter 2, the current state of the system will be elaborated, highlighting flaws and impurity in the architecture and code base. In Chapter 3, the evolution of Tribler over the past 10 years will be presented and we build foundations for the next decade of scientific research with Tribler by proposing a new future-proof and robust architecture. Various efforts to improve quality assurance, code quality and infrastructure while paying off technical debt will be explained in Chapter 5. The performance of Tribler after refactoring efforts will be discussed in Chapter 6. By conduction various benchmarks and performance measurements, the user experience of Tribler will be assessed. We will end with the conclusions and propose future work in Chapter 7.

2

Problem Description

The goal of this thesis project is to help Tribler mature from an experimental research prototype into production-level code with potentially reliable usage by millions of users.

After careful analysis it was decided that within the context of a nine month project the strongest contribution to the future of Tribler would be a step forward in technical debt. At this point we believe the project does not need a particular focus on feature improvements, novel additional features, or boosting performance. After over ten years of software development by 44 unique contributors the amount of accumulated technical debt is worrying.

Tribler suffers from all kinds of technical debt, including instability issues, race conditions, coding style violations, code complexity and feature pollution in the graphical user interface. To illustrate, there is even a dedicated file, called *hacks.py* that facilitates some workarounds caused by incompatible software.

This thesis is focussed on a round of invasive maintenance and cleaning of the code and all other infrastructure such as installers and testing environment. Our work aims to ensure that it is possible to conduct another decade of experimental distributed systems research with the Tribler code base. The alternative is continued usage and expansion of the code, which are likely to lead to a forced clean slate approach.

The structural problem is the lack of maintenance capacity. Each contributor to the Tribler research in the form of a bachelor, master, or PhD student needs to be primarily focussed on their thesis work. A thesis requires concrete experimental results, contribution to theory, or both. We believe the lack of student enthusiasm for fixing bugs and writing documentation is the root cause of current state of the code base.

In this remaining of this Chapter, various problems within the Tribler project will be highlighted.

2.1. A large code base

Tribler has a very large, complex code base. This makes Tribler an unattractive open-source project for external developers since the process to get familiar with the code base takes a long time. Figure 2.1 illustrates the number of commits over the past 10 years. The evolution of number of source code lines is shown in Figure 2.2. The magnitude of the project is also presented by Figure 2.3. From these figures, it is clear that Tribler has continued to grow to a project with an unmaintainable amount of code. According to the basic COCOMO[19] model, the established costs of the project is $2,371,403 with an estimated effort of 43 person-years.

This continue growth can be explained by the fact that Tribler is a research-oriented prototype. Students often contribute to Tribler by implementing a specific feature of the system, such as anonymous

downloads, credit mining mechanisms or an adult filter to filter out explicit content. After completions of these features, the student leaves the project and the knowledge about the specific part of Tribler he or she contributed to, is lost. Afterwards, that part of Tribler is not maintained any more, due to lack of knowledge and manpower constraints.

Continuous expansion of a system inevitably leads to feature pollution. During the past 10 years, no single effort has been made to do a proper clean up of the current code. The code base contains a huge amount of technical debt. If this trend continues, Tribler will evolve into an tremendously complex system where the choice to use a clean-slate approach is favoured over continued usage of the current code base.

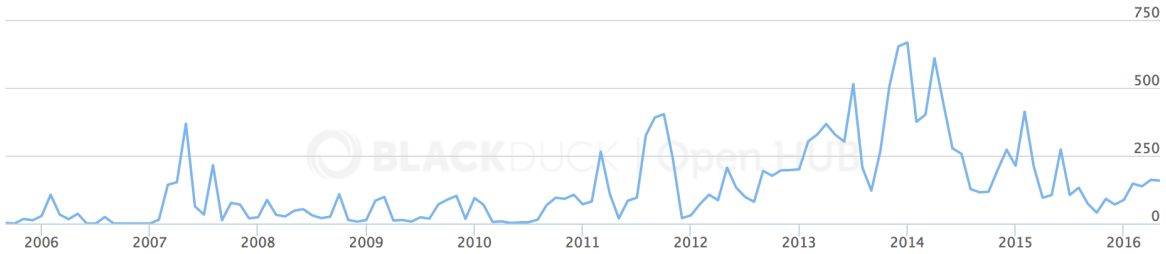


Figure 2.1: A history of commits per month on the Tribler project, as reported by Open Hub.

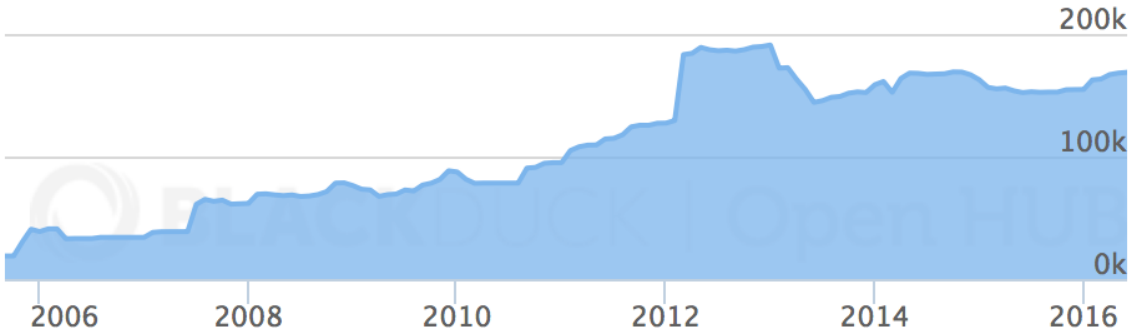


Figure 2.2: The evolution of lines of code in the Tribler project, as reported by Open Hub.

	All Time	12 Month	30 Day
Commits:	17175	1670	390
Contributors:	111	38	16
Files Modified:	10190	1048	270
Lines Added:	2223864	93098	27490
Lines Removed	2037749	57019	11508

Figure 2.3: Statistics about modifications to the code base, as reported by Open Hub.

2.2. Lack of Maintenance

Many features of Tribler are completely unmaintained, due to lack of knowledge or resource constraints. There are even some experimental features that are not working anymore and could be removed.

The lack of maintenance is clearly visible in the synchronization system of Tribler, called *Dispersy*. Dispersy is a platform to simplify the design of distributed communities and is mostly designed and written by N. Zeilemaker and B. Schoon[31]. After these two developers left the project, knowledge of

the Dispersy system disappeared and the system transited to an unmaintained state.

Most researchers working with Tribler have a specific feature to deliver. This means that defects in unmaintained parts of Tribler are not prioritized, causing long outstanding issues on GitHub that are not resolved and delayed for many major or minor release. Of the 300 total issues on GitHub, 100 issues are older than one year.

2.3. Architectural Impurity

During the lifetime of Tribler, the architecture has been subject to various minor and major modifications. Starting as a fork from ABC, a Bittorrent client based on Bittornado, Tribler has evolved to a platform that allows users to discover, share and download content. The evolution of the Tribler architecture will be explained in-depth in Chapter 3.

On the highest code-base level, two main components in Tribler can be identified: the module with code providing the graphical user interface and the code that contains the core of Tribler. These modules have a mutual dependency on each other which is considered bad design since the Tribler core should never be dependent on the user interface. Splitting this dependency is a high priority issue.

Overall, the code base feels like a bunch of glued together research works whereby every developer maintained his own code style and practices. No clear design patterns can be identified throughout the code and there is a staggering amount of legacy code that is either unused or can be removed. After more analysis of the core module, we managed to identify some other issues. Code that facilitates a video player is present in the core package while this is clearly related to the user interface. We have two files with configuration parameters that can easily be merged to reduce complexity and increase maintainability.

The code related to the graphical user interface is of poor quality and plague with many cyclic dependencies. Having two files being dependent on each other, makes testing of the classes that these files contains, significantly harder. To better see where the main problems are, we created an import graph of the code base that is related to the user interface, visible in Figure 2.4. A red edge indicates that this import is part of a cycle. Having many cyclic imports can indicate a bad design. Besides the huge amount of cyclic references, we notice that there are various files which seems to have a huge number of incoming references, possibly indicating that these classes have too much responsibilities and should be split into smaller components.

While the current decade of software engineering provides a plethora of visual designers that requires barely any hand-written code, our whole graphical user interface is built with code that's unmaintained and hard to read. Many features and visual elements in the graphical user interface are unnecessary and unintuitive, contributing to the technical debt. Finally, the user interface has been written with the *wxPython* framework which is not maintained anymore since late 2014. The library builds upon native APIs, i.e. Cocoa on OS X and Win32 on Windows. While the library claims to be cross-platform with a native look and feel, various features in Tribler are limited to a subset of the supported platforms.

2.4. Unstable and incomplete testing framework

Proper testing is the responsibility of every developer. Over the past 10 years, this responsibility has been completely neglected by the majority of contributors. This is clearly visible in Figure 2.5 where we plot the ratio between the amount of code lines in the tests package and the number of code lines related to production code in Tribler. Tribler has a structural lack of proper designed unit tests. Currently, around 100 tests are available that cover 71,2% of the Tribler core. Many of these tests are taking over half a minute to complete and are bootstrapping an extensive Tribler session. Only a small fraction of the test suite has the characteristics of unit tests. Having tests that are doing a broad range of operations, inevitably leads to undesired side-effect and failing tests. No single attempt has been made to mock components of the system to simplify tests and focus on the part of the system that has to be verified.

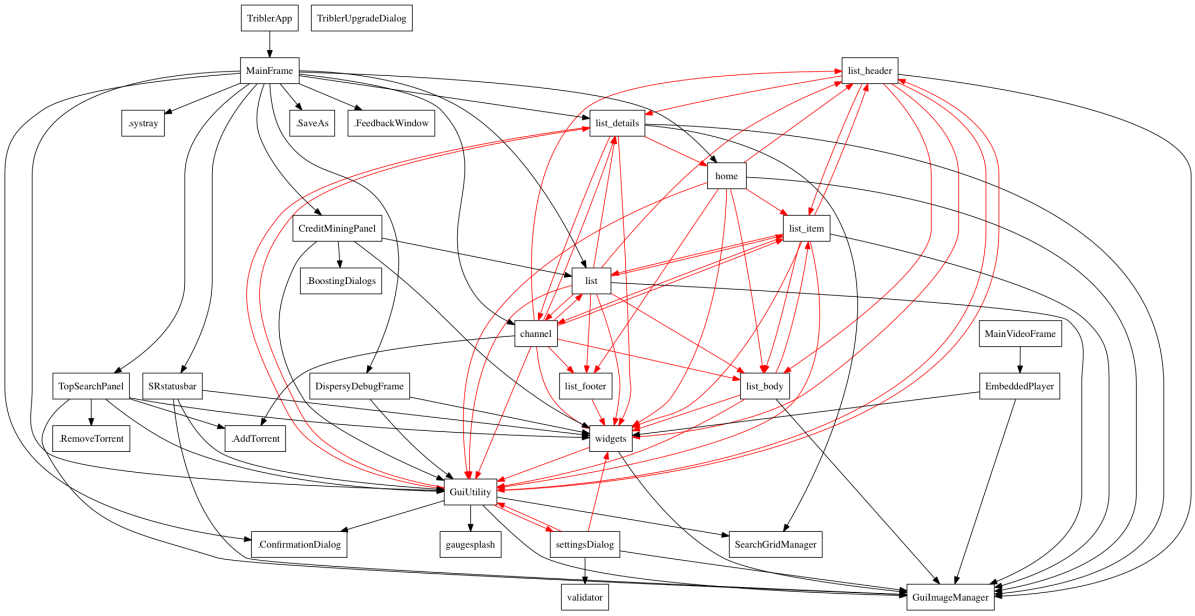


Figure 2.4: A generated import graph of the user interface code package. A red edge indicates that this edge is part of an import cycle.

There is one more factor that contributes to the instability of the current test suite. A significant part of the test suite is depending on external network resources, ranging from trackers and seeders for a specific torrent to other peers in the decentralized network. This fragile architecture gives rise to failing tests due to unavailable nodes, unexpected responses from external peers and other unpredicted circumstances.

In general, well designed tests exclude any dependency on external resource that is outside the control of the developer. This can be achieved by mocking method calls to return dummy data. Additionally, one can make sure that the external resource is available in the local testing environment. For instance, when a test is dependent on a specific torrent seeder, a local libtorrent session can be started that seeds this torrent.

While Tribler is packaged and distributed for multiple platforms, unit tests in our continuous integration environment are only executed on a machine running a Linux operating system. Limiting test execution to one platform, lowers the overall code coverage and covers platform-specific bugs. Attention should be given to make our test execution multi-platform.

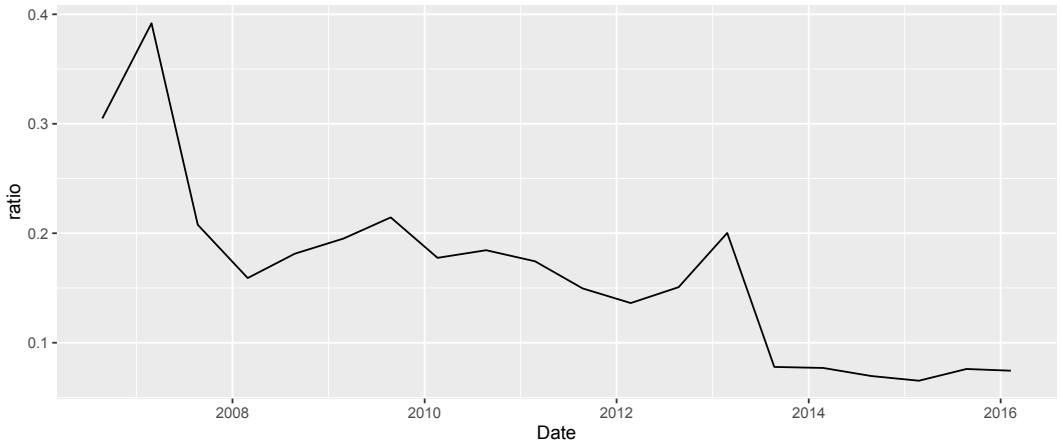


Figure 2.5: The ratio between the lines of code in the tests package and other lines of code over time.

3

Architecture and Design

Before we reach the point of proposing a new future-proof, modern architecture and design of Tribler, the evolution of the Tribler architecture throughout the last decade of research in the area of decentralized networks will be elaborated. Understanding the past decisions regarding architecture and design, will help us to shed light on the question why Tribler has accumulated such an amount of technical debt.

According to OpenHub which accumulates statistics about open-source projects, the Tribler project has seen a total of 111 unique contributors so far[7]. This list is most likely not complete since some work of missing contributors might have been finished by other members of the Tribler team. Searching for *Tribler* in the TU Delft repository, results in a total of 66 hits, consisting of 35 results which are contributions in the form of a MSc or BSc thesis and 31 research-oriented papers in the form of a PhD dissertation or (published) paper.

The remaining of this Chapter will present a historical description of the evolution of the Tribler platform. The purpose is to present a historical evolution of the code base so we can better understand the design decisions made in the past and how they might have led to the present state of the system.

3.1. Tribler: A social-based peer-to-peer system

In April 2005, Tribler started out as fork from *Another BitTorrent Client* (ABC), an improved BitTorrent client. ABC is based on *BitTornado* which extended from the *BitTorrent* core system, written by Bram Cohen. In the current code base of Tribler, various references to ABC can be found, mostly notable in the naming of files, variables and classes.

In 2007, the first major research paper was published, describing Tribler as a social-based peer-to-peer system[25]. The key idea is that social connections between peers in a decentralized network can be exploited to increase usability and performance of the network. This is based on the idea that peers belonging to a social group are not likely to steal (free-ride) bandwidth from each other. The system architecture of the system is visible in Figure 3.1. In the remainder of this Section, the most important components of the architecture as described in [25] will be explained in more detail.

3.1.1. Collaborative Downloads

The BitTorrent engine allows the mechanism to download and seed torrent files using a BitTorrent-compatible protocol. In addition, the module allows usage of the *collaborative downloader* module which significantly increases download speed by exploiting idle upload capacity of online friends.

The protocol to facilitate collaborative downloads is called *2Fast* and the idea is that a user invokes the help of friends to increase download speed of his files. The protocol uses social groups where members who trust each other collaborate to improve their download performance. Peers that are participating in a social group are either *collectors* or *helpers*. A collector is a peer that is interested in obtaining a complete copy of a particular file. A helper is a peer that is recruited by a collector to help downloading

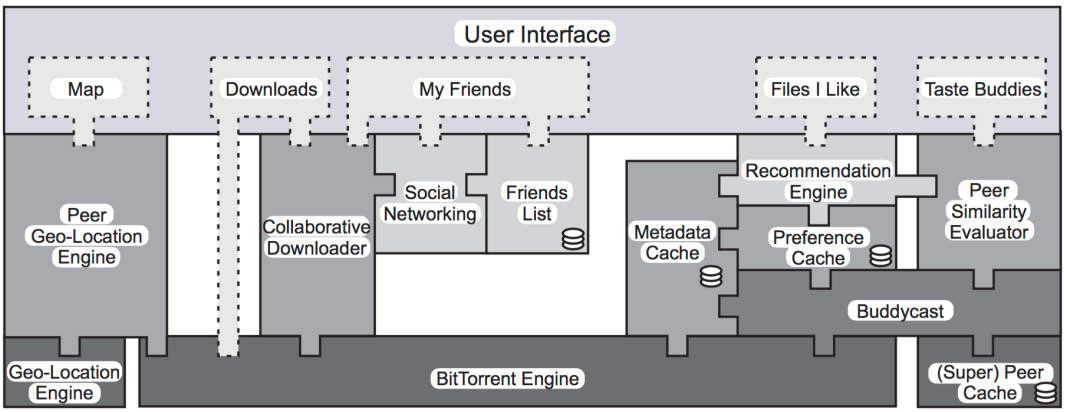


Figure 3.1: The system architecture as described in [25].

that file.

3.1.2. Peer Geo-Location Engine

The *Peer Geo-Location Engine* is an engine built on top of a module that enables geographical lookup of IP addresses (*http://hostip.info*). When a peer downloads a file, peers in the download swarm are displayed on a map in the user interface.

3.1.3. Content Discovery and Recommendation

The *BuddyCast* algorithm is designed to make recommendations and to enable peer and content discovery. BuddyCast is an epidemic protocol which works as follows: each peer in the network maintains a number of taste buddies with their preference lists and a numer of random peers without any information about their preferences. Periodically, an *exploration* or *exploitation* step is performed. When an exploration is executed, the peer connects to one of its taste buddies. The peer connects to a random peer if an exploitation step is performed. When the connection is successful, a *BuddyCast* message is exchanged.

A Buddycast message contains the identities of a number of taste buddies along with their top-10 preference lists, a number of random (and fresh, see below) peers, and the top-50 preferences of the sending peer. The age of each peer is included in the message to help others know the freshness of peers. After the BuddyCast messages are exchanged, the received information is stored in the local database of each peer. To limit redundant messages, each peer maintains a list of recently contacted peers.

3.2. Tribler in 2007 and onwards

The next version of Tribler, version 4.0, got released in 2007[2]. Many features from the 3.x release cycles are taken over and some new functionality have been added. Using an embedded video player, users can play videos (while being downloaded) directly from within the user interface. This video player is powered by the VLC library and bindings for the user interface library, *wxPython*. Tribler 4.0 allows users to remotely search for content inside the Tribler network but also content on YouTube and Liveleak. The search results are displayed in a YouTube-like thumbnail grid. The interface of Tribler 4.0 is displayed in Figure 3.2.

3.2.1. Tribler 5.x

The development of Tribler continued with the release of Tribler 5.0 in 2009[4]. The interface got a complete overhaul, introducing a more dark theme, which was replaced by a white theme a short while after the release. In the 5.x series, several improvements to Tribler itself and the user interface has been made. The focus of Tribler 5.0 has been on remote search and downloads. The thumbnails have been dropped and the search results are displayed in a paginated list instead.

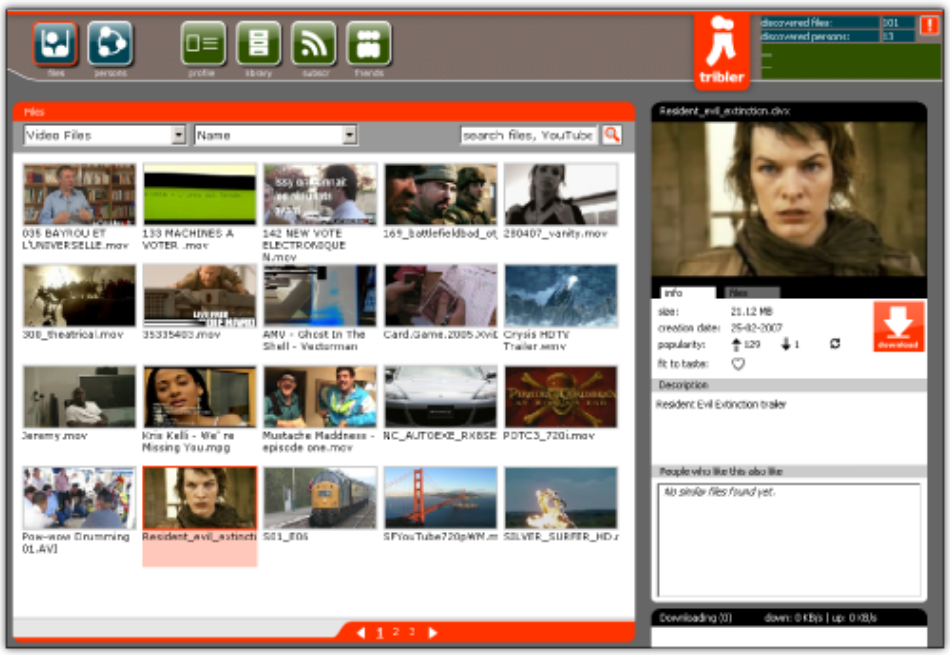


Figure 3.2: The user interface of Tribler 4.0.

Tribler 5.1 contained some major improvements to the user interface, thanks to the feedback of the community. A new major addition have been added to Tribler 5.2: the concepts of channels, similar to YouTube, has been introduced. One goal of channels was to prevent spam inside the network by favouring content in more popular channels. The popularity indicator associated with each torrent got removed but has been placed back later. In Tribler 5.3, all custom widgets got replaced by native buttons, creating a more native feel on each platform. A tag cloud has been added to the home page of Tribler to help users determine which content they possibly want to look for. Moreover, the paginated list was replaced by a single, scrollable list of items. Tribler 5.4 introduced a magic search feature where similar search results are collapsed using text similarity functions and digit extraction. This leads to a much cleaner and comprehensive results list when searching.

The final release in the 5.x series, Tribler 5.9, bought some major additions. The complete BuddyCast core has been rewritten, moving away from a TCP overlay to an implementation based on UDP, bringing huge benefits to the compatibility with NAT-firewalls. Also, *libswift* has been introduced as the new download engine, providing download capabilities over UDP, removing the TCP layer from libtorrent.

The architecture around the time of Tribler 5.5 is visible in Figure 3.3. This quite complex architecture model is the result of gradually adding smaller components to Tribler that have been the product of research, such as *2fast*, *BuddyCast* and the *Secure Overlay*. We notice several different threads that have to be synchronized, increasing the complexity of the code.

3.3. Tribler 6.x

Shortly after the release of Tribler 5.9, Tribler 6.0 is released where a complete new user interface has been implemented. This version contained also some minor bug fixes that increased performance and usability. After the release of 6.0.1, several smaller releases (6.1, 6.2 and 6.3) were released. The focus shifted toward anonymous downloads and end-to-end encryption. These features were part of the Tribler 6.4 release, providing an experimental anonymous download mechanism and hidden seeding services. Moreover, the *libswift* dependency got dropped since it was not stable enough. This release also introduced the Trivial File Transfer Protocol (TFTP), used to transfer torrent files between peers in Tribler. The next release, Tribler 6.4.1, contained some major security fixes after an external

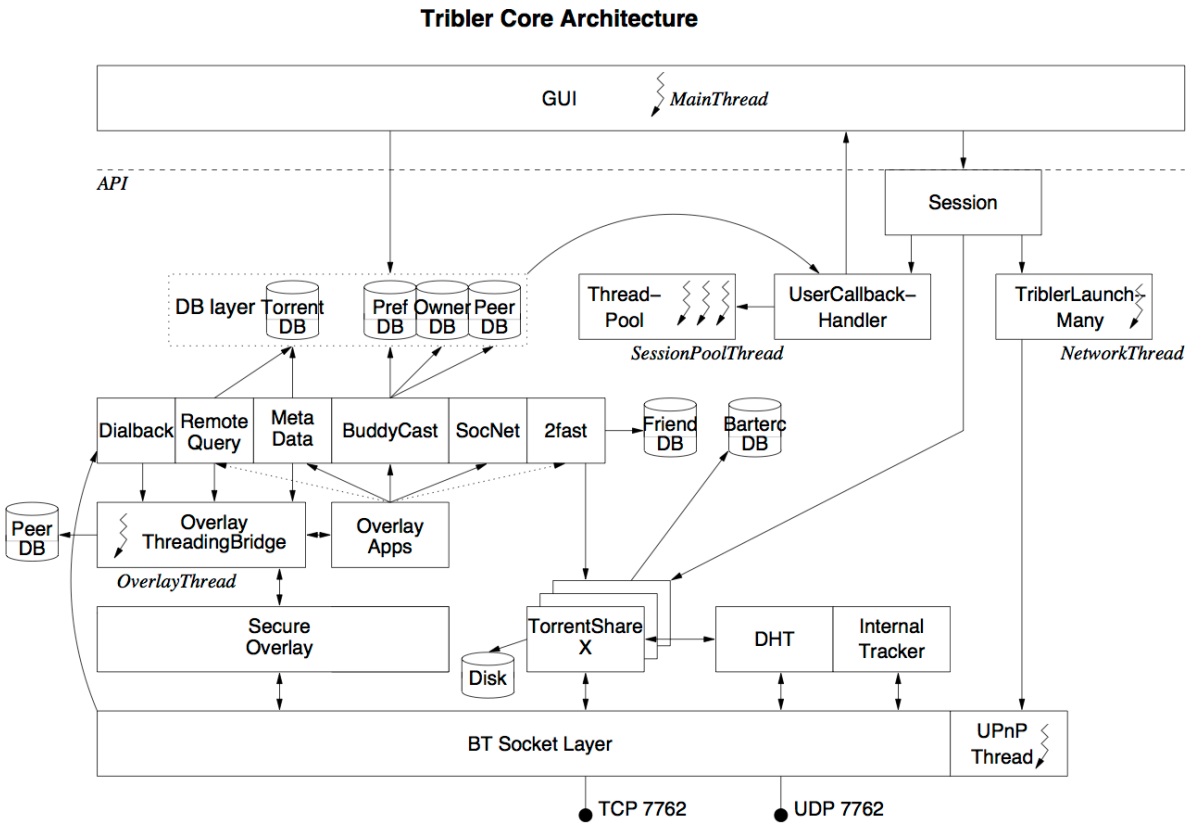


Figure 3.3: The architecture of the Tribler core around Tribler 5.5.

code review.

3.3.1. Dispersy

With the release of Tribler 6.1, Dispersy got introduced. Mainly developed by N. Zeilemaker and B. Schoon and described in [31], Dispersy lies at the foundations of Tribler's messaging and synchronization system and is designed to deliver messages reliably in unpredictable networks. Moreover, it contains a Network Address Translator (NAT) traversal mechanism to improve connectability of peers in the network. Dispersy provides tools to quickly create overlay networks, called communities, that users can join and where they can disseminate messages. The available communities, together with a short description, is described in Table 3.1. Dispersy is available as submodule to the project. While being a major dependency of Tribler, a throughough analysis of Dispersy is considered outside the scope of this thesis.

3.3.2. Twisted

In 2014, it was decided to make significant changes to the architecture by utilizing Twisted, an event-driven networking engine written in Python. Twisted allows programmers to write code in an asynchronous way. At the heart of Twisted, we find the reactor. The reactor is the core of the event loop within Twisted – the loop which drives applications using Twisted[8]. The event loop is a programming construct that waits for and dispatches events or messages in a program. The new threading model of Tribler 6 has been illustrated in Figure 5.4. In most Twisted applications, the reactor runs on the main thread of the Python application. In Tribler, the reactor runs on a separate thread since our main thread is occupied by the *wxPython* main loop. User interface-related code such as refresh operations of lists, should always be executed on the Python main thread. Twisted operations however, should be scheduled on the reactor thread. The Twisted threadpool provides some additional threads to dispatch work to and can be used for longer-running operations that should not block the main or reactor thread. Several method decorators have been introduced to make thread switching much easier to implement. These decorators are also visible in Figure 5.4.

Community	Purpose
AllChannel	This community is used to discover new channels and to perform channel search queries.
BarterCast4	While currently not enabled, this community is used to spread statistics about Tribler inside the network.
Channel	This community represents a single channel and is responsible for managing torrents, playlists and moderations inside a channel.
Multichain	The Multichain community utilizes blockchain technologies and can be regarded as the accountant mechanism that keeps track of shared and used bandwidth.
Search	This community contains messages to perform remote keyword searches for torrents and torrent collecting operations.
HiddenTunnelCommunity	The hidden tunnel community contains the Tor-like protocol that provides anonymity when downloading and contains the implementation of the hidden seeder services protocol used for anonymous seeding.

Table 3.1: An overview of implemented communities in Tribler as of July 2016.

Developers should be aware of the threading context when implementing new features. Long blocking calls on the main thread should be avoided as much as possible since they lead to an unresponsive user interface. Database calls however, should be scheduled on the reactor thread. While this reduced the number of threads if we compare to Figure 3.3 and makes our threading model somewhat easier, we are still stuck with a dedicated thread for the reactor.

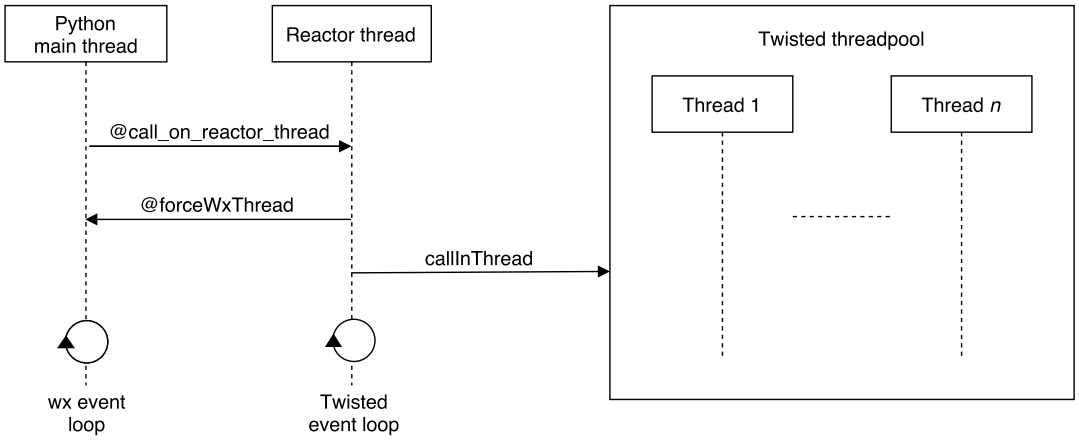


Figure 3.4: The threading model in Tribler 6, together with the primitives to schedule operations on different threads.

3.4. The roadmap of Tribler

In the previous Section, the evolution of Tribler has been evolved, up to the current architecture. We now turn our attention to the future of Tribler and propose a new architecture that will enable Tribler for another 10 year of research. The proposed architecture is visible in Figure 3.5. This architecture follows a layered approach with a clear distinction between the visible layers. This Section will discuss the proposed design and highlight decisions that have been made during the development process.

3.4.1. Trusted Peer Discovery

The trusted walker is located at the lowest layer of the architecture and is the central component for discovering other peers. At the moment, Dispersy is responsible for discovering other peers in the Tribler network, using a gossiping protocol[31]. This mechanism is illustrated in Figure 3.6. This mechanism is executed at a fixed interval. Suppose node *A* wants to discover an additional peer. First, he sends an *introduction-request* to a random peer he knows, say node *B*. Node *B* now replies with an

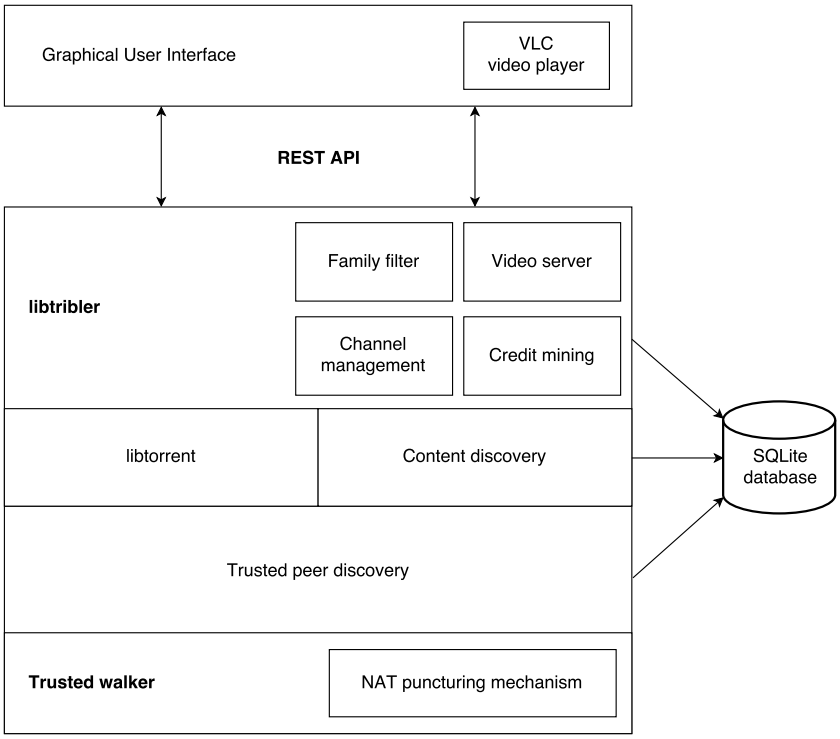


Figure 3.5: The proposed architecture of Tribler 7.

introduction-reply message, containing information about a node that *B* knows, in this case node *C*. Meanwhile, node *B* sends a *puncture-request* message to node *C* which in turn punctures the NAT of node *A*, making sure that node *A* can connect to him.

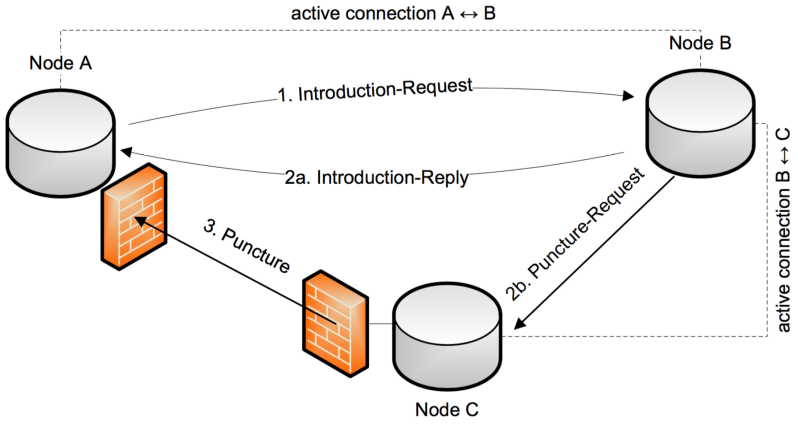


Figure 3.6: The discovery and NAT puncture mechanism as implemented in Dispersy.

The described mechanism will be replaced by a trusted walker that makes use of accumulated reputation in the *Multichain*, the accountant mechanism that keeps track of shared and used bandwidth, providing more reputation when a user provides bandwidth to help other users. The key idea is that sybil nodes, forged identities in the network, are ignored and not considered as a trusted peer since their reputation is low.

New peers in that network that have not built any reputation yet, start out by creating some random interactions with other nodes, learning about the network and the reputation of other users. With an

interval, every node runs an algorithm to calculate the reputation of known peers. The amount of uploaded data does not have to be the only factor of this reputation mechanism: the uptime of the user in question can also be considered, where a higher uptime might lead to a better reputation. This leads to a trust network where each node inside the network knows about other trusted peers.

3.4.2. Libtorrent and Content Discovery

Above the walker primitives explained in the previous Section, we find two components that are already present in the current system. The popular *libtorrent* library is used to facilitate downloads. Libtorrent is written in C++, however interfaces for other programming languages such as Python, Go and Java are available. *libtorrent* uses an alert mechanism to notify the program that is using the library about events in the library, such as download state transitions, peer discovery or completion of a meta info lookup in the Distributed Hash Table (DHT). The current way of using *libtorrent* in Tribler requires minimal changes to adhere to the proposed design, except for some optional refactoring and clean-up of the current code.

Currently, the methods to fetch peers from the DHT in *libtorrent* is private and not accessible from Python. The current solution for this is to make use of a third-party library, named *pymdht*, an implementation of the Mainline DHT protocol written in Python. This dependency is undesirable since it introduces some extra complexity and load of the system. Effort should be made to make the DHT method public so Tribler can get rid of the *pymdht* dependency.

The content discovery components allows users to discover and search torrents, channels and playlists in Tribler. The current implemented mechanism for information exchange in Tribler works well enough for this purpose, except for the exchange of meta data such as torrent thumbnails. Providing users with a visual preview of a torrent in the form of several thumbnails is a good opportunity to make the user interface more appealing, however, a robust implementation in a complete decentralized network might be challenging due to the fact that a channel can have many torrents, thus many thumbnails. We wish to keep the overhead introduced by thumbnail synchronization to a minimum and we must have a decent filtering algorithm to avoid inappropriate imagery from being shown unexpectedly in the user interface. This feature will be considered future work and not be discussed in the remainder of this thesis.

3.4.3. libtribler

Libtribler provides the interface to developers to make use of the above described primitives and contains the implementation of the REST API that is used to control and fetch data from *libtribler*. The remainder of this Subsection will discuss the components that can be found at this layer in more detail.

Family filter

The freedom to upload any content users want, comes with a price. The legal aspect of the available content inside the network can be legally disputable. Legal content that is often undesired for most users is also available in a high number, such as sexual content. A mechanism is currently implemented in the Tribler core to filter out such explicit content, called the family filter. This filter is enabled by default and uses a list of keywords that can be associated with pornographic content. Unfortunately, this ad-hoc approach is not very effective since there are various false positives and negatives. While it does the job, it also demonstrates that the keyword-based approach can be greatly improved. However, we would consider this as an enhancement rather than a defect that prevents a correct usage of Tribler.

Video Server

The video server streams the video data to a video player outside *libtribler* after or during downloads. The current implementation of the video server runs on a separate thread and is implemented using the *SimpleHTTPServer* library, a built-in module that can be used to easily implement a HTTP web server. The video server is based on HTTP range requests, allowing the web server to serve a part of a file when the user seeks in a video.

The inner working of the video player is illustrated in Figure 3.7. When the user starts playing a video in an external video player, the player performs a HTTP range request to the video server in Tribler. This

range request contains information about the requested range of the video file in the header. When the video server contains the request, it first checks whether the requested range has been downloaded already. If so, the video server immediately returns the requested data in the HTTP response. If the requested range is not available, the video server notifies libtorrent that the bytes in the requested range should be prioritized, reducing the latency before the requested range is completely downloaded. When all pieces are available, libtorrent notifies the video server and the video server completes the request by sending the data to the server.

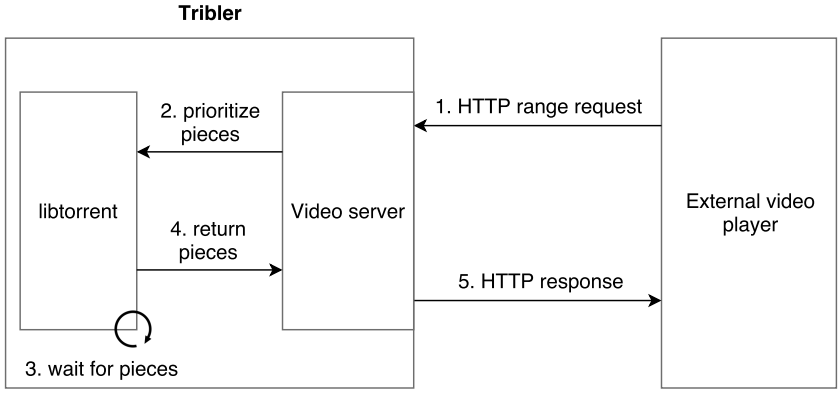


Figure 3.7: The inner working of the video server implemented in Tribler.

While the video server in the current form is functional, there is a major improvement that can be considered. The video player runs on a separate thread and uses blocking calls to wait for the buffer to be available. By integrating the video server inside the Twisted reactor thread, we can reduce complexity of this component and make use all of facilities that Twisted provides, for instance, for decoding a range request header. An additional consideration if performance is an issue, is to run the video server in a dedicated process. This might increase the complexity since a communication mechanism between the Tribler and video server process is required to inform libtorrent about the prioritization of pieces.

Credit Mining

The work on credit mining in a decentralized system has been extensively described by the work of Capotą et al[11] and is defined as the activity performed by peers for the purpose of earning credit. These credits can be used for accessing new content or for receiving preferential treatment in case of network congestion. Although not shipped to users, A credit mining system (CMS) has been implemented in Tribler which is responsible for contributing bandwidth to the community, without any intervention of the user. This mechanism is displayed in Figure 3.8 and works as follows: first, the user selects a source of swarms for the system to take in consideration. Possible sources can be channels, RSS feeds or a directory of torrent files. Next, the CMS periodically selects a subset of the chosen swarms by the user. Finally, Tribler join the swarms and tries to maximize earned credits by downloading as little as possible and uploading as much as possible.

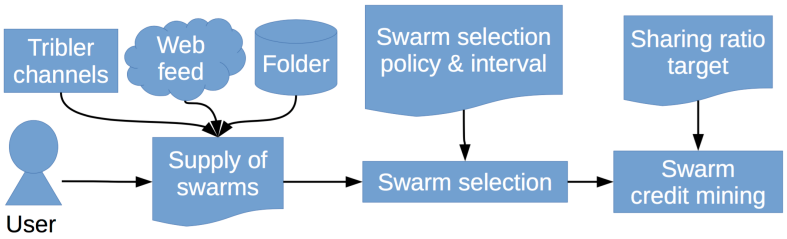


Figure 3.8: The credit mining system described in the work of Capotą et al.

CMS can apply different policies for swarm selection. A first policy is that it selects a swarm with the lowest ratio of seeders to all peers (where all peers consists of leechers and seeders). Intuitively, this boosts swarms that are under-supplied. The second policy is to select swarms based on their age. The intuition behind it is that newer content is often better seeded. The final policy that can be used is a random policy that selects a swarm with an uniform distribution.

This credit mechanism is a convenient way for users to build reputation by supplying bandwidth to the community. The Multichain can be used as accounting tool to keep track of upload and download rates.

Channel Management
Tribler allows users to create their own channel and share content within that channel. Content can be shared in the form of torrents and playlists where a playlist consists of a bundle of possibly related torrent, for instance, some episodes of a tv show. Users can add content to the channels of other users, providing that the owner of the channel has set the status of the channel to *open*.

3.4.4. Communication between the GUI and *libtribler*

The communication between the upper level of the Tribler architecture, the GUI, and *libtribler* is facilitated by a Representational State Transfer (REST) architecture. This architecture was introduced and defined by Roy Fielding in 2000[15] and is used frequently when building APIs that operate on the World Wide Web. A service that that conforms to the REST architecture, is called RESTful.

In a REST architecture, web resources are served to users, identified by Uniform Resource Identifiers (URIs). HTTP verbs (*GET*, *POST*, *PUT* and *DELETE*) are used to manipulate or retrieve these web resources. Operations can either be done on a collection of resources or on a specific resource and Table 3.2 illustrates the most common operations in a REST API, together with their usage.

Resource type	**GET**	**PUT**	**POST**	**DELETE**
Collection	Get the collection.	Replace the collection.	Create a new entry in the collection.	Delete the collection.
Item	Retrieve the item.	Replace the item, create it if it does not exist yet.	Often not used.	Delete the item.

Table 3.2: A summary of REST verbs and their usage when dealing with a resource collection or a resource item.

Prior to implementation of this API, we can already define some of the web resources. In Tribler, we can identify torrents, channels, playlists and downloads as resources that should be available for retrieval or modification using the API. Other than that, we might define a *debug* object that contains various statistics that are tracked by Tribler so developers can build a debug panel which is helpful during development or performance measurements.

A REST API provides a very flexible and high-level interface that allows developers to write applications around Tribler, ranging from a command-line interface (CLI) to graphical appealing user interfaces. Moreover, the implementation of these utility applications are not bound to a specific programming language, providing a huge amount of implementation freedom. It also allows to run the user interface and Tribler in separate processes, improving responsiveness and testability.

3.4.5. Graphical User Interface

At the highest level of the Tribler stack, we find the user interface that is shipped to and used by end-users. The user interface should be able to communicate with the Tribler core using the REST API as described in the previous Subsection. Another critical component of the user interface is the ability to play and control a video. The current user interface uses the *wxPython* bindings to the VLC player, however, this bindings are not functional on OS X due to platform incompatibilities.

The implementation of this interface is not limited to one programming language, however, to be able to reuse prior-existing code in the Tribler code base, it is a decent choice to write the interface in Python. Since the programming language is high-level and relatively easy to learn, new developers can easily make modifications to the user interface. We might also consider to refactor the current user interface to support the API. This consideration will be analysed in more detail in Chapter 5.

4

Towards a new architecture

The discussion in Chapter 3, ended with a proposal of a new architecture. Now we will shift our focus on efforts on making this design a reality. This will mostly involve building the components that are located on a higher level in the proposed architecture, namely the new user interface, the REST API and *libtribler*.

4.1. A clear distinction between GUI and core

As described in Chapter 2, the code for the user interface and Tribler core code is interleaved to a large extent and there is no clear separation between these two. There are many instances where we find code present in the user interface code base that should be moved to the core and vice versa. To realise a clear separation between *libtribler* and the user interface, instances like these should be fixed.

In the current code base, the *Core* package is dependent on the user interface which is a bad situation since this package cannot be reused without shipping the core. The exact dependency is visible in Figure x and is caused by the *DefaultDownloadStartupConfig* class which is located in the *globals.py* file, part of the user interface module. This class is responsible for getting some configuration options in case the user did not override the default options, such as destination of the downloaded file and the amount of anonymous hops used. Since the superclass of *DefaultDownloadStartupConfig*, *DownloadStartupConfig*, is already located in the core, the best option is to move the *DefaultDownloadStartupConfig* class to the *DownloadConfig.py* file, which already contains the *DownloadStartupConfig* class. After we moved this class to the core, the core is completely independent of the user interface.

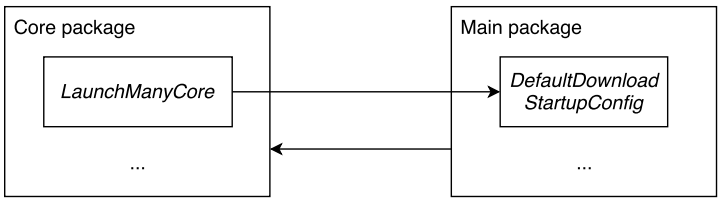
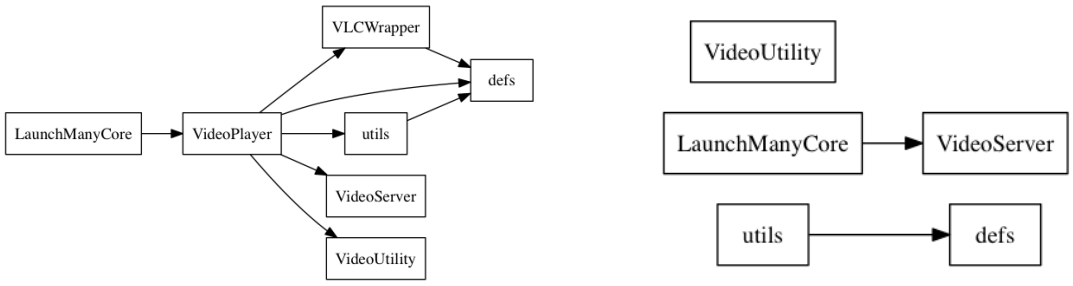


Figure 4.1: The cyclic dependency between the core and the main package.

We will now focus on the core package which contains some code that should not be present in the core package. The most interesting piece of interface-related code is attributed to the (embedded) video player in Tribler which is handled by the *VideoPlayer* class in the *Video* package. This class makes use of the VLC bindings for Python, however, the core does not need to have any knowledge about VLC: Managing the video player is a task that should be done on the level of the user interface. Figure 4.2a shows the *Video* package before refactoring. The *LaunchManyCore* class contains code to start-up all components available in Tribler, including the *VideoPlayer*. This *VideoPlayer* creates a *VideoServer* when being started. Moreover, the *VLCWrapper* class contains various utility methods to work with raw VLC data such as the time position within a video. We removed the *VideoServer* and *VLCWrapper* classes and the composition of the *Video* package after this operation is visible in Figure



(a) Before refactoring

(b) After refactoring

Figure 4.2: The import graph of the *Video* package in the Tribler core before and after refactoring.

4.2b. We modified the code such that *LaunchManyCore* the starts a video server instead of a video player. We note that there are some classes that appears to be unused, such as *VideoUtility* and *utils*: these classes contains some helper methods to retrieve thumbnail images from a video file. Due to time constraints, we are unable to implement these features in the new user interface so we keep these files as reference for future development.

4.2. REST API

As described in Chapter 3, communication between the graphical user interface and the Tribler core is facilitated using a REST API. This Section explains the implementation of the API in more detail.

The REST API is written using Twisted. While there are plenty of Python libraries available that allow developers to facilitating a web server in their application, Twisted is used since it already represents a big part of the internal structure of Tribler. With the possibility to integrate the REST API into the main application flow, we avoid having to create special constructions to run the API for instance on a separate thread. API endpoints in Twisted are represented as a resource tree. This is in accordance with REST where the URL of the request can be treated like a path in the resource tree. This structure is also visible to some extent in the import graph of the API module as displayed in Figure 4.3. We will highlight and discuss some important files in the API module:

- *rest_manager*: the *rest_manager* file contains the *RESTManager* class which is responsible for starting and stopping the API. In addition, this file contains the *RESTRequest* class which is a subclass of *server.Request* (which in turn is instantiated by Twisted on an incoming request) and handles any exceptions that occurred during the serving of the HTTP request.
- *root_endpoint*: this file hosts the *RootEndpoint* class which represents the root node of our resource tree. This class dispatches incoming requests to the right sub nodes in the tree.
- *util*: the *util* file contains various helper functions, such as conversion utilities to easily transform channel and torrent data from the database into JSON format that can be sent to the client that initiated a request.

Except for some endpoints, all data returned by the API is structured in JSON format. The JSON format is well adopted in the field of web engineering and easy to parse. Most of the endpoints are straightforward implementations where the client performs a request and some data is returned. There are situations where the client does a request and a stream of data should be returned. For instance, this is the case when the user performs a search query. Sometimes, data should be returned to the client, even if the client did not ask for this data. When a crash in the Tribler core code occurred, the client should be notified of this crash and possibly warn the user that he or she should restart the application. For this purpose, an asynchronous events stream has been designed and created. Clients can open this event stream and interesting notifications are sent over this stream. All messages that are sent over the *events* connection are shown in Table 4.1.

The API is started in two stages. Just before starting Tribler, the *events* connection is activated. This connection is initialized earlier so we can send interesting updates of the upgrader to the client. When

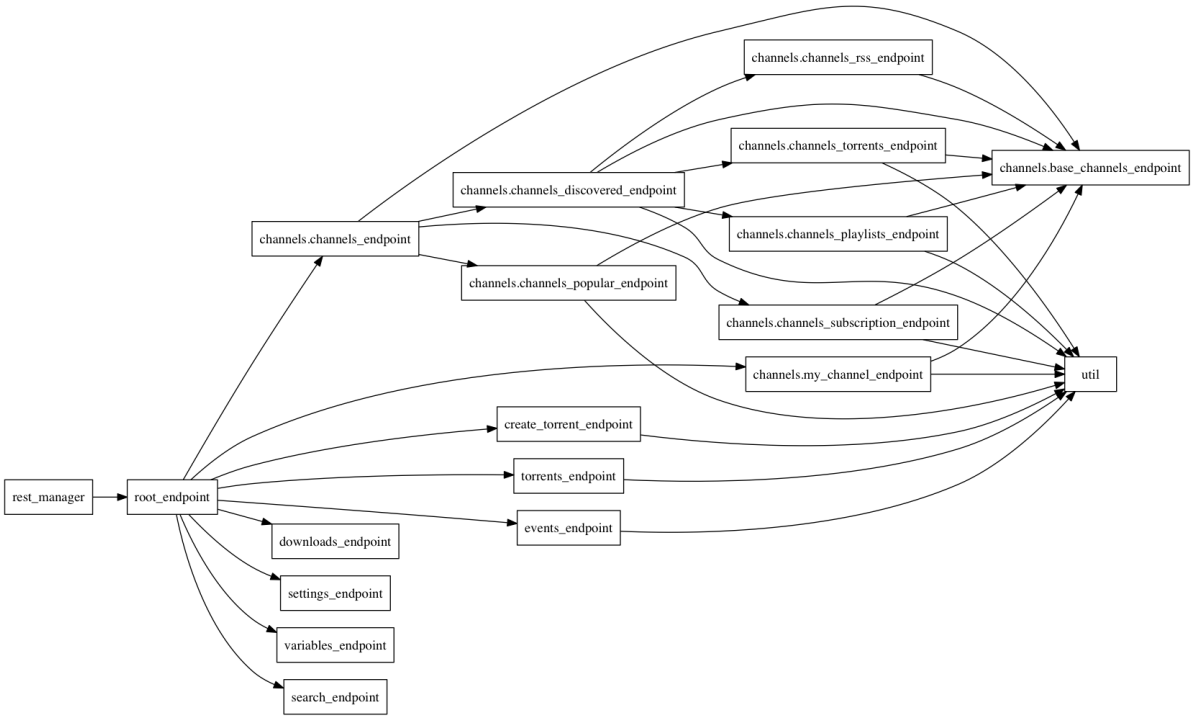


Figure 4.3: The import graph of the REST API module.

the upgrader is completed and Tribler has been started, the other endpoints are activated and the API is ready to be used.

4.3. Graphical User Interface

The amount of accumulated technical debt in the current graphical user interface of Tribler is devastating. After going through several development cycles where some impacting changes have been made, the code base of the user interface has reached the point where it is more beneficial to create a complete new user interface. This section will continue the discussion that has been initiated in Section 2. First, the structure of the current user interface will be described. Next, the new interface, introduced in Chapter 3 will be elaborated. Encountered design decisions and challenges are presented and discussed.

4.3.1. Analysis of the current user interface

The current user interface is unintuitive and cluttered with widgets and many functionality that should be in the core module of Tribler, can be found at higher levels in the user interface code. There is actually no clear separation between the core and the user interface, making it hard for developers to develop and test new functionalities since they have to deal with a bigger code base. One of the goals of this thesis is to make this separation more explicit for developers so they do not have to touch interface-related code if they are not interested in that.

The code base of the user interface is full of undesired workarounds and code smells. There is no clear, documented structure to be identified throughout the code and there are several reasons for this issue. One of the underlying causes is the mindset of developers that the user interface code base is subordinate to the code related to the core functionalities of Tribler. While it is often true that minor defects in the user interface are less critical than errors in important core functionalities such as the download engine, developer should always strive to write maintainable code. The fact that the user interface has undergone dramatic changes throughout the 10 years of research is an additional reason that led to this unstructured code base. Making short-term decisions were clearly favoured over decisions that benefit the longer-term development process.

Event name	Description
events_start	The events connection is opened and the server is ready to send events.
search_result_channel	Tribler received a channel search result (either remote or locally). The event contains the channel result data.
search_result_torrent	Tribler received a torrent search result (either remote or locally). The event contains the torrent result data.
upgrader_started	The Tribler upgrader started.
upgrader_tick	The status of the Tribler upgrader changed. This event contains a human-readable string with the status update.
upgrader_finished	The Tribler upgrader finished.
watch_folder_corrupt_torrent	The watch folder module has encountered a corrupt .torrent file. The name of the file is part of this emitted event.
new_version_available	A new version of Tribler is available. The version number is contained in the event.
tribler_started	Tribler has completed the startup procedure and is ready to serve HTTP requests on all endpoints.
channel_discovered	A new channel has been discovered. The events contains the discovered channel data.
torrent_discovered	A new torrent has been discovered. The events contains the discovered torrent data.

Table 4.1: An overview of all events that are passed over the asynchronous events connection, part of the REST API.

If we take a closer look at the structure of the user interface code base, several files with many class definitions can be found. In the *widgets.py* file, we can identify 30 distinct widget definitions. The *SearchGridManager* file acts like the glue between the Tribler core and the user interface. Most of the requests to the core are passing through method definitions inside this file.

4.3.2. Building a new user interface

Before starting to create a new user interface, we should first decide which graphical user interface library we would like to use. There are plenty of libraries that are suited for this job. Below, several of such libraries are summarized, together with a small description and some (dis)advantages.

- *wxPython*: this is the current user interface library we are using. *wxPython* is built upon *wxWidgets* and provides the Python bindings to this latter library. The library is cross-platform and we can continue to use *wxPython*. We already have a large code base written in *wxPython* so continued usage of this library could allow us to reuse several widgets. The main disadvantages of this library are the minor inconsistencies across different platforms and the lack of a visual designer, requiring us to specify the complete layout in Python code.

- *Kivy*: the cross-platform library Kivy has been used by Tribler research, particularly in the past attempts to run Tribler on Android[14][27]. The decision of using Kivy for the new user interface enables us to reuse the interface logic on Android. The layout of Kivy can either be created in *.kv* files or specified in code which is based on the separation of concerns principle. While not as old as *wxPython* or *pyQt*, the library has gained significant attention and adoption in the Python community.

- *TKinder*: the *TKinder* library is a layer built upon the Tcl/Tk framework and is considered the de-facto graphical user interface library for Python. Like the other discussed frameworks, *TKinder* does not provide a visual designer. The library is built-in in Python which means that no additional libraries have to be installed in order to start writing code. *TKinder* however is considered more suitable for simple applications due to the simplistic nature of the library.

- *PyQt*: *PyQt* provides the Python bindings for the Qt framework and is widely used in open-source and commercial applications. With a first version released in 1995, the Qt framework has evolved into a mature state. The library is very well documentation and provides many different addons

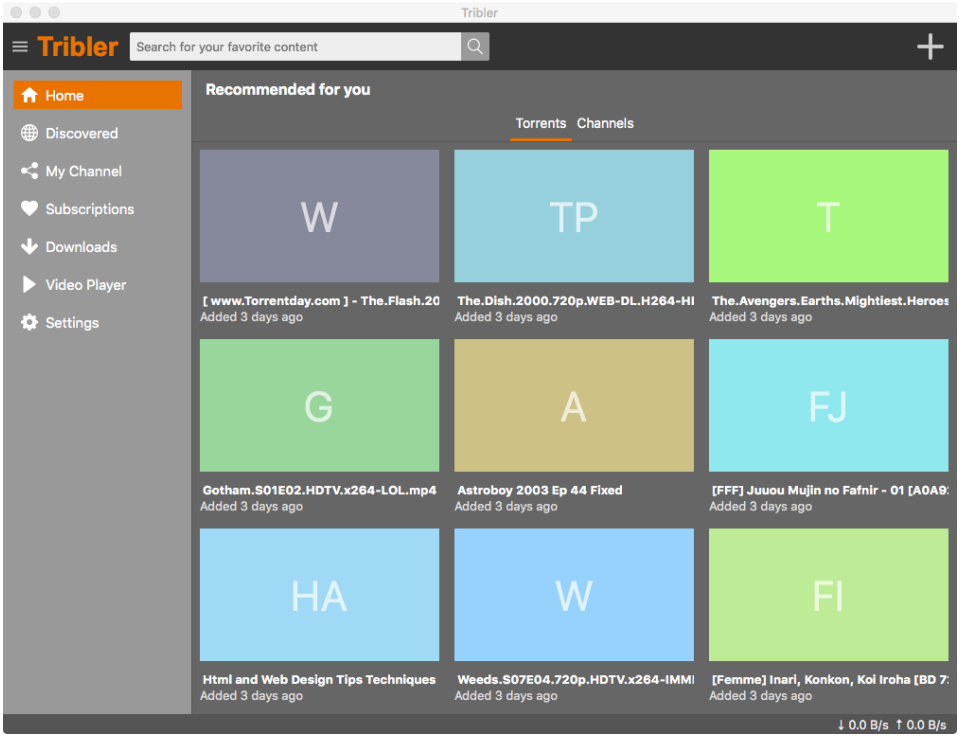


Figure 4.4: The home page of the new user interface.

and plugins to support a wide range of applications. One of these plugins is a visual WYSIWYG designer where the layout of an interface can be specified in a drag-and-drop manner. This generates a *xml* file which can be read and parsed by *Qt*. Visual styles can be specified using the *Cascading Style Sheet* (CSS) language. The documentation of *Qt* is very comprehensive, although the documentation regarding *PyQt* is somewhat less maintained.

Since the graphical user interface will be an important aspect of Tribler, we wish to use a library that is future-proof, well-maintained and easy to use. We think that in the context of this thesis, choosing *PyQt* is the best choice to build a new user interface. The fact that we can specify our layout using an editor is an enormous advantage since this will mean that we have less code to contribute. In addition, this allows other developers that are not familiar with the Tribler code base to contribute to the graphical user interface. The *Qt* visual designer also offers tools for internationalization and translation of the interface in foreign languages. Tasks like these are perfect opportunities for contributions in an open source project and can potentially attract new developers. A screenshot of the used visualizer is visible in Figure x.

The result of the development of the new interface is displayed in Figure 4.4. Our vision during development of this interface is that in essence, it should only display data and do as few processing of the data as possible. The majority of the new user interface has been built using the visual designer. Some code has been written to handle requests to the Tribler core, display the right data in lists and to manage interface-related settings. Connections between widgets and Python code are made using the signal-slot mechanism in *Qt*. This mechanism is used for communication between objects and is a central feature in the *Qt* framework. Widgets in *Qt* can have signals, events they want to possibly notify to other widgets. Some widgets have some built-in signals, i.e. a button emits a *clicked* signal if the user clicks the button with the mouse. Other widgets or Python objects can subscribe to these signals. These signals and slots can either be specified using code or the visual designer, however, to keep the amount of Python code to be maintained to a minimum, we decided to specify our widgets connections in the visual designer as much as possible.

During development of the user interface, some issues that required some careful analysis were en-

countered. The remained of the Section will describe these issues in more detail.

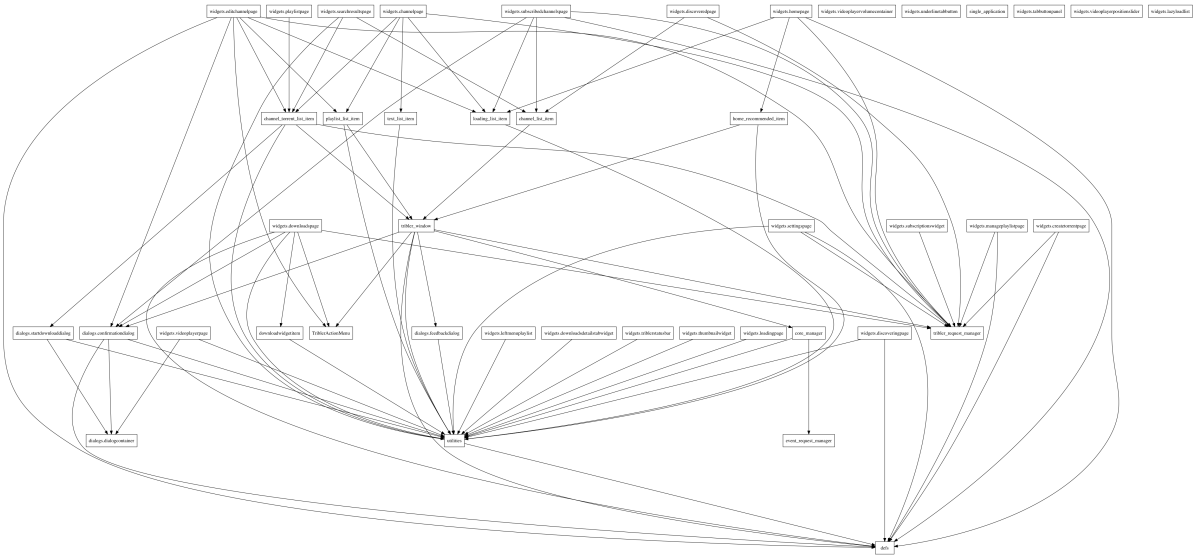


Figure 4.5: The generated import graph of the code base related to the new user interface.

Scalability of list items

Qt allows to display possibly many items in a simple list. The performance decreases dramatically if custom widgets are rendered in a list, like we are doing. Loading 1.000 of such list items takes over 22 seconds on a high-end iMac device, introducing an unacceptable amount of latency when displaying items. For each item in the list, the associated user interface file has to be loaded, parsed and rendered, possibly many times per second. Channels hold potentially several thousand of torrents which should be displayed in the user interface.

This problem has been solved by using two techniques. The first one is lazy loading: by using a lazy-loading approach where more data is loaded when the user has scrolled through the end of the list, we can postpone and possibly avoid loading the whole list at once. This solution has also been implemented in the interface written in *wxPython*. By loading only a subset of the list rows, the user experience can be significantly increased since users don't have to wait until the whole list of items is loaded. The implementation of this lazy-loading solution is reusable and can be found in the *lazyload-list.py* source file. This however, still resulted in a minor period of waiting when the next set of items is being loaded. It turned out that loading the interface definition file every time is a time-consuming operation. The implemented solution to reduce this processing time is to pre-load the interface definition as soon the user interface starts. This has only a minor effect on the total start-up time (around 40 milliseconds on a high-end iMac device).

A working VLC on multiple platform

The old user interface made use of the VLC library for the implementation of the embedded video player. This embedded video player did not function on OS X due to incompatibilities with the wx library used. We started the design of the new user interface by creating a prototype where the implementation of a cross-platform, embedded video player with support for starting and stopping a video is centrally involved. While example code was available for the PyQt4 library using VLC bindings, there were some minor quirks when implementing the video player using PyQt5, mostly involved around obtaining a reference to the frame of the video player (which should be done differently for each platform). The code for this player has been used as reference for the implementation of the video player in the new user interface of Tribler and is available on GitHub[13].

4.4. Improvements to the threading model and performance

The work as described in the last two sections, has led to a better and stable threading model. The complex threading model as used by Tribler 6 is vulnerable for bad code practices, leading to confusion and race conditions.

Developing the new user interface came with an additional advantage. Now that the user interface runs in a separate process, we have the opportunity to run the reactor on the main thread. This allows us to get rid of the confusing decorators since we only have one thread (besides the threadpool) to schedule calls on. Getting rid of the abundant thread switching should increase performance since we avoid overhead introduced by the context switch. The new threading model is visible in Figure 4.6.

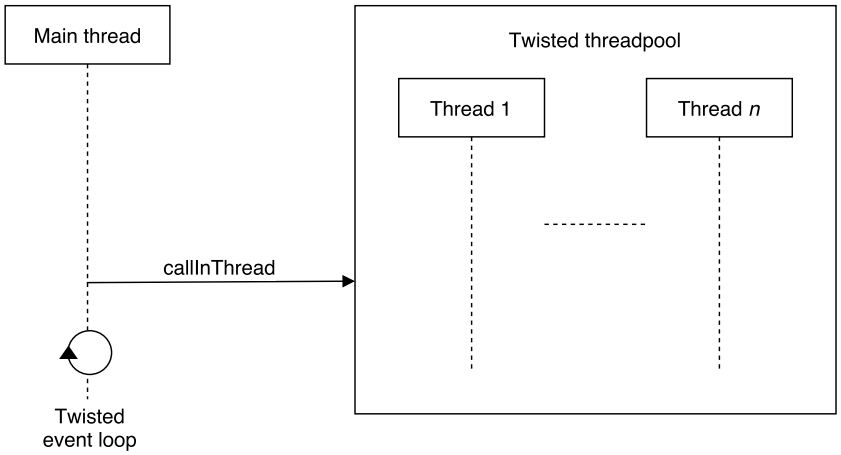


Figure 4.6: The new, simplified threading model in Tribler 7, together with the primitives to schedule operations on the threadpool.

4.5. Relevance ranking algorithm

Serving users relevant information as fast as possible is important in Tribler. When users are performing a search in the user interface, the returned search results are sorted according to a relevance ranking algorithm that considers several factors of the search results. A key problem of this algorithm is that the implementation is located inside the code base associated with the user interface. Essentially, sorting search results can be considered a task of the Tribler core. Moving the algorithm to the core module seems to be a adequate solution but this requires us to understand the old relevance ranking rules so we can reimplement the algorithm in the core module. Unfortunately, the code that is responsible for the relevance ranking lacks proper documentation and is hard to read and understand. Moreover, the code it split between several classes, making it harder to understand its behaviour.

4.5.1. Old ranking algorithm

Sorting of channels and torrents are both using a different algorithm. Channels are sorted on the number of torrents where channels that have a higher number of torrents, are displayed higher in the list. The algorithm to sort torrents on relevance is more complex and uses five different scores. These scores are determined as follows (ordered on importance):

1. The number of matching keywords in the name of the torrent. Keywords are determined by splitting the name of a torrent on non-alphabetical characters and common keywords such as *the* are filtered out.
2. The position of the lowest matching keyword in the torrent name. For instance, when searching for *One* and there is a torrent result named *Pioneer-One-S01E03.avi*, the position of the lowest matching keyword is 2, since *Pioneer* is not present in the search query.
3. The number of matching keywords in the file names that the torrent contains.
4. The number of matching keywords in the extension of the files (for instance, *avi*, *iso* etc).

5. A score that is based on several (normalised) attributes of the torrent. This score is determined after the set of local search results are constructed. To calculate this score, the following formula is used: $s = 0.8 * n_s - 0.1 * n_{vn} + 0.1 * n_{vp}$ where s is our score, n_s denotes the number of seeders (0 if this information is not available yet), n_{vn} the number of negatives votes of this torrent and n_{vp} the amount of positive votes this torrent has received. We should note that the number of positive and negative votes do not exist any more and as a consequence will always be 0, making this score only dependent on the number of seeders. The normalization process calculates the standard score for every data item, using the following formula:

$$z = \frac{x - \mu}{\sigma} \tag{4.1}$$

where z is our normalized score, x the score to be normalized, μ the mean of the data set and σ the standard deviation of the data set.

For each torrent, the set of five scores as described above is determined. The comparison between two torrents now proceeds based on these five determined scores, starting with the first score, proceeding to the next score in case when two scores are equal.

Finally, the list is prepared and a channel result is inserted between every five torrent items in the list. This is done since usually, the amount of torrents is much bigger than the amount of channels. Not only channels matching the search query are displayed: for each torrent, the most popular channel that contains this specific torrent, is determined and also considered in the list of results.

While the algorithm as described above takes many factors in consideration, we detected some problems and possible improvements:

- One of the main problem is that the amount of matches inside a torrent name/torrent file name is not taken into consideration. For instance, when searching for *Pioneer One*, a torrent named *Pioneer One Collection* probably has a higher relevance than a torrent named *Pioneer One - Episode 3, Season 4* since the matching in the first case is considered better.

- The relevance sorting of channels in the result set is only dependent on the number of torrents in that channel. The number of matching terms in the channel name and description is not even considered.

- When building the inverted index in the SQLite database for the full text search, duplicate words are removed. This means that when we search for *years*, a torrent named *best years* will be ranked equal to a torrent named *years and years* (if we only consider a ranking based on the torrent name). However, the torrent named *years and years* should be assigned a higher relevance since the keyword *years* occurs twice in the latter example. Another example is when searching for *iso*. A torrent file that contains 100 *iso* files is currently ranked equivalent to a torrent file that only has one *iso* file.

- The current relevance ranking algorithm only returns results that matches all given keywords. So when searching for *pirate audio*, only torrents are returned that are matching on both terms. It might be better to show the user also torrents matching 'pirate' and matching 'audio' (while still giving a higher relevance score to torrents that matches 'pirate audio').

- The ranking of search results are dependent on each other. This is noticeable when calculating the score based on normalized data. To normalize this data, we should have information about other search results. This prevents a "streaming-like" search operation where the relevance score of each search item is only dependent on data that search item contains and no other data.

4.5.2. Designing a new ranking algorithm

In the previous Subsection, we described the old ranking algorithm, together with some problems and improvements. In this Subsection, we will design a new, robust and simplified algorithm. The heart of the algorithm will be based on Okapi BM25, a ranking function used by search engines to rank matching

documents according to their relevance to a given search query[18]. BM25 can be implemented using the following formula:

$$s = \sum_{i=1}^{n} IDF(q_i) \frac{f(q_i, D)(k_1 + 1)}{f(q_i, D) + k_1(1 - b + b * \frac{|D|}{avgdl})} \tag{4.2}$$

In the formula above, we have a document D where the length of D is denoted as $|D|$. There are n keywords present in our search query, q_i representing the keyword at index i. $f(\cdot, D)$ gives the frequency of keyword q_i in document D. The $IDF(q_i)$ denotes the *inversed document frequency* of keyword q_i which basically states how important a keyword is in a document. The IDF is usually calculated as:

$$IDF(q_i) = log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \tag{4.3}$$

where N is the total number of documents and $n(q_i)$ is the number of documents containing keyword q_i. The full text search engine in SQLite offers tools to easily calculate a BM25 score when performing a query. Unfortunately, this is not implemented in the engine we are currently using, FTS3. This motivates us to upgrade to a newer engine, FTS4, which offers the necessary tools to easily calculate the BM25 score. This requires a one-time upgrade of the database engine of users which should be performed when Tribler starts.

Each search result is assigned a relevance score. The final relevance score assigned to a torrent is dependent on three other sub-scores that are calculated using the BM25 algorithm and is a weighted average of the sub-scores, determined by the name of the torrent (80%), the file names of the torrent (10%) and the file extensions of the torrent (10%). The final relevance score of a channel is the weighted average of the BM25 score of the name of the channel (80%) and the description of the channel (20%).

While this works well when searching for local search results, we should also be able to assign a relevance rank to incoming remote torrent or channel results. To do this, we keep track of the latest local searches and the gathered information that is used by Equations 4.2 and 4.3. If we receive an incoming search result, we are using that stored information to quickly determine the relevance score of the remote result. Using this approach, we avoid a lookup in the database for every incoming remote search result. If we have no information about the latest local database lookup available, we assign a relevance score of 0 to the remote result.

4.5.3. Ranking in the user interface

After each search result got a relevance score assigned, we should order the search results in the user interface. We cannot make the assumption that the data we receive from Tribler is already sorted (however, a relevance score should be available) thus we need a way to insert items dynamically in the list. The lazy-loading list we are using in the user interface makes this task more difficult since we both have to insert items dynamically in the list and make sure that we are not rendering too much row widgets. We also wish to avoid reordering operations as they are computational expensive to perform.

The implemented solution works as follows: in the user interface, we maintain two lists in memory: one list that contains the torrent search results and another list that contains channel search results. We guarantee that these lists are always sorted on relevance score. Insert operations in this list are performed using a binary search to determine the new position of the item in the sorted list, leading to a complexity of $O(log\ n)$ for each insert operation (where n is the number of items in the list). In the visible result list, we first display channels, which are usually only a few. The rationale behind this idea is that users prefer to see matching channels since these channels might contain many relevant torrents. This solution is scalable to many search results. The performance of the new relevance ranking algorithm is discussed in Chapter 6.6.

5

Improving Software Metrics and Refactoring

In Chapter 2, the high amount of technical debt Tribler has accumulated over the past decade has been highlighted. We presented a history of architectural evolution and proposed a new robust and future-proof architecture in Chapter 3. Now, we will start working on this architecture and the most impacting activities performed to improve the system during the span of this thesis work will be presented in this Chapter. This includes a accessible REST API, a modern, simplistic user interface, major modifications to the available testing framework and refactoring of several components present in Tribler. A summary of the re-engineering efforts conducted is displayed in Table 5.1.

Lines modified	765
Lines added	12.429
Lines deleted (without GUI)	12.581
Lines deleted (with GUI)	25.010

Table 5.1: A summary of refactoring efforts as conducted during this thesis work.

5.1. Software Aging

We start the discussing with introducing and discussing the term *software aging*, which is relevant to Tribler in a sense that the system has symptoms of aging. Software aging is a serious problem in a society that is dependent on all kinds of software. The term has been coined by Dave Parnas in his talk about software aging in 1994[23]. He points out two major reasons why software aging is a problem. The first reason is the lack of movement when software fails to meet the requirements of an always-changing environment: software that users perceived as state-of-the-art several decades ago, might now be considered as legacy software.

The second reason is in particular interesting since we think this is one of the most important reasons why Tribler has evolved to the complex, hard-to-understand architecture it is today and Parnas references to this phenomena as *ignorant surgery*. Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. This is especially true for the development process of Tribler which had many contributors who are making changes to specific parts of the architecture, often not knowing the design concepts of the original authors of the code. Moreover, some developers might not be satisfied with the implemented design and decide to change the architecture to suit their own needs.

Some of the consequences of aged software are reduced performance, decreased reliability and a system that does not function correctly in a modern environment.

afmaken

	November '15	July '16
Number of unit tests	x	x
Number of assertions	x	x
Number of failed runs after 10 runs	x	x
TLC/PLC ratio	x	x
Total Linux test duration on Jenkins	x	x
Average execution time per test	x	x

Table 5.2: A summary of improvements to the test suite between November '15 and July '16.

5.2. Influences of Python on software metrics

Tribler is written in: Python, an accessible, easy-to-learn language that is widely used in scientific applications. It is a high-level language, allowing one to express complex constructs with only a few lines of code. One of the most distinguishable properties of the language is that it is dynamically typed, which means that the type of a variable is not known at compile-time. This is in contrast to static typing, where this type is known to the compiler. The dynamic nature of the language has consequences on the way programmers are writing there code. A dangerous pitfall is that developers could make wrong assumptions about types of variables, leading to bugs that are only visible on runtime. Even then, it is not guaranteed that these kind of errors reveal themselves since they might be located inside a branch that is entered in a small amount of the application executions.

Dynamic typing also influences generated software metrics. While import graphs might give a good indicator of dependencies, they do not tell the whole story. In fact, there might be dependencies that are not visible in a generated import graph. These 'hidden' dependencies are often made between classes using *dependency injection* (DI), a technique to not denote dependencies in the source code. Dynamic typing makes it even harder to capture such dependencies since hardly any information about types of attributes within a class can be determined at compile-time, especially not when dependencies are passed through getters or the constructor of a particular class.

5.3. Improving the test suite

The most fundamental way to verify the correctness of software and to detect issues as soon as possible in the development cycles, is by having an well-designed and stable test suite. As pointed out in Chapter 2, the current test suite is plagued with unstable and non-functional tests. We will discuss the performed work to strengthen and stabilize the test suite in this Section. A summary of improvements made during this thesis is presented in Table 5.2.

5.3.1. Identifying code smells in the tests

As described in the work of van Deursen et al[30], there is a difference between refactoring test and production code in a sense that test code often has a characteristic set of code smells, originating from the way tests are structured. Before we start to make major modifications to the test suite, we present a list of code smells identified after a manual code review in the test suite of Tribler. This list is displayed in Table 5.3 where for each code smell, we describe it and propose a solution.

Table 5.3 has been used as reference during the refactoring efforts of the test suite and in this thesis work, we fixed various of the outlined code smells. Dependencies on external resources have been reduced to a minimum as explained in Subsection 5.3.4. The efforts on increasing the stability of the tests is outlined in Subsection 5.3.5. During the refactoring process of tests, we placed clear assertions, added comments and got rid of managing Tribler sessions as much as possible.

5.3.2. Improving Code Coverage

Code coverage is defined as the percentage of source code that is covered by at least one test in the test suite. Our continuous integration environment offers tools to track the code coverage over time. After each automated test suite execution a comprehensive report with detailed information about the code coverage is generated. The reported metrics by this report are not accurate enough since some

Code smell	Description	Solution
Dependencies on external resources	Various tests are using external resources, leading to unpredictable and unstable tests.	Remove the dependency on the resource or make sure that the resource is locally available (see Chapter 5.3.4).
State leak	The state of a previous executed test is leaking to the next test, mostly notable due to delayed calls left in the Twisted reactor after shut down.	Make sure that any delayed call in the reactor is removed when shutting down Tribler.
Too much responsibility	Many tests have multiple responsibilities, testing both parts of the user interface and core components in Tribler.	Make sure that each test is only verifying one unit in the system. Also implement a separate test suite for the user interface.
Tests with a high runtime	There are some tests that are taking long to complete (sometimes over 30 seconds). This can be an indicator that the test has too much responsibilities.	Identify why the test takes long to complete and shorten the runtime i.e. by splitting the larger test in smaller tests.
Unclear assertions	Tests that have multiple assertions often do not annotate their assertion well with a clear and meaningful description	Add an annotation with the cause of the failure if an assertion fails so developers can pinpoint the problem quicker.
Dependency on a Tribler session	Some tests are starting a complete Tribler session while only a small subset of the system is tested	Use mocking techniques to inject a mocked session or refactor the component so no session is required to test the component.
Resource writing to the source code directory	Various tests are writing resources to the source code directory. They might accidentally end up in the VCS if developers are not noticing these files.	Temporary resources produced by tests should always be written to a temporary directory that is cleaned after test execution.
Claiming the same local network port	Some tests that are running in parallel are claiming the same local network port, leading to test failures.	Reserve port ranges to individual parallel test runs or try to avoid the allocation of local ports.
Timing issues	Various tests are asserting a condition after a fixed time interval. This interval is often based on intuition rather than empirical data. This is particularly dangerous when the test is dependent on external resources.	Refactor the test so the condition check is no longer necessary.
No usage of comments	There are no comments, explaining what the the tests are testing and what the expected output is.	Comments should be added that explains the purpose of the test together with the expected in- and output.
No directory structure in the test package	There is no directory structure and almost all tests are located inside the same directory.	Restructure the tests package and organise tests in different, logical named directories.

Table 5.3: Identified code smells in the test suite of Tribler as of November ’15.

	November '15		July '16	
	Lines coverage	Branch coverage	Lines coverage	Branch coverage
Core	71,2%	58,1%	81,2%	67,3%
REST API	-	-	99,4%	92,7%
wx GUI	65,8%	42,7%	-	-
Qt GUI	-	-	70,0%	x

Table 5.4: The difference in code coverage between November '15 and July '16.

third-party libraries are included in the coverage report, such as the VLC bindings and *pymdht*, a library to fetch peers from the Distributed Hash Table (DHT). Also, the code coverage of Dispersy is included in these reports while we consider Dispersy as a separate engineering project.

The difference in code coverage during the span of this thesis can be found in Table 5.4. Branch coverage is a metric that specifies how well conditional statements are covered. This metric includes the fact that a conditional is either resolved to true or false, possibly influencing the program execution path. In the ideal scenario, we wish to have tests that cover all conditional statements in the case they resolve to *true* and in the case they resolve to *false* so we cover all possible execution paths in the program. This objective gets significantly harder to achieve when code with many nested conditional statements is written. The cyclomatic complexity as developed by McCabe in 1976[22] is a quantitative measure of the number of linear independent paths through a program's source code. Any conditional written has a negative effect on the cyclomatic complexity.

While at first sight it may look like the code coverage has not increased significantly, we should emphasize that the complete architecture of the tests have been overhauled in parallel. Refactoring of the test suite had consequences on the code coverage in other locations in the code base. For instance, the smaller unit tests are not starting the old user interface, leading to a lower coverage in that module.

Improving the code coverage has been done by writing small unit tests where we are using mocked objects to have better control over the system we are testing. Using mocking in Tribler is a necessary since some components have many other dependencies that are hard to keep under control without using custom, controlled objects. Libtorrent is a good example of this. During this thesis, many unit tests have been written as can be seen in Figure x where the number of unit tests over time is presented. Writing tests makes a developer more aware of the written code and can be a good way to get familiar with an unknown system. Due to this, various bugs have been solved during the process of writing additional tests.

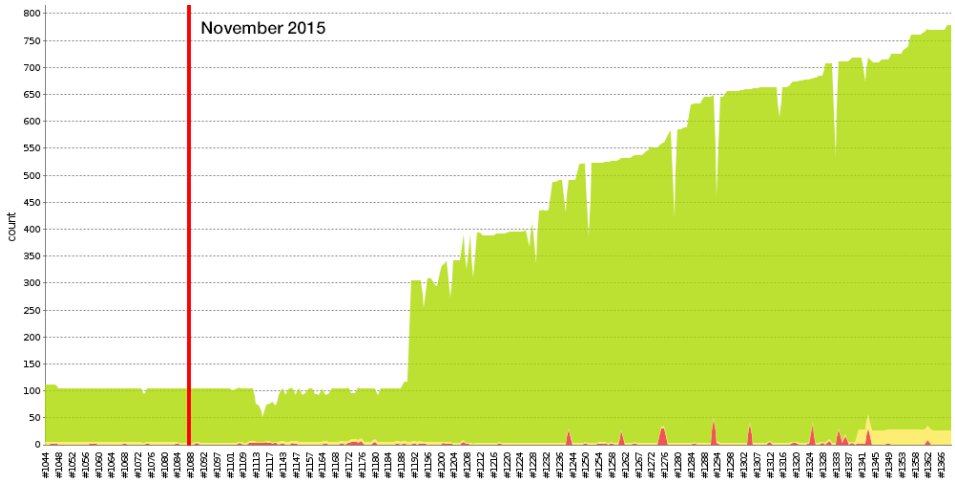


Figure 5.1: The number of tests over time where November 2015 is annotated.

In Chapter 2, Figure 2.5 we presented the ratio between the number of code lines in the tests package

and the amount of other code lines over time. Together with the code coverage, this number can be a useful metric to developers. While one might argue that having a high code coverage in conjunction with a low TLC/PLC ratio is a desired result, it indicates that the tests are not granular enough and are actually doing many different operations. A low code coverage with a high TLC/PLC ratio indicates that there are some flaws in the tests, possibly that they are testing the wrong components of the system. When starting the thesis, Tribler has a low code coverage together with a low TLC/PLC ratio. beter uitleggen

After writing additional unit tests, removal of the old user interface and addition of the new one, the new ratio is *0.16* which means that there is roughly one line of test code for six lines of other source code in Tribler. Defining a good TLC/PLC ratio is dependent on the used programming language, development methodology and application structure. Discussion on the wiki of Ward Cunningham[5] proposes an optional ratio of 1:1, however, several other ratios have been proposed on the same page such as 2:1 and 5:1. In the work described in [30], an ideal ratio of 1:1 is proposed. Overall, the trend seems to be that the amount of test line code is around the same or a bit higher than the lines of production code. An important question is whether this proposed ratio also holds for Tribler. Tribler differs from a commercial software engineering project in the sense that it is used for scientific research. When performing research, it is easy to ignore testing and focus on the results that are gathered by the system. The difficulty here is that Tribler is distributed and used by over a million of users, requiring at least some form of quality assurance. We think a better optional TLC/PLC ratio for the Tribler project might be 1:2.

To make sure that the responsibility of code coverage is not neglected in future work on Tribler, an addition check for each pull request has been added that verifies that the code contributed in the respective pull request is covered by tests. While not created by the author of this thesis, this check is an effective way to keep the code coverage metric under control.

5.3.3. Testing the New User Interface

One of the issues in the old testing framework, was that there is no clear separation between tests that are testing the user interface and tests that are testing core functionalities of Tribler. This is one of the reasons that have led to big, extensive tests in the old test suite. Since testing is an important aspect of this thesis work, writing proper tests for the user interface has been a prioritized task earlier in the development process of the new user interface.

GUI testing is an interesting area in the field of software engineering and is part of the application testing methodology. GUI testing can also be more involving than unit testing since a user interface might have many different operations. afmaken, cites et

Testing the new Qt user interface makes use of the *QTest* framework. This framework provides various tools to perform non-blocking waits and to simulate mouse clicks and keyboard actions. A sample of a test written with the *QTest* framework is illustrated in Listing 5.1. After the interface is started, the test navigates to the home page, clicks on the *channels* tab button and waits for items to be loaded. During the test execution, two screenshots are captured, one when we are loading items and another one when the requested items are loaded and displayed.

Primitives to capture screenshots during test execution has already been implemented and used in the old test suite, using the rendering engine of *wxPython*. The *Qt* frameworks offers similar tools. Captured screenshots are exported to *jpg* files under a name specified by the developer. In the sample given in Listing 5.1, the exported screenshots are saved as *screenshot_home_page_channels_loading.jpg* and *screenshot_home_page_channels.jpg* respectively. At the end of each test run, an image gallery is generated where the generated screenshots are archived and displayed in a grid. This allows developers to manually verify whether visual elements are correctly displayed. A part of the generated image gallery is displayed in Figure 5.2.

To avoid dependencies on Tribler itself and thus re-introducing the problem we are trying to solve, we created a small piece of software that provides the same interface as the REST API implemented. This 'fake' API is much simpler in nature and has a very simplistic in-memory data model. The downside of this approach is that new endpoints have to be written twice, once in Tribler and once in this fake API,

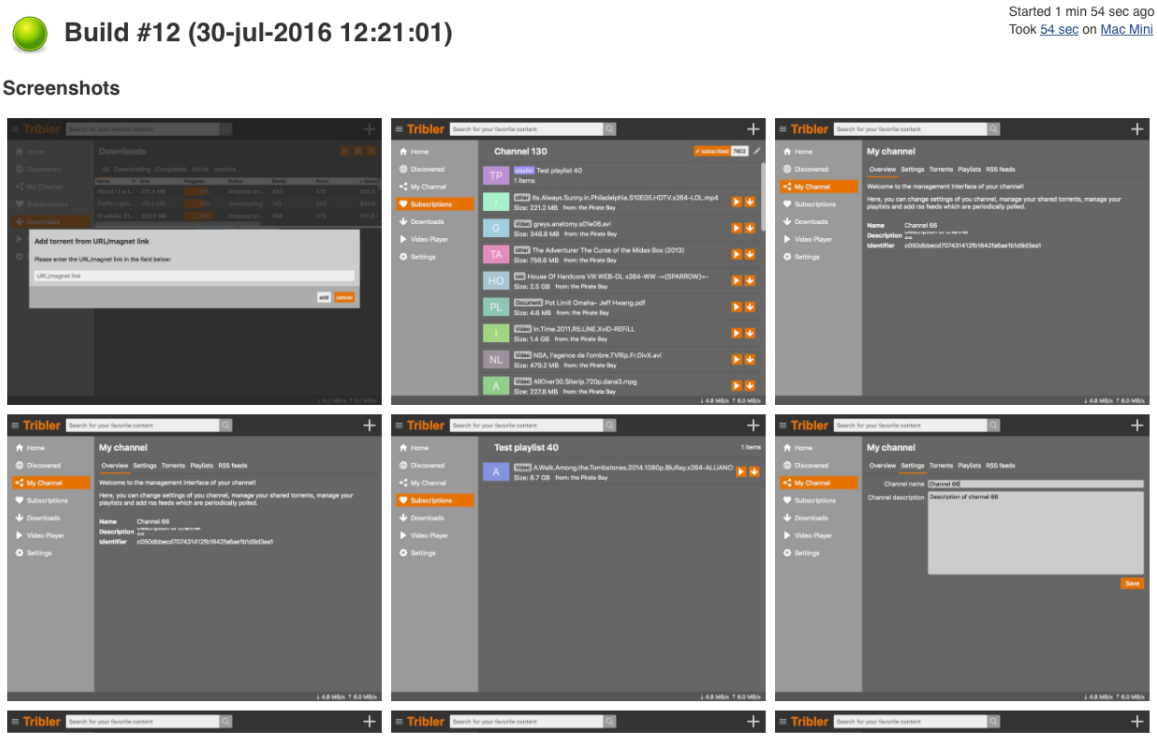


Figure 5.2: The generated image gallery after executing of the user interface tests, generated by Jenkins.

providing that the new endpoint will covered by a user interface test.

Listing 5.1: A sample of a test that tests the new Qt Tribler GUI.

```
def test_home_page_channels(self):
        QTest.mouseClick(window.left_menu_button_home, Qt.LeftButton)
        QTest.mouseClick(window.home_tab_channels_button, Qt.LeftButton)
        self.screenshot(window, name="home_page_channels_loading")
        self.wait_for_home_page_table_populated()
        self.screenshot(window, name="home_page_channels")
```

5.3.4. External Network Resources

On of the problems with the test suite was that dependencies on external network resources should either be removed or one should verify that the resources are under the control of the developer and always available. The test suite contains various tests where external torrent files are fetched from the internet, in particular, from the Ubuntu repository. While this repository can guarantee high availability, any downtime in this external resource can lead to failing tests. The implemented solution for this design flaw is to start up a local HTTP server that serves the torrent file. While this approach requires more code to manage this local server, it completely removes the dependency on the Ubuntu repository.

The same solution has been applied to solve the dependency on external seeders. A small number of tests makes assumptions on the availability of torrent pieces of the network. This certainly makes tests fail if the executing machines has a bad or even no internet connection. The process of setting up a local seeder session is straightforward. Again, this approach requires code to properly start and shut down the seeder session. The implementation is reusable to an extend that developers of tests can reuse the implemented solutions with only a few lines of code.

Unfortunately, there are some external network dependencies left which are considered harder to refactor. A handful of tests are performing a remote keyword search, requiring various communities in Dispersy to be loaded. These tests are dependent on available peers in the respective community in order to make sure there are incoming search results. The proposed solution here is to start various

dedicated Dispersy sessions on the local machine. Due to time constraints, the implementation of this solution is considered future work.

5.3.5. Instability of Tests

Well-designed tests should only fail if some new code is breaking existing functionality. If no changes are presents, the tests should always succeed. Reducing dependencies on external network is not sufficient to guarantee this in Tribler. The structural problem of the tests is that the system is infested with race conditions. Race conditions can be invisible since they often occur in a very specific runtime setting of the system, making the debugging process of these kind of errors frustrating.

During this thesis, many race conditions have been detected and solved. One interesting observation is that some issues only occurred on a specific platform. We believe can be explained by differences in the implementation of underlying threading model across operating systems. The most common cause of failing tests can be addressed to delayed calls in the Twisted reactor. During the test execution, Tribler is restarted many times. If a developer leaves by accident a delayed call behind when the shut down procedure has been completed, this delayed call might be executed in the wrong Tribler session, possibly leading to an inconsistent state of the system. Making sure the reactor is completely clean is not straightforward: if one is not aware of scheduled calls in the system, the mistake is easily made.

Writing stable tests also requires the test to be limited in what they do. Each test should only be focussed on the specific part of the system that has to be tested. While often unnecessary, a significant amount of the available tests are focused on starting a complete Tribler session, testing a small subset of the system, and shutting down Tribler again. While this approach is relatively easy to code, starting a fully-fledged session often leads to more instability and unexpected side-effects during test execution. Instead, only the classes to be tested should be instantiated and any dependencies this class have, should be mocked. Mocking ensures that developers have control over dependencies, allowing them to specify any expected return value. Moreover, the execution time of these small unit tests is significantly lower than the tests where a Tribler session is managed. The additional unit tests that have been written during this thesis, are following the described design.

cite?

5.3.6. Continuous Integration Enhancements

Tribler makes use of the popular continuous integration (CI) platform Jenkins. Jenkins allows developers to define jobs which can be executed manually or when pushing a commit on the code base. This continuous integration platform is responsible for running the tests, packaging Tribler and running research-oriented experiments.

When we focus on the execution of the tests, it is immediately noticed that they are only executed in a Linux environment. Beller et al[9] conducted research on CI adoption and usage and it turned out that for some languages, it might benefit to run tests in different environment. An addition argument for this is the usage of many platform-specific workarounds we are using in Tribler. To make sure that these statements are covered, we must run the tests in this environment. This will allow developers to detect defects on other platforms more earlier in the development process. By aggregating the generated coverage report on each platform, this multi-platform setup should benefit the code coverage.

The setup of the testing environments on Windows and OS X is straightforward. New slave nodes to specify the Windows and OS X test runners have been created. The tests on OS X are executed on a Mac Mini. In order to run the tests on Windows, two virtual machines using Proxmox have been created, both 32-bit and 64-bit environments. In total, the tests are executed on four platforms now: Linux, Windows 32-bit, Windows 64-bit and OS X. So far, the OS X and Windows test executers have completed over 2.500 test runs. Each test runner generates a coverage reports and these reports are merged in the final analyse step in the build pipeline.

specs

While this is certainly a step in the right direction, there are various additional steps in the execution plan that can be performed. In Figure 5.3, the ideal test execution plan is displayed, together with the various stages in this pipeline. We start by executing the tests on multiple platforms where during these runs, the code coverage is being tracked. After this phase, the coverage reports are combined

and the total difference with the upstream branch is determined. When the commit decreases the total code coverage, the job fails.

A static Pylint checker has been available to check for code style violations, however this only gave insight in the total amount of Pylint errors and did not stimulate to actually fix errors in the committed code. While not implemented by the author of this thesis, the Pylint checker has been extended to fail if new violations are introduced in the committed code. Additionally, a report is generated with an overview of the introduced violations. This helps developers to get more aware of the code style. This checker is ran in parallel with the tests to decrease the total time of the pipeline execution.

After the coverage phase has passed, jobs to package Tribler for distribution should be added. In parallel, the *AllChannel* experiment can be executed. This experiment is executed on the DAS5 supercomputer and starts 1.000 Tribler clients that are synchronizing torrents and channels with each other. When the experiment is completed, several graphs are generated, providing developers insights in the consequences of their modified code when Tribler runs in a more comprehensive environment. For instance, the experiment can highlight issues in the message synchronization between peers in the network.

In parallel with the *AllChannel* experiment, we can package Tribler for distribution to end-users. On Windows, an installer will be created that installs Tribler to the *Program Files*. On OS X, we create a *DMG* file that contains an app bundle. On Linux, the required files are bundled in a *deb* archive. All these jobs can be executed in parallel. Finally, we should test whether the final distributions are working. This should be achieved by executing the final Tribler executable. A small test suite that makes use of the REST API can be created.

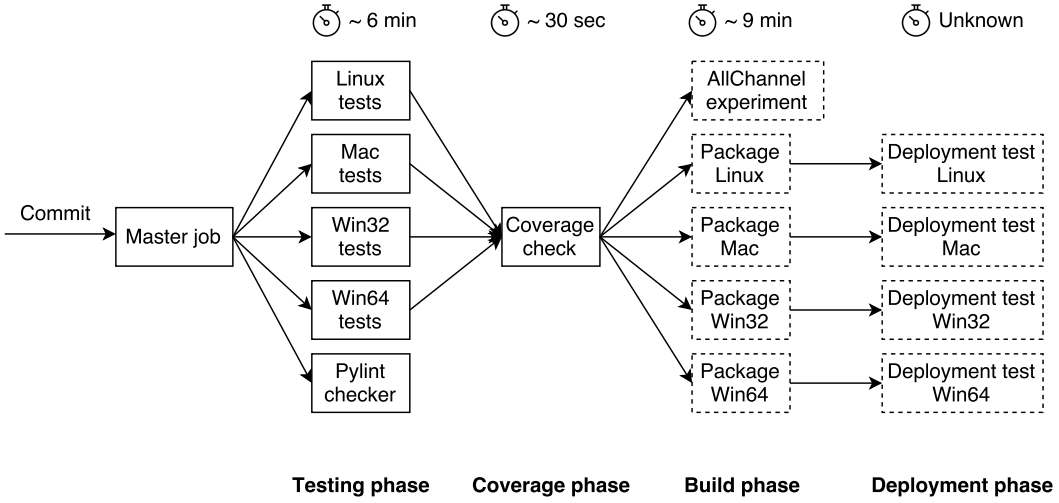


Figure 5.3: The desired test execution plan in our continuous integration environment. Dashed boxes are not implemented yet.

5.4. Updating software dependencies

A cause of aging software is the inability of developers to adopt to changing environment. This might be addressed to adoption of dependencies in the past, dependencies that are not maintained any more at a point in the future. Replacing these dependencies might be a non-trivial programming task, requiring the programmer to get familiar with both the interface of the old and new dependency.

Tribler has a long list of dependencies, both in the code base and dependencies that are used for software packaging and testing. Keeping these dependencies on the latest version is often neglected or overlooked. Sometimes, this is not even possible, due to missing software in package managers such as *yum*, the package manager used by CentOS or *apt*, the package manager of Debian and Ubuntu. While this restriction holds for operating systems where we are not packaging dependencies,

we have the freedom to package any dependency we want on Windows and MacOS so preferably, we often want to ship the latest, stable release of a dependency in our Tribler distribution.

During this thesis, we updated several dependencies to newer versions, most notably *libtorrent*. The code to handle the communication with this library (located in the *Libtorrent* package) contained calls to deprecated functions in the *libtorrent* library, functions that are not guaranteed to be maintained or compatible. We identified these calls as follows: first, a version of *libtorrent* has been compiled without deprecated methods and assertions. Next, we manually ran Tribler and the test suite, observing any crash due to libtorrent. In total, this method yielded seven calls to deprecated methods that have been updated to call the correct function. Not only deprecated calls have been removed, we also fixed various assertions that were triggering due to incorrect assumptions made in the code. In order to remain backwards-compatible with older version of the libtorrent library (that some Ubuntu or Debian users might have installed on their system), some checks in the Tribler code base had to be implemented to check for the presence of a particular method in libtorrent.

An additional outdated dependency in Tribler is *py2exe*, used to create a Windows executable file out of the source code. *py2exe* performs a static code analysis and determines code dependencies that should be bundled in the executable. Unfortunately, the library has not been updated since 2014 and requires a significant amount of code to make sure that everything works when Tribler is packaged and archived. We made attempts to replace *py2exe* with the more mature, well-maintained *PyInstaller* that also offers support for *PyQt5*, the framework used for creation of the new interface. This library is not only easier to use, it also works across multiple platforms, allowing us to also drop the *py2app* dependency which is used to distribute Tribler on MacOS. While not ready for deployment yet, a proof-of-concept has been created that successfully packages Tribler together with the new user interface into Windows and MacOS executable files. Further work should focus on the removal of *py2exe* and *py2app* in favour of *PyInstaller*.

5.5. Improving Sofware Artifacts

During the last ten years of development on Tribler, the main focus of the project has been to deliver working code. The project had a severe lack of maintained software artifacts, including documentation, comments and architectural diagrams. Most of the conducted research was documented on the Tribler wiki[1], however, this wiki is very outdated and not maintained anymore. After the migration of the project to GitHub, this platform was favoured over continued usage of the Tribler wiki archive.

At the moment, there are several distinct locations where we store the few software artifacts we have. Documentation is either stored in the GitHub wiki or in the ‘docs‘ directory in the Tribler source code. The ideal situation is to have one single, useful location for all generated documents during the process. Many Python projects are using *readthedocs*, a platform to host documentation for free. The hosted documentation should be located in the Tribler repository, in *reStructuredText* (RST) format. By using the Python module Sphinx, a HTML website can be generated from all the available documentation. Sphinx also provides tool for localization of documentation.

During this thesis, all available documentation of Tribler has been rewritten to make use of RST in conjunction with *Sphinx*. Moreover, the available documentation has been improved with several guides, in particular, guides that help users to create a developer environment on their machine. Prior, these guides were not available and development on other platforms than Linux was not supported. By the addition of these guides, new developers can start as soon as possible with development on Tribler.

The REST API in particular has been very well documented. Since external developers can use the REST API to control Tribler, we wish to provide a clear and comprehensive documentation base for this API. To simplify the process of writing documentation, the documentation can be written as doc strings above each method. The *autosummary* tool that is executed each time the documentation is built, navigates through the API code base, extracts all doc strings and generates page sections for each documented method. The doc string can also be attributed with *RST* syntax. This feature decreases

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

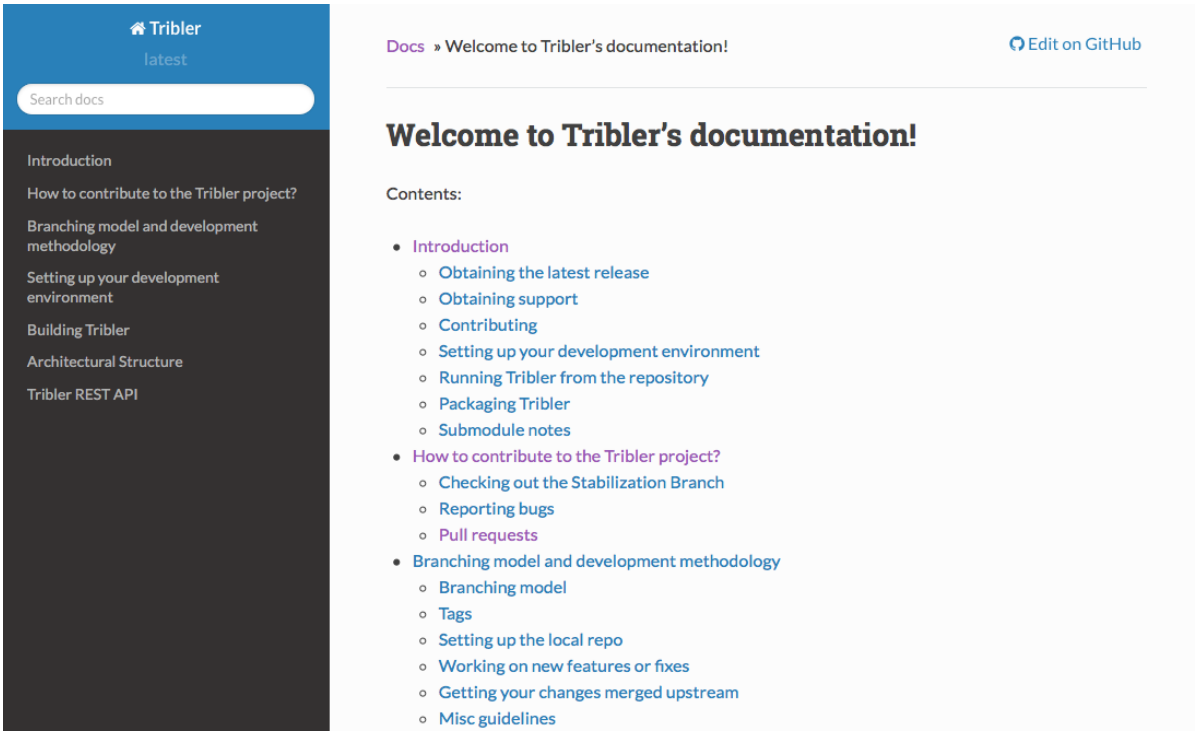


Figure 5.4: The documentation of Tribler, as displayed on the *readthedocs* website.

chances that developers accidentally forget to write documentation since the code and documentation is present in the same file instead of being spread across different files.

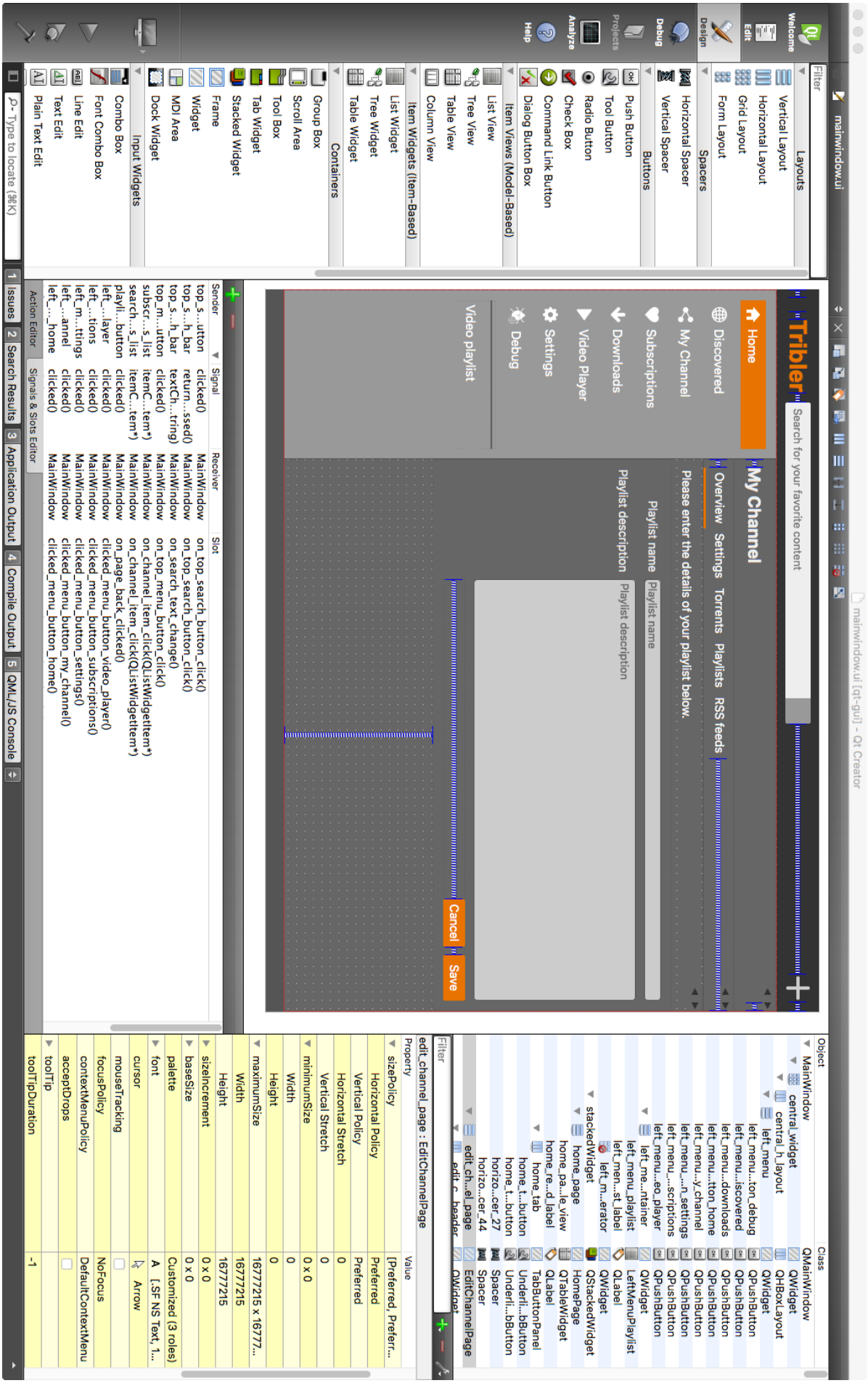


Figure 5.5: The Qt visualizer tool used to create the new user interface of Tribler.

6

Benchmarking and Performance Evaluation

The goal of this Chapter is to quantify the performance of various common operations performed by users in Tribler. We will perform a number of experiments and for each experiment, we will present and discuss the observed results. The underlying reason for this experiment is twofold: on the one hand, we would like to show that the performance of the system did not degrade with to unacceptable extent. On the other hand, we use the performance measurements to identify possible issues that is classified as future work.

6.1. Environment specifications

The experiments performed in this Chapter using the Tribler core are executed on a virtual private server. An important condition is to stay as close as possible to the specifications of a machine that an actual user could be using. The virtual server has 8GB of memory and 1 processor with 4 cores (where each core has a clock speed of 2.5GHz). The used operation system is Ubuntu 15.10. The experiments are not executed in an isolated, artificial environment but instead in the wild, using the deployed Dispersy network. While the obtained results may be different between users, this setup can be used to get insight in the performance of Tribler from a user's perspective.

If not stated otherwise, the default values of the Tribler configuration file are used. These default values can be found in the 'defaults.py' file in the source code directory of Tribler[1]. In this configuration file, all communities, except for *BarterCast*, are joined.

Some of the experiments are specified by the usage of a scenario file. In such a scenario file each line specifies a specific command of a peer at a specific point in time into the experiment. Our framework to run the experiments, Gumby, contains code to read scenario files, interprets the commands to be executed and to schedule them on the reactor thread. Several utility methods have been implemented to gather and write statistics to files in a processable and readable format that can be parsed by visualization tools such as *R*. The usage of scenario files is already adopted in various Dispersy experiments, mainly in our *AllChannel* experiment that runs on the DAS5 supercomputer. We extended the usability of this approach to run a Tribler client and we improved the framework by adding various commands to support operations as conducted in the performed experiments in this Section. An overview of the implemented commands can be found in Appendix A. The flexibility of these scenario files gives the next generation of Tribler developers a robust framework to use when conducting performance analysis, scientific research and benchmarking.

tabel met comma

[1] https://github.com/Tribler/tribler/blob/devel/Tribler/Core/defaults.py

6.2. Profiling Tribler on low-end devices

The addition of a REST API allows developers an option to run and control Tribler remotely using the HTTP API. For instance, one can run Tribler on a low-end, cheap devices such as a Raspberry Pi. Android is another example of a device that can run Tribler and during the last years, various research have been conducted to explore the possibilities of Tribler on Android devices[27][14]. Executing and profiling Tribler on a low-end device with limited resources can yield much information about bottlenecks that might not be directly visible when running Tribler on a desktop or supercomputer.

The experiments described in this Section are all executed on a Raspberry Pi, third generation with 1GB LPDDR2 ram, 4× ARM Cortex-A53, 1.2GHz CPU and 16GB storage on a microSD card. The used operating system is Raspbian, a system specifically designed for the Raspberry Pi and derived from the desktop Debian operating system.

Regular usage of Tribler on the Raspberry Pi using the REST API has us suspected that the Raspberry Pi is under heavy load when running Tribler. Monitoring the process for a while using the *top* tool, reveals that the CPU usage is often around 100%, completely filling up one CPU core. To get a detailed breakdown of execution time per method in the code base, the Yappi profiler has been used to gather statistics about the runtime of methods. This profiler has been integrated in the *twistd* plugin and can be started together with Tribler by passing a flag. The output generated by the profiler is a *callgrind* file that should be loaded and analysed by third party software. The breakdown of a 20-minute run is visible in Figure 6.1. This breakdown is generated using *QCacheGrind*, a *callgrind* file visualizer. In this experiment, we start with a clean state directory which is equivalent to the first boot of Tribler.

Incl.	Self	Called	Function	Location
131.47	1.22	340 246	wrapper /home/pi/Documents/tribler/Tribler/dispersy/util.p…	util.py
98.28	0.00	(0)	EPollReactor.run /usr/local/lib/python2.7/dist-packages/tw…	base.py
98.14	0.01	1	EPollReactor.mainLoop /usr/local/lib/python2.7/dist-pack…	base.py
98.06	0.05	5 132	EPollReactor.runUntilCurrent /usr/local/lib/python2.7/dist-…	base.py
90.33	0.21	9 635	AllChannelCommunity._on_batch_cache /home/pi/Docu…	community.py
79.84	0.22	13 050	AllChannelCommunity.on_incoming_packets /home/pi/Do…	community.py
66.00	0.06	9 596	StandaloneEndpoint.dispersythread_data_came_in /hom…	endpoint.py
65.49	0.11	9 665	Dispersy.on_incoming_packets /home/pi/Documents/tribl…	dispersy.py
57.93	0.74	59 406	DiscoveryConversion.decode_message /home/pi/Docum…	conversion.py
51.27	0.31	9 508	AllChannelCommunity.on_messages /home/pi/Document…	community.py
47.22	0.04	34 523	Implementation.has_valid_signature_for /home/pi/Docum…	authentication.py
47.18	0.11	34 523	Member.verify /home/pi/Documents/tribler/Tribler/dispers…	member.py
47.02	0.10	34 523	ECCrypto.is_valid_signature /home/pi/Documents/tribler/…	crypto.py
46.23	0.20	33 908	M2CryptoSK.verify /home/pi/Documents/tribler/Tribler/dis…	crypto.py
45.89	0.05	33 908	EC.verify_dsa /usr/lib/python2.7/dist-packages/M2Crypto/…	EC.py
45.81	45.81	33 908	ecdsa_verify M2Crypto.__m2crypto:0	M2Crypto.__m2crypto
26.76	3.52	5 987	AllChannelCommunity._resume_delayed /home/pi/Docu…	community.py

Figure 6.1: The breakdown of a 20-minute run of Tribler on the Raspberry Pi.

The file created by the Yappi profiler provides a detailed overview of the execution time of methods and can be used as a tool to detect performance bottlenecks in the system. Referring to Figure 6.1, the column *Incl.* denotes the inclusive cost of the function, in other words, the execution time of function itself and all the functions it calls. The column *self* denotes only the execution time of the function itself, without considering callees. The other columns are self-explanatory and can be used as reference to the location of the respective function in the source code.

If we analyse the breakdown, it is clear that Dispersy has a big impact on the performance of Tribler when running on the Raspberry Pi. The *ecdsa_verify* method (second method from the bottom) is dominating the runtime of Tribler: 45.81% of the time Tribler is running, is spent inside this method. This specific method verifies the signature of an incoming Dispersy message and is invoked every time a signed message is received. Disabling cryptographic verification of incoming messages should improve the situation, however, this is a trade-off between security and performance: by not verifying incoming messages, fake messages by an adversary can be forged and are accepted in such as system.

To verify whether the responsiveness of Tribler improves when we disable cryptographic verification of incoming messages, we measure the CPU usage of two runs. Both runs start with a non-existing Tribler state directory and have a duration of ten minutes. In the first run, we are using the default

configuration of Tribler, like in most of the other experiments described in this Chapter. In the second run, we disable verification of incoming messages. The CPU utilization over time of the two runs are displayed in Figure 6.2.

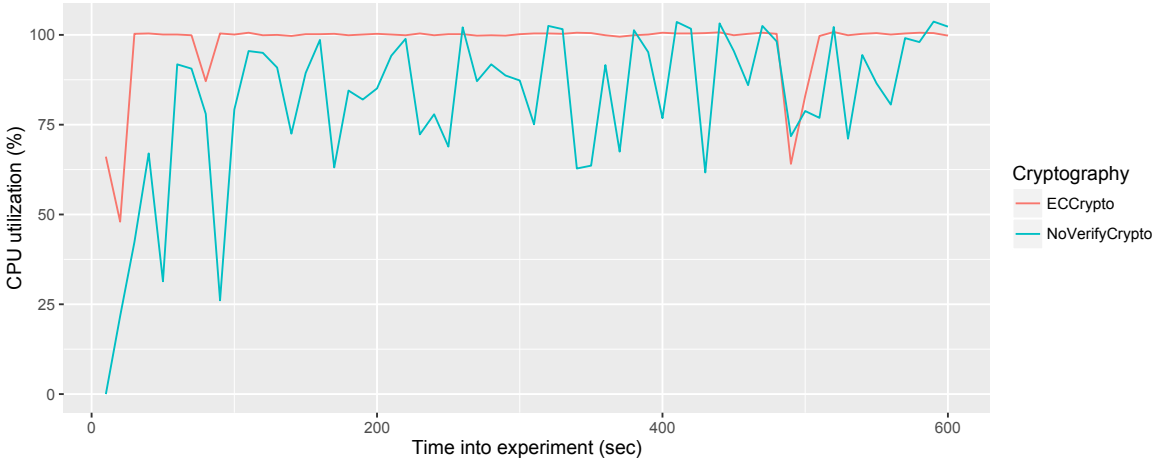


Figure 6.2: The CPU utilization of one core on a Raspberry Pi device when running Tribler with different cryptographic policies.

In Figure 6.2, some occurrences are visible where the CPU usage appears to be slightly over 100%. This is explained by the fact that some of the underlying code is designed to run on multiple processors. While the threading model of Tribler is limited to a single core, the Python interpreter might execute code on additional cores to improve performance. In the run where we enable all components of the system, the CPU usage is often 100%. When verification of Dispersy messages is disabled, we observe a somewhat lower CPU usage but overall, this utilization is still high. Unfortunately, disabling incoming message verification is clearly not enough to guarantee a more usable and responsive system.

To detect other performance bottlenecks, we sort the report generated by the Yappi profiler on the *Self* column to get insight in methods that are taking a long time to complete. This is visible in Figure 6.3. An interesting observation here is that the Python built-in *all* method takes up a significant amount of time (6.13% of the runtime). The *all* method takes an iterable object and returns *true* if all objects of this collections are true. Both the *all* method and *zip* method (also visible in Figure 6.3) is used in the *_resume_delayed* method, indicating that this method might causing performance issues. Since further analysis of this method requires more knowledge of Dispersy, optimization of this bottleneck is considered future work and described in GitHub issue 505[2].

Incl.	Self	Called	Function	Location
45.81	45.81	33 908	ecdsa_verify M2Crypto.__m2crypto:0	M2Crypto.__m2crypto
26.76	3.52	5 987	AllChannelCommunity._resume_delayed /home/pi/Documents/trib…	community.py
6.13	2.86	3 013 394	all __builtin__:0	__builtin__
0.00	2.61	454	isinstance __builtin__:0	__builtin__
1.73	1.73	6 773 230	<genexpr> /home/pi/Documents/tribler/Tribler/dispersy/community…	community.py
1.51	1.50	1 607	ecdsa_sign M2Crypto.__m2crypto:0	M2Crypto.__m2crypto
131.47	1.22	340 246	wrapper /home/pi/Documents/tribler/Tribler/dispersy/util.py:145	util.py
1.20	1.05	340 246	<method 'format' of 'unicode' objects> __builtin__:0	__builtin__
0.96	0.96	2 380 444	zip __builtin__:0	__builtin__
10.56	0.89	67 526	Implementation.__init__ /home/pi/Documents/tribler/Tribler/disper…	message.py

Figure 6.3: The breakdown of a 20-minute run of Tribler on the Raspberry Pi, sorted on the *Self* column.

In this Section, we demonstrated how adequate usage the Yappi profiler can lead to the detection of bottlenecks in the system. Integration in the twistd plugin makes it convenient for developers to run

[2]https://github.com/Tribler/dispersy/issues/505

Requests/sec	Avg. (ms)	Std. dev (ms)	Median (ms)	Min. (ms)	Max. (ms)	KB/S
1	241	476.34	76	56	4246	585.58
2	170	327.86	68	58	3394	1127.04
5	123	210.23	60	52	2082	2538.36
10	115	238.72	60	50	2450	4120.70
15	182	497.61	68	52	4937	3296.45

Table 6.1: A summary of the experimental results when measuring the performance of the REST API.

and analyse Tribler sessions under different circumstances.

6.3. Performance of the REST API

The responsiveness of the REST API is directly influencing the user experience. If the response times of API calls is high, users of Tribler have to wait longer before their data is available and visible. For this reason, we wish to make the API serve requests as fast as possible. The experiments to assess the performance of the API will particularly focus on latency of requests, however, some other statistics will be considered such as average request time, standard deviation of the response times and observed bandwidth. These statistics will help us to get more insights in the performance of the REST API.

We make use of Apache JMeter[17] that is used to perform HTTP requests to Tribler and to gather and process performance statistics. The application allows to simulate a realistic user load, however, in this experiment we will limit the load to one user that performs a request to Tribler with a fixed interval. This request will be targeted to a specific endpoint in the API: */channels/discovered*. This exact call happens when users are pressing the *discover* menu button in the new Qt GUI and the response of the request contains a JSON-encoded dictionary of all discovered channels that Tribler has discovered. As a consequence, the returned response can be rather large, especially if Tribler has been running for a long time and has discovered many channels (in our experiments, the average response size is around 613KB). When Tribler is processing the request, a database lookup is performed to fetch all channels that are stored.

We perform various experiments with different intervals between requests made and a fixed total amount of 500 requests. First, we perform the experiment with one request every second and we expect that the system should easily be able to hand this load and serve these requests in a timely matter. Next, the frequency of requests is increased to respectively 2, 5, 10 and 15 requests per second. These numbers are determined emperically and are based on the average request time, which appears to be several hundred milliseconds. Each experiment is started around five seconds after Tribler has started where we are using a pre-filled database with around 100.000 discovered torrents and 1.200 channels. A summary of the results of these experiments are visible in Table 6.1.

The most interesting observation in this Table is that it appears that requests are completed faster if we are performing requests at a faster rate, indicating that Tribler is able to handle the incoming requests well. This is surprising since one would expect the results to be the other way around: when the frequency of requests is increased, the average request time increases since Tribler might receive incoming requests while the previous incoming request is still being processed. The observed result is most likely due to caching of data which might be performed by the underlying database engine.

The standard deviation of the request times in Table 6.1 is rather large compared to the average request time. We suspect that this can be explained by the fact that Tribler is performing many different operations that are influencing the request times. In particular, we think that the reactor thread is busy with processing other calls that are scheduled earlier, causing the API calls to be processed later. To verify this, we ran the experiment again where we enable Dispersy, the component responsible for many calls in the reactor (as described in Section 6.2). We perform five requests per second and 500 requests in total for this experiment. The observed results of are illustrated in Figure 6.4 (the left plot corresponds to the 5 requests/sec row in Table 6.1). In the left plot, the response times of the performed requests with a regular Tribler session is displayed whereas in the right plot, we display the response times of the

run where we disable Dispersy. Note the different scale on the vertical axis, indicating that the requests performed when Dispersy is disabled, are substantially faster. Indeed, the average request time of the right plot is 48 ms, significantly lower than the average of the response times when Dispersy is enabled, namely 123 ms. While both graphs are producing a spiky pattern, the standard deviation of the right plot is 5.73 ms and the standard deviation in the left plot is 210.23 ms. We conclude that the variation in response times is much lower in the right plot and that most likely DIspersy is producing a big amount of work for the reactor thread, introducing considerable amounts of latency when performing API requests.

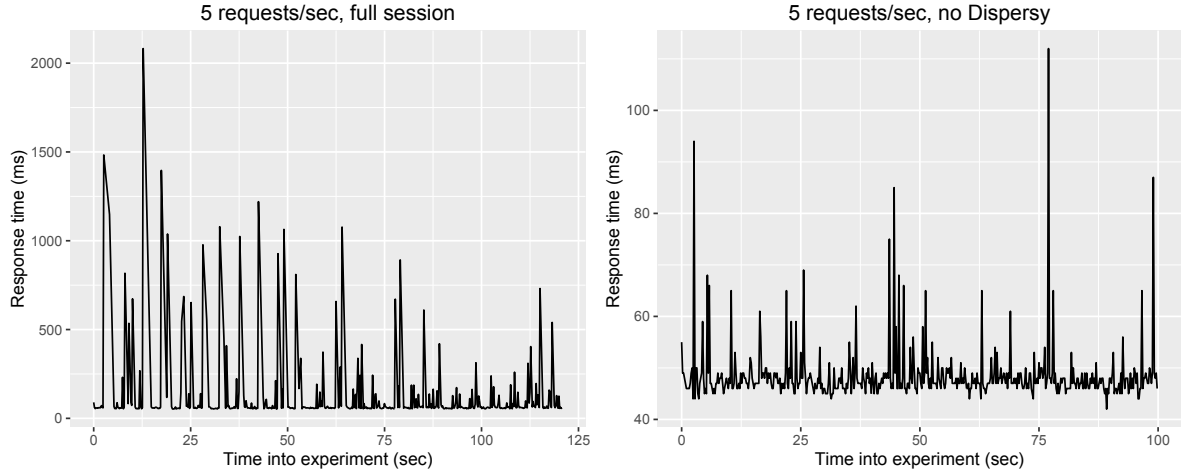


Figure 6.4: The response times of API requests, both within a full session and in a session where Dispersy has been disabled.

We identified a key issue here: the latency of methods to be processed in the Twisted reactor is high, causing the processing of incoming requests to be delayed. This does not only hold for the REST API: we can conclude the same for Dispersy and the tunnel community where possibly many incoming connections have to be served. The most important part of the solution is to make sure that there are no big blocking calls scheduled on the reactor thread that are taking a long time to complete. When a method call with a high runtime is executed on the reactor thread, Tribler is unable to process other events during that period, leading to an unresponsive system. Ongoing work is focussed on making the disk operations on the reactor thread non-blocking and as a consequence, reducing the latency of event processing and improving the responsiveness of the system in general.

Table 6.1 gives us another interesting observation, namely that we it appears that the bandwidth is reducing as the number of requests per second increases. This is in particular clear if we plot the theoretical maximum bandwidth against the observed bandwidth during the experiments, see Figure 6.5. In this Figure, we presented both the obtained bandwidth for a run using the regular Tribler session and a session where Dispersy has been disabled. We assume that each request contains 613KB of data in the response. The theoretical bandwidth is calculated as $b = 613 * n$ where n is the number of requests per second and b is the theoretical maximum bandwidth in KB/s. In practice, we will never reach this theoretical bandwidth since some time is required to create the connection and to process the response data in Tribler which we do not consider in our simple bandwidth model. We still notice that the obtained bandwidth is somewhat becoming constant, indicating that the bandwidth we can obtain is limited. This picture clearly shows the impact of a running Dispersy on the bandwidth. Whereas we almost obtain the theoretical output when we disable Dispersy, the gap between the theoretical and observed bandwidth becomes bigger in the run where we use a full session. When performing fifteen requests per second, the bandwidth even decreases, possibly due to the high system load the concurrent requests are causing.

6.4. Start-up experience

The first interaction that users have with Tribler, is the process of booting the software. During this boot process, various operations are performed:

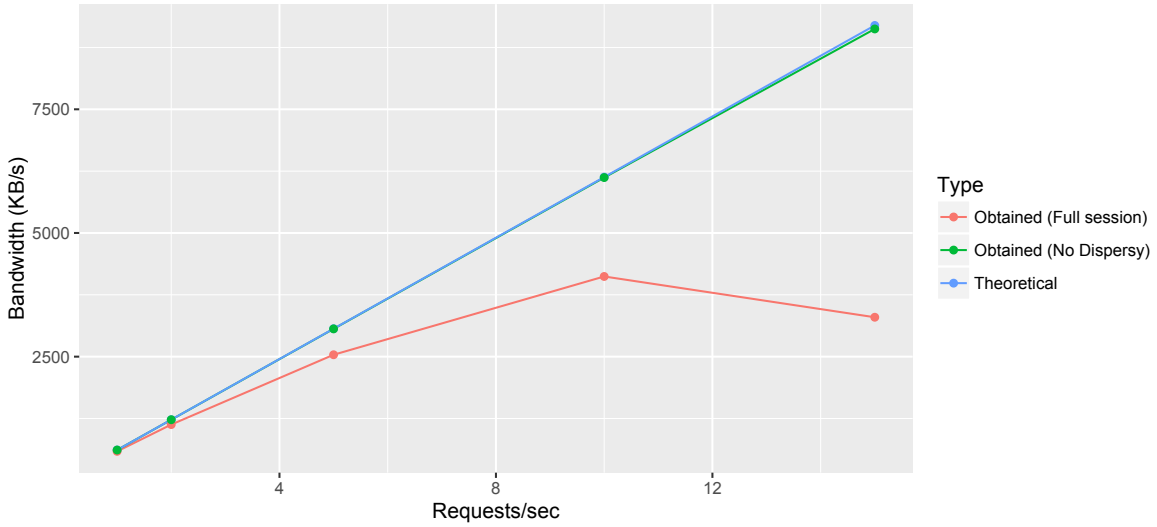


Figure 6.5: The maximum theoretical bandwidth compared to the obtained bandwidth (using a full Tribler session and disabled Dispersy) in the experiments.

- The Tribler state directory is created and initialized with necessary files such as the Dispersy key pair and configuration files.
- A connection to the SQLite database is opened and initialized. If this database does not exist, it will be created first.
- Dispersy is started and the enabled communities in the configuration file are loaded.
- Various Tribler components are created, including the video streaming server, the REST API, the remote torrent handler and the *leveldb* store.

The start-up process of the Tribler core proceeds sequentially and no parallel operations are implemented to speed up the process. Depending on the number of enabled components, the start-up time might vary.

To analyse the start-up time, Tribler is started 50 times. The experiments are performed several times where in one experiment the software is started for the first time, with no prior existing state directory. In these runs, a state directory is created and the required files are initialized. In another experiment, a database containing just over 100.000 torrents is used. This database is the result of running Tribler idle for several hours, after subscribing to some popular channels to fetch as much torrents as possible. The filled Tribler database comes in conjunction with a filled Dispersy database. In both scenarios, there are no active downloads. The experiment starts when the *start* method of the *Session* object is called and ends when the notification that Tribler has started, is observed. During the span of this thesis, there have been various changes to the start-up procedure of Tribler where some code has been modified. Since we wish to make sure that our modifications do not significantly decrease the start-up speed, we make a comparison between the code in November ’15 and July ’16. The results are displayed in Figure 6.6, where for each commit we compare, we present an ECDF with the boot time on the horizontal axis and within each plot, the distribution of boot times from a clean and pre-filled state.

In both plots, It is clear that magnitudes of the Tribler and Dispersy databases have impact on the time for Tribler to completely start. However, this impact is relatively minor since Tribler still starts well within a second. We think that this statistic justifies removal of the splash screen that is shown in the old user interface. The relatively short time the splash screen would be visible in the new interface is so short that users would not even be able to read and interpret the content of the splash screen. In contrast to the old user interface, the new interface starts Tribler and shows a loading screen after the interface has started. However, users are able to already browse through other tabs such as their downloads and discovered channels.

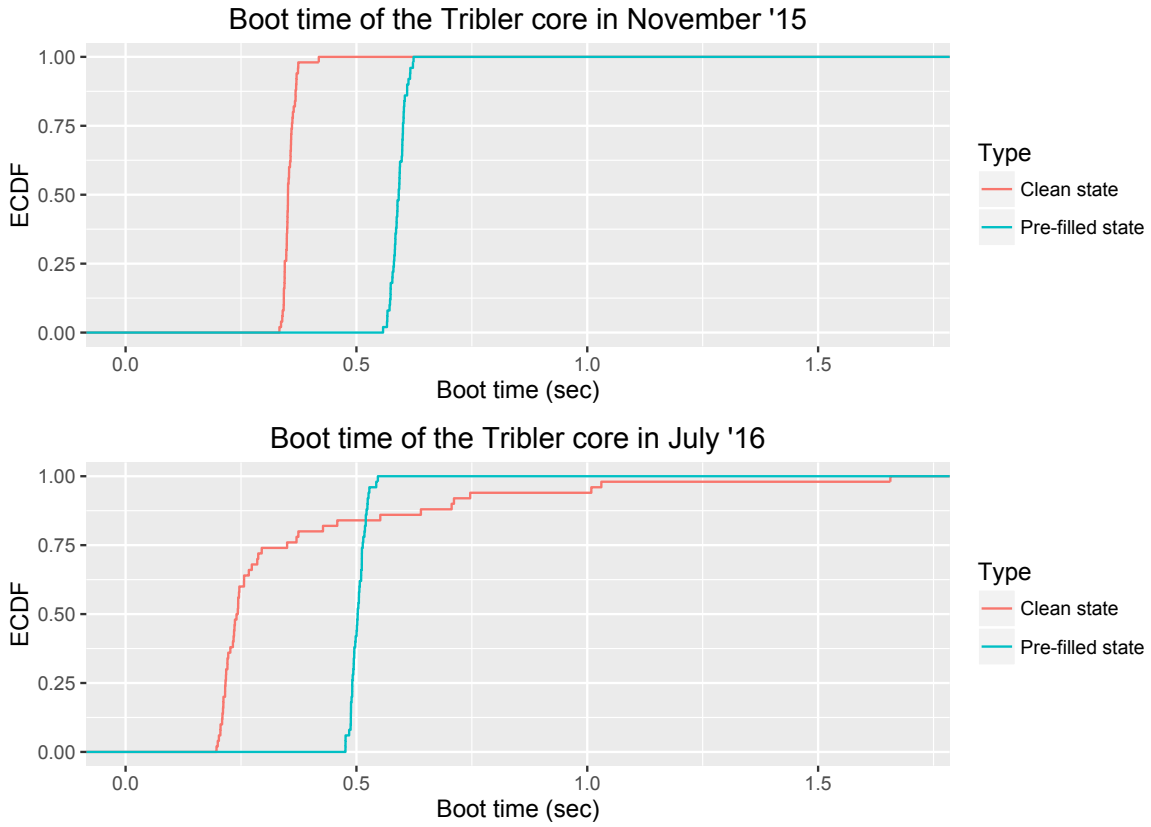


Figure 6.6: The boot time of Tribler from a clean and pre-filled state using the code base in November ’15 and July ’16.

Whereas the boot times of the runs performed with the November ’15 code are very constant, we notice a variation in the runs with the code base from July ’16, indicating that there is some component that has a high variation in initialization time. Further analysis learns us that this variation can be addressed to Dispersy, possibly caused the boot procedure of one of the communities. However, further analysis of the boot time of Dispersy is outside the scope of this thesis work.

6.5. Remote Content Search

We wish to serve relevant information to users as fast as possible. To help users search for content they like, a remote keyword search has been implemented, where users can search for torrents and channels. Channel results are fetched by a query in the *AllChannel* community whereas torrent results are retrieved by a query in the *search* community.

To verify the speed of the remote torrent search, various experiments are conducted. We are using a list of 100 terms that are guaranteed to have matches in the network. Each search query is executed when there are at least 20 connected peers in the *SearchCommunity*. The time-out period of the remote search is 60 seconds, indicating that search results that are coming in after this period are not regarded. This experiment has a particular focus on two statistics: the time until the first remote torrent search results comes in and the turnaround time of the search request, indicating the time until the last request comes in. We should note that users performing a remote search might see results earlier since a search query in the local database is performed in parallel. The results are visible in Figure 6.7.

Overall, the remote torrent search as implemented in Tribler is very fast and performs well. For every search query performed, we had at least one incoming result and on average, 61 search results are returned for each query where the first incoming torrent result takes 0.26 seconds to arrive. As we

see in Figure 6.7, over 90% of the first remote search results are available to the user within a second. During our experiment, we always have the first incoming torrent result within 3.5 seconds. The other graph shows the turnaround time of the request, indicating the time until the last response within our time-out period. On average, it takes 2.1 seconds for all torrent search results to arrive. In the plot, we see that in over 90% of the search queries, we have all results within 10 seconds.

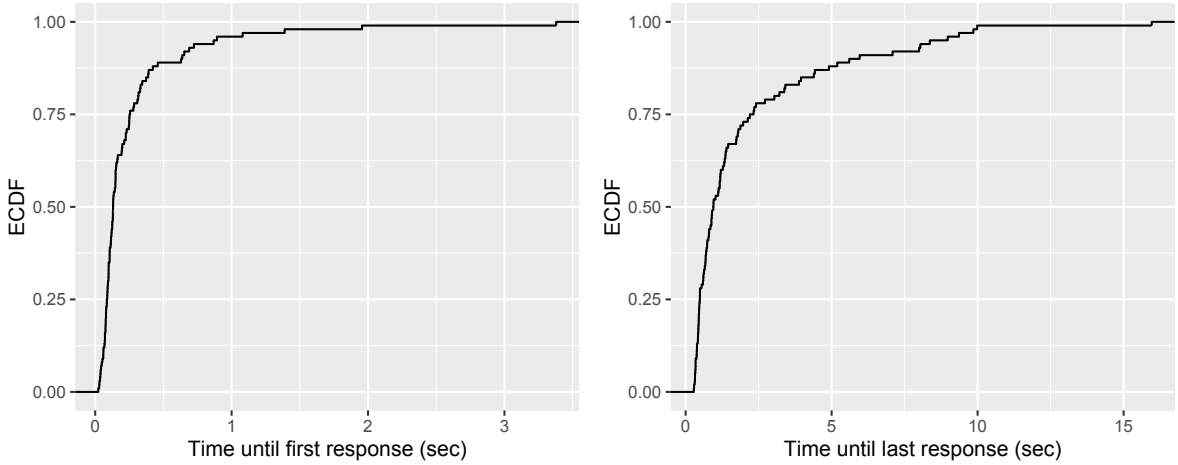


Figure 6.7: The performance of remote content search, expressed in the time until the first response and time until last response.

The same experiment has been performed in 2009 by Nitin et al. where 332 remote search queries have been performed, see Figure 6.8 where the time until the first response from any remote peer in the network is measured. The graph makes a comparison before and after a significant improvement to the I/O mechanism, causing messages to be exchanged faster. The observed average time until first response in 2009 is 0.81 seconds whereas our observed average time is 0.26 seconds.

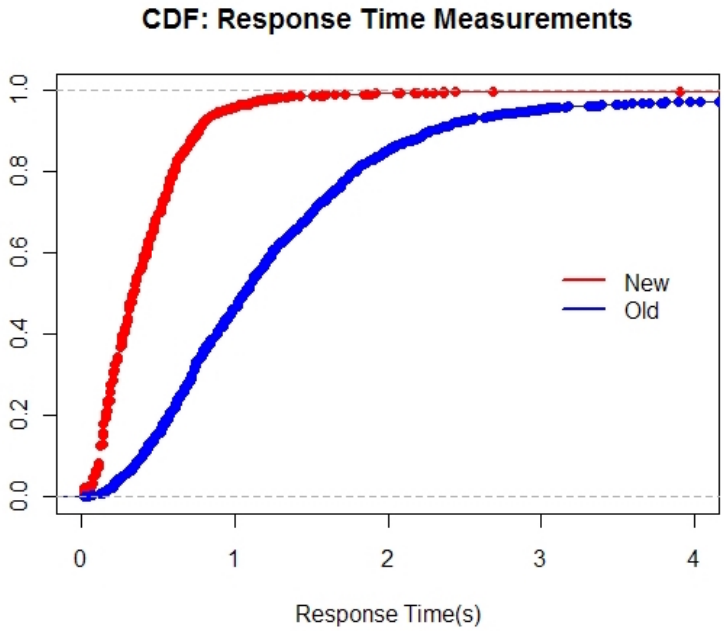


Figure 6.8: The performance of remote content search, performed by Nitin et al. in 2009. The new remote search had an improved I/O mechanism, causing messages to be exchanged faster.

6.6. Local Content Search

In the previous Section, we demonstrated and elaborated the performance of the remote content search mechanism. Now, we will shift the focus to performance measurements of a local content search, which is considered more trivial than the remote search counterpart where network communication is required. In particular, our goal is to quantify the performance gain or loss when switching to the new relevance ranking algorithm, as described in Chapter 4.5.

The setup of the experiment is as follows: a database with just over 100.000 torrents is used. Around ten seconds after starting Tribler, we perform a local *torrent* search every second and we do this for 1.000 random keywords that are guaranteed to match at least one torrent in our database. We will measure both the time spent by the database lookup and the time it takes for the data to be post-processed after being fetched from the database. In the old relevance ranking algorithm, this post-processing step involves determining the associated channels for each torrent result. This experiment is performed for the old ranking algorithm that uses the Full Text Search 3 (FTS3) engine and the new ranking algorithm that uses the more recent FTS4 engine. According to the SQLite documentation, FTS3 and FTS4 are nearly identical, however, FTS4 contains an optimization where results are returned faster when performing searches with keywords that are common in the database. The results are visible in Figure 6.9, presented in two ECDF plots.

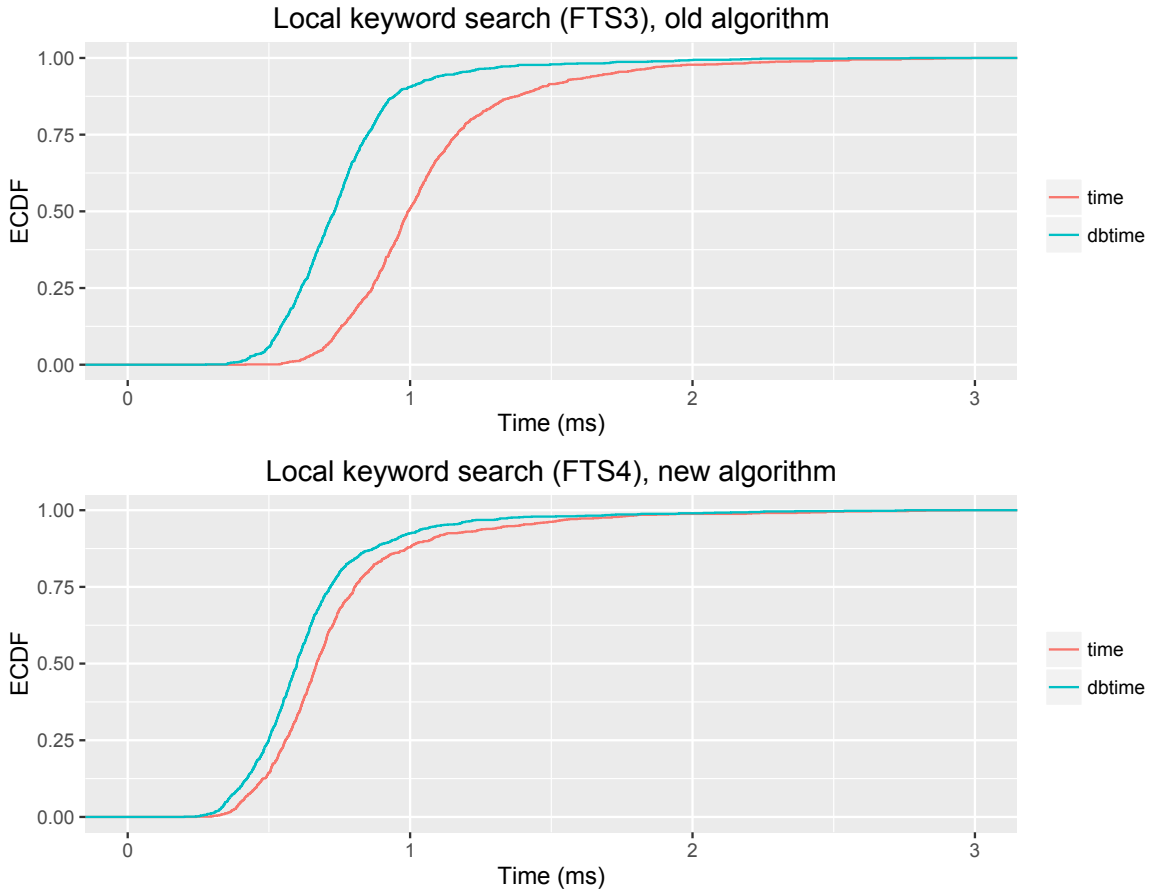


Figure 6.9: A comparison of the performance of local keyword searches between the old ranking algorithm with the FTS3 engine and the new ranking algorithm utilizing the FTS4 engine.

Local content search is very fast, delivering results in several milliseconds and low priority should be given to performance engineering on the local content search engine. We see that the two lines in the FTS3 and FTS4 graphs have moved closer to each other which means that the speed of post-processing of torrent results has increased. This is in line with our expectations since the new relevance

ranking algorithm should be less computationally expensive than the old one. In addition, the new algorithm takes less factors in considering, for instance, the swarm health of the torrent. The increase in performance from FTS3 to FTS4 is visible but not significant.

In 2009, Nitin et al. performed the same experiment where they used a database filled with 50.000 torrents. Their generated ECDF is displayed in Figure 6.10. We notice that the current performance of local search in our experiment is dramatically better than the performance obtained during the 2009 experiment. This can be explained by the fact that Tribler used a custom inverted index implementation when the experiment in 2009 was conducted. An inverted index is an index data structure where a mapping is stored from words to their location in the database and is used on a large scale by search engines, including the FTS engine of SQLite. By utilizing this mapping when performing a full text search, we can get results in constant time. However, there is a slight overhead for maintaining and building the inverted index when new entries are added to the database, also impacting the size of the database disk file. The built-in FTS engine of SQLite is optimized to a large extent and clearly offers a higher performance than a custom implementation.

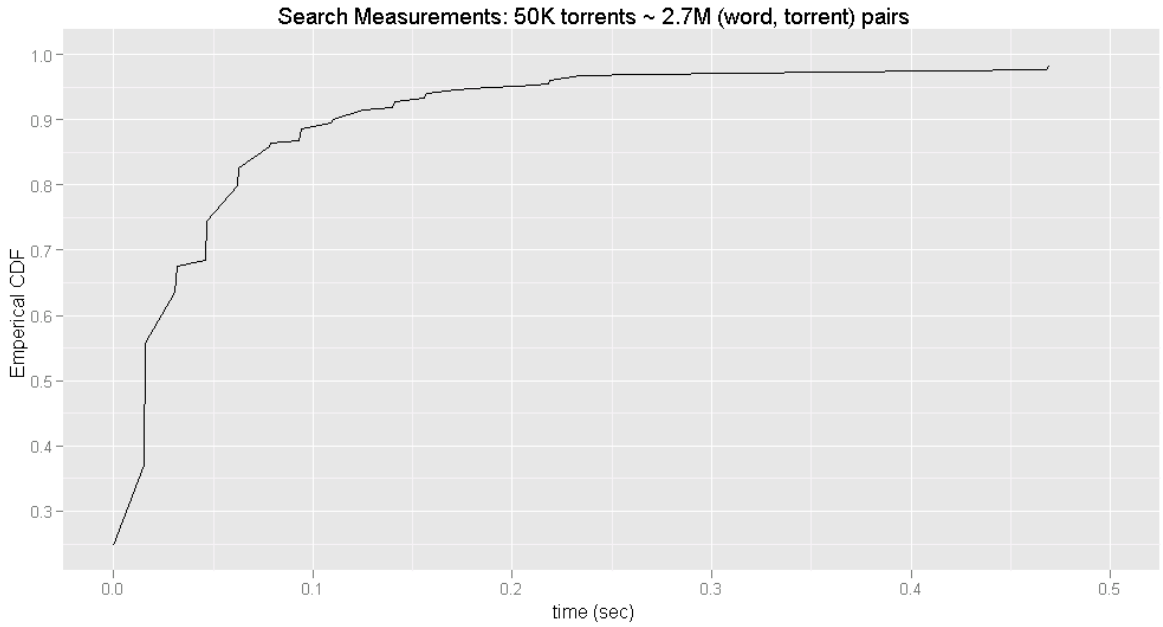


Figure 6.10: The performance of a local database query as verified by Nitin et al. in 2009.

6.7. Video streaming

The embedded video player in Tribler allows users to watch a video that is being downloaded and is explained in more detail in Chapter 3.4.3. Video playback has been available since Tribler 4.0. It is based on VLC and offers support for seeking so the user can jump to a specified offset in the video. Video downloads have a special Video On Demand (VOD) mode which means that the *libtorrent* library piece picking mechanism uses a linear policy mode. In this mode, pieces are downloaded in a timely matter. When the user seeks to a position in the video, the prioritization of the pieces are modified, giving priority to pieces just after the specified seek position. Users also have the possibility to use an external video player that support playback of HTTP video streams.

The bytes are streamed to a VLC client using a HTTP stream. When Tribler starts, a HTTP video server is started. This server supports HTTP range requests which means that a specific part of a video file can be queried by using the HTTP *range* header. This is useful when the user performs a seeking operation since only a specific part of the file has to be returned in the HTTP response. If some pieces are not available, the video server will wait until these bytes are downloaded before returning these bytes in the response.

First byte	Request time (sec)
0	11.6
$1 * 10^9$	64.4
$2 * 10^9$	64.6
$3 * 10^9$	65.9
$4 * 10^9$	100.6
$5 * 10^9$	115.6
$6 * 10^9$	115.8
$7 * 10^9$	12.2
$8 * 10^9$	66.6
$9 * 10^9$	52.4

Table 6.2: Performance of the video server when requesting bytes at different offsets of the video being downloaded.

To improve user experience, we wish to minimize the delay that users experience when performing a seek operation in the video player. The experiment performed in this Section, will quantify this buffering delay. For this purpose, the well-seeded *Big Buck Bunny*[3] movie will be downloaded. The movie file itself is 885.6 MB in size and has a duration of 9:56 minutes. We will perform various HTTP range requests using the *curl* command line tool, immediately after starting the download in Tribler. For every run, we will request 10 megabyte of data and we will measure the total time it takes for each HTTP request to complete. The results are visible Table 6.2.

Theoretically, we would expect around the same request time for each range request, assuming that the availability of each piece is high. When performing a seek operation in the video, the piece picking mechanism adjusts priorities and these prioritized pieces should start to download immediately. The experiments shows some serious flaws in this mechanism where it might take up to two minutes for data to be available. Further investigation of this issue learns us that the video player always tries to download the first 10% of the video file, except in the 7th run of the experiment, where the prioritizing policy seems to be correctly applied. Solving this bug is considered further work and documented in GitHub issue 2508[6].

6.8. Content discovery

Content discovery is a key feature of Tribler. By running Tribler idle for a while, content is synchronized with other peers using the Dispersy library. When a user starts Tribler for the first time, there is no available content yet. We will verify the discovery speed of content after a first start. The experiment is structured as follows: we measure the interval from the completion of start procedure to the moment in time where the first content is discovered. We perform these experiments for both torrents and channels and repeat this 15 times. The results are visible in Figure 6.11.

The delay of discovering the first channel is reasonably: on average, this happens 18 seconds after start-up. In all runs, we have our first channel discovered within 35 seconds. Discovery times of the first torrent is slightly slower and in all runs, the first torrent in a channel is discovered within 40 seconds. Figure 6.11 suggests that a torrent discovery always happens after there is at least one discovered channel. This is true: after the channel is discovered, the *PreviewChannel* community is joined where torrents are exchanged and discovered.

In the old user interface, users were presented with a blank screen with no feedback about new discovered content. In the new interface, the user is presented with a screen that informs the user that Tribler is discovering the first content. This screen is only shown the first time Tribler is started and goes away when there are five discovered channels, after which the page with discovered channels is displayed to the user.

[3] https://peach.blender.org

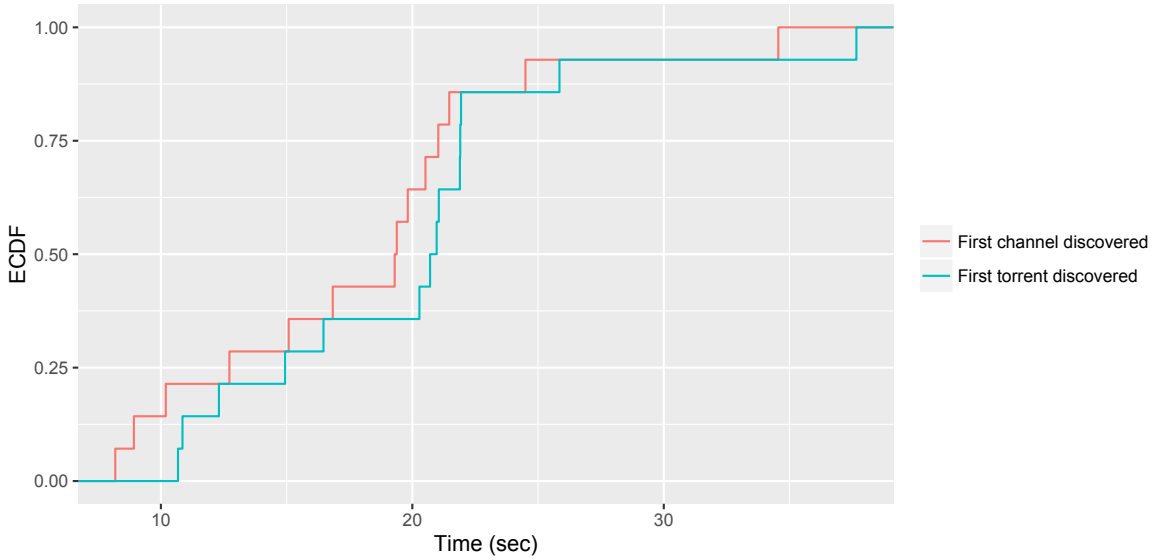


Figure 6.11: The discovery speed of the first channel and torrent after starting Tribler for the first time.

6.9. Channel subscription

When Tribler runs idle, not all available content in the network is discovered. The majority of content is discovered when users subscribe to channel (in the old user interface, this is referenced to as marking a channel as favourite). When Tribler discovers a new channel, users are presented with a preview of this channel. Internally, Tribler connects to the *PreviewChannelCommunity* associated with that channel, a community derived from the *ChannelCommunity*. In this preview community, the amount of torrents that are collected is limited. The *ChannelCommunity* in turn is joined the moment the user subscribes to a channel, after which the full range of content is synchronized. Removing the preview mechanism might significantly increases the resource usage of the Tribler session since the amount of incoming messages to be decoded and verified will increment.

The experiment as described in this Section, will focus on the discovery speed of additional content after the user subscribes to a specific channel and on the resource allocation when we are running Tribler without enabling a preview mechanism of channels. For the first experiment where we determine the discovery speed of additional content inside a channel, the twenty most popular channels (with the most subscribers) are determined. For this purpose, we are using a Tribler state directory with many discovered channels but void of any channel subscriptions. Exactly ten seconds after Tribler started, we subscribe to one of these popular channels and we measure the interval between subscription to the channel and discovery of the first additional torrent. Tribler is restarted between every run and the state directory is cleaned so we guarantee a clean state of the system. The observed results are visible in Figure 6.12.

The average discovery time of additional torrents after subscription to a channel is 36.8 seconds which is quite long, compared to the discovery speed of the first channel and torrent as described in Section 6.8. The discovery times have a high variation as can be seen in Figure 6.12. This can be explained by the fact that immediately after subscribing to a channel, Tribler will connect to the *ChannelCommunity* that is joined after subscription and peers have to be found.

To verify the impact of automatically subscribing to each channel when it is discovered, we perform a CPU usage measurement. In two idle runs of a Tribler session, both lasting for ten minutes, we measure the CPU usage every ten seconds using output provided by the *top* tool. In the first run, a regular Tribler session is used where previews of discovered channels are enabled. In the second run, we bypass the preview of a discovered channel and immediately join the channel, synchronizing all available content. Both types of runs start with an empty state directory. The results of this experiment are visible in Figure 6.13.

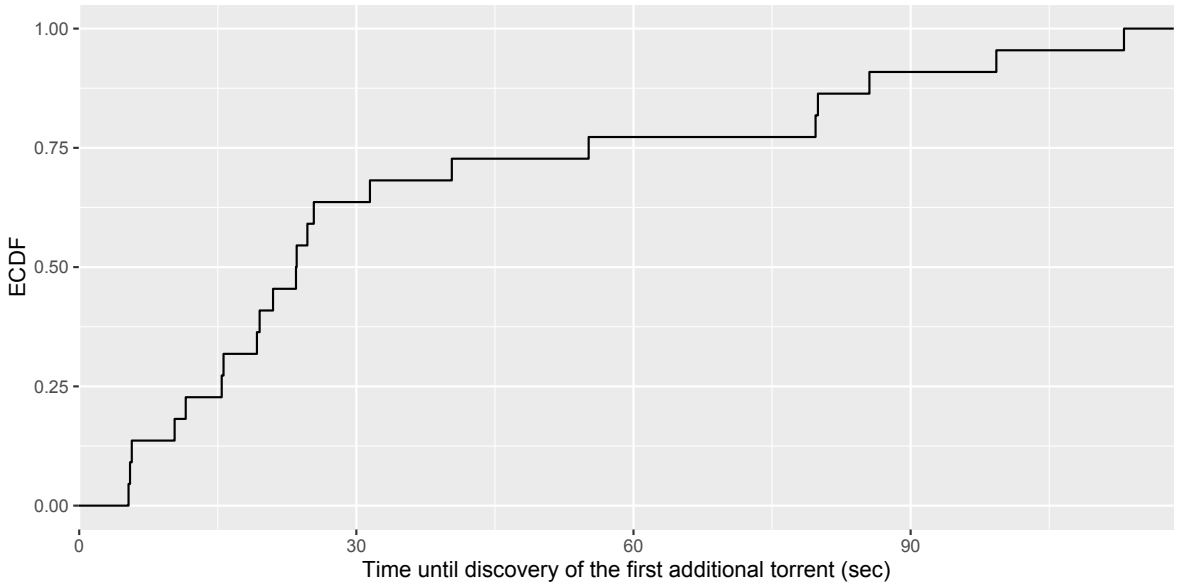


Figure 6.12: The distribution of discovery times of the first additional torrent after subscribing to a popular channel.

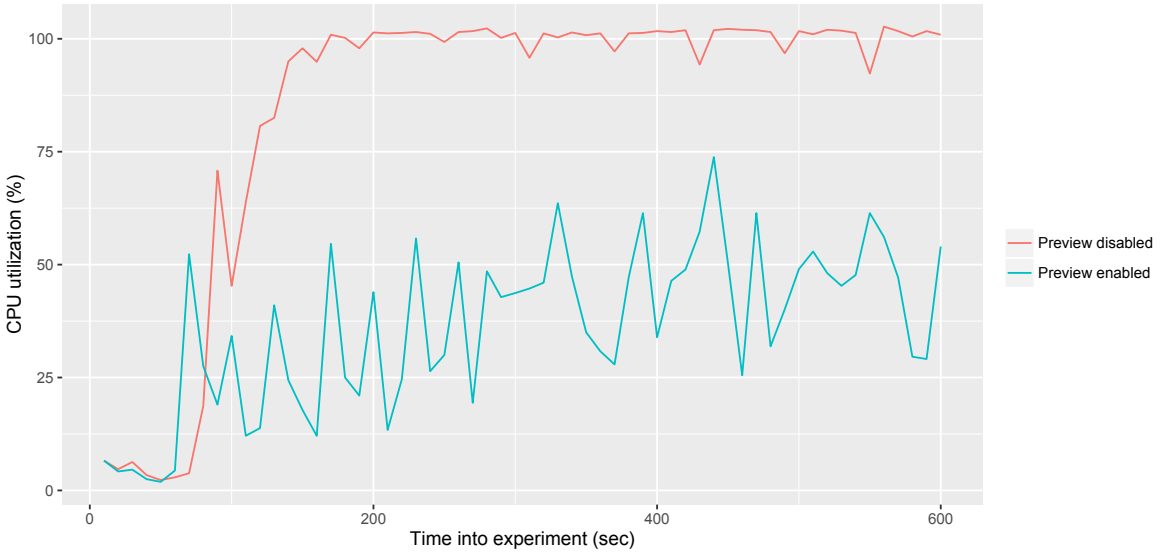


Figure 6.13: The CPU utilization of one core during a period of ten minutes with channel preview enabled and disabled.

Whereas the CPU usage of the normal run is around 45% on average, the CPU is busy and quickly rises to 100% utilization when we enable the auto-join mechanism of channels. This shows that it is infeasible to enable this auto-join feature if we still wish to guarantee a responsive system. One might limit the rate at which discovered torrents are fetched, however, this requires a feedback mechanism where we should notify other peers in the community to limit the amount of messages sent to the peer that is discovering content. Implementing of such as feature is outside the scope of this thesis work and is considered future work.

6.10. Torrent Availability and Lookup Performance

While specific information about torrents such as the name and file names are distributed within the Dispersy communities, this does not hold for the meta info about the torrent itself, which includes additional data such as trackers and piece information. This meta info can be important for users since trackers provides information about the health of a torrent swarm. The experiments as explained in this

Section, will investigate the torrent availability and lookup performance of meta info of torrents, either by using downloading them from remote peers in the Tribler network or from the Distributed Hash Table (DHT).

6.10.1. TFTP Handler

When users are performing a remote torrent search, the first three incoming results are pre-fetched which means that the meta info of these torrents are fetched automatically. An incoming search result can possibly contain information about remote peers (candidates) that have the meta info of this torrent available. If candidates for a specific remote torrent result are present, an attempt to fetch the torrent meta info from this candidate is scheduled. This request is performed using the Trivial File Transfer Protocol (TFTP)[28] which is a simplified version of the more sophisticated File Transfer Protocol (FTP), commonly used to transfer files over the internet. TFTP is also used to transfer meta data about torrent files such as thumbnails between peers, however, meta data of torrents is currently disabled in Tribler. The implementation of TFTP can be found in the core package in the code base.

There has been no published studies yet about the performance of our TFTP implementation so we have no available reference material. The experiment performed in this Subsection will focus on the performance of TFTP when fetching meta info from remote peers. We start from a clean state directory and exactly one minute after starting Tribler, we perform a remote torrent search. For each incoming remote search result, we perform a TFTP request for each candidate attached to this result. We perform ten remote torrent search operations, with interval of 60 seconds between them. After eleven minutes, we stop Tribler and gather the statistics of the TFTP sessions.

Total requests scheduled	1008
Requests in queue	761 (75.5%)
Requests failed	106 (10.5%)
Requests succeeded	141 (14.0%)

Table 6.3: A breakdown of the performed requests during the TFTP performance measurement.

We notice that the queue keeps growing: when our experiment is finished, 75.5% of the initiated requests is still in the queue. The second observation is the high failure rate, when compared to the amount of succeeded requests (42.9% if we do not consider the requests in the queue). We identified two underlying reasons for the failed requests. Some of the requests timed out, possibly due to the fact that some remote peers are not connectible. A solution for this kind of failure would be a more robust NAT puncturing method. The other reason is that the remote peer does not have the requested file in the local persistent storage. While this situation might seem unusual, it can happen if the remote peer has the requested torrent in the SQLite database but not in the meta info store. We can solve this by not returning this peer as candidate if the torrent is not available in the meta info store. This solution will also reduce the total bandwidth used by the TFTP component.

Next, we will focus on the turnaround time of the successful TFTP requests, these are displayed in figure 6.14 in an ECDF. Here, we notice the weird distribution of the turnaround times. We would expect that the total request times to fetch torrents is somewhat constant, however, we see outgoing requests that take over 400 seconds to complete. This trend will probably continue if we did not stop the experiment after ten minutes. The most logical explanation for this is that requests are added to the request queue at a faster rate than the processing speed of these requests. This also explains the values denoted in Table 6.3 where 75.5% of the request are still in the queue after the experiment ends. Better support for parallel requests should help, however, this is considered further work and outside the scope of this thesis work.

6.10.2. Distributed Hash Table

t tot 1 laten

Apossible source of torrents is the Distributed Hash Table (DHT). The DHT provides primitives to query torrent files and peers, based on a specific infohash of a torrent. Querying the DHT for torrent files can be done by invoking the *download_torrentfile* in the *Session* object. One should specify the callback

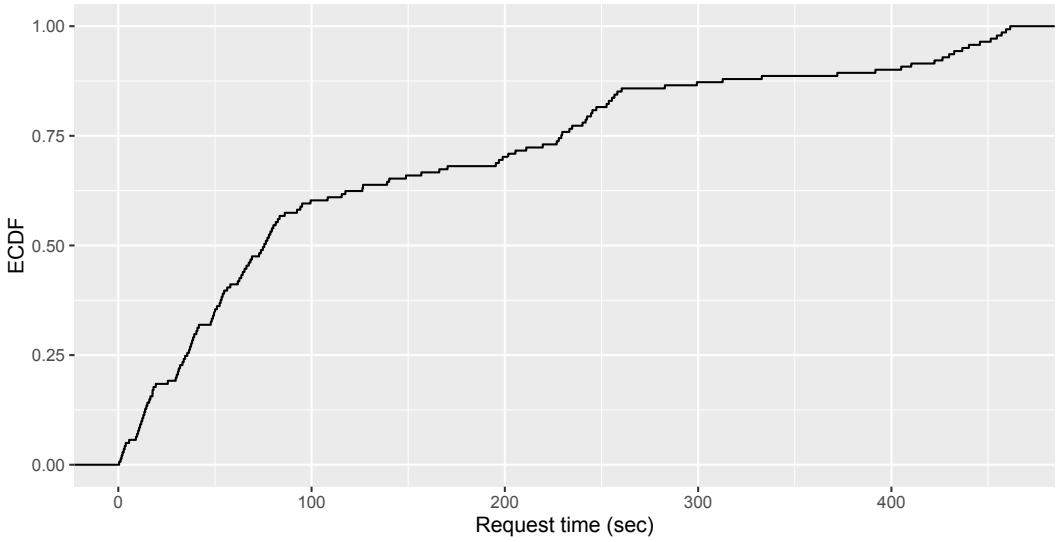


Figure 6.14: An ECDF of the performance of the torrent meta info download mechanism using TFTP in Tribler.

to be invoked after the meta info is successfully downloaded. In this Section, experiments will be conducted to get insights in the availability of torrent files and the performance of lookup operations in the DHT. This experiment is has a relevance to user experience since users that want to determine whether specific content is interesting or not, they first might want to view meta info of the torrent file, such as names and sizes of the files. This meta info should be available as soon as possible.

In the current user interface, the torrent file is fetched when the user single clicks on a torrent in the list of torrents, either when browsing through contents of a channel or after performing a remote keyword search. In addition, when executing a remote search, the first three top-results are pre-fetched since the user might be interested in them. For this experiment, a popular channel with over 5.000 torrents is considered and a subset of torrent infohashes in this channel is taken. Every 40 seconds, a DHT query is performed with one of the 1.000 random infohashes. The time-out period used in Tribler is 30 seconds, after which a failure callback is invoked and an error is displayed in the user interface to notify the user about the failed request. The results of this experiments are visible in Figure 6.15. Torrents that cannot be successfully resolved from the DHT, are assigned a value of 30 seconds in the graph.

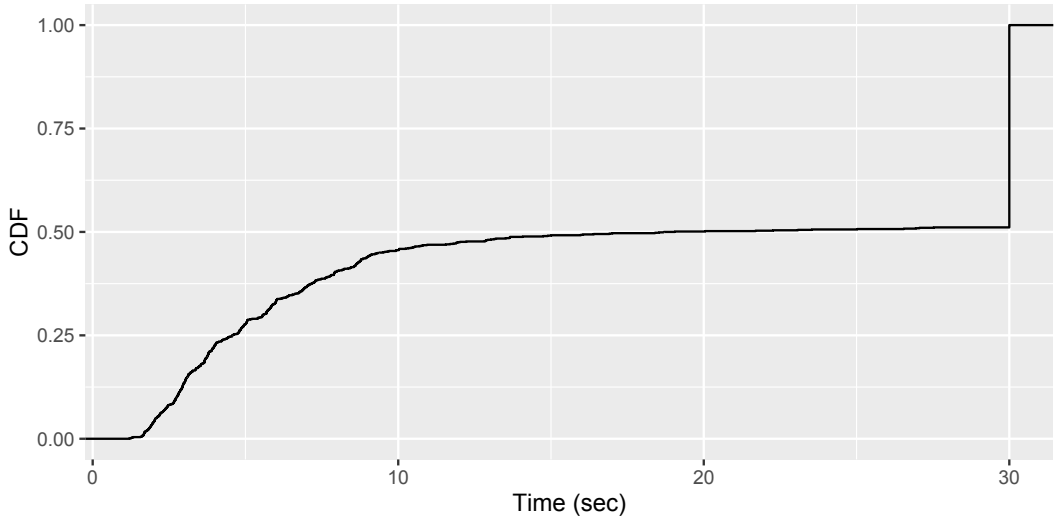


Figure 6.15: Test.

We immediately notice that the failure rate of DHT lookups is quite high: a little under 50% of the lookup operations are timing out. This issue might be addressed to dead torrents (when no peers in the DHT have this torrent information available) or private torrents (torrents which information is not available in the DHT). The amount of failures might be even higher in a less popular channel since the content in these channels are probably less seeded. As explained in the previous Subsection, the DHT is not the only source for torrents in Tribler and we might also fetch torrents from other peers using TFTP. Unfortunately, the approach of fetching meta info about torrents from other peers is only usable when searching for torrents. Caching and exchanging torrent candidates is not successful since the availability of candidates cannot be guaranteed.

The average lookup time of torrents that are successfully fetched from the DHT is 5.81 seconds which is reasonably fast. Additionally, Figure 6.15 shows that a little over 90% of the successfully fetched torrents are retrieved within 10 seconds.

To improve performance of metainfo lookup, dead torrents should be handled correctly. One possible solution might be an implementation of a periodical check for each incoming torrent. By limiting the number of outstanding DHT requests, this approach does not create require much additional resources. To further improve performance, the result of DHT lookups might be disseminated to remote peers in the network. Torrents that are not successfully fetched from the DHT, could be hidden automatically in the user interface. The downside of this approach is that it might not give a realistic view of the availability of a torrent since their might be candidates which have a copy of this torrent available.

7

Conclusions

The work in this thesis investigates the technical debt that has been accumulated by over 40 unique contributors over the last ten years of scientific research in the area of decentralized networks. After a comprehensive discussion of the architectural evolution Tribler has made, A future-proof and robust architecture of Tribler is proposed, discussed and some parts of the new design have been implemented. The main components as created in this thesis are a new user interface, built using the Qt framework, and a REST API, allowing Tribler to run on remote devices while giving developers high amounts of flexibility and ease when developing with Tribler. The testing environment has been improved with the addition of proper and stable unit tests and the tests are now executed on multiple platforms, allowing us to find defects due to platform incompatibilities earlier in the development process. Summarizing, this work transformed Tribler from an unattractive, unmaintained and untested system into a platform that is ready for development during the next decade of scientific research.

While this is a step in the right direction, there still is a lot of work to do by the next generation of Tribler developers. We should learn from the mistakes made in the past years. Being critical towards the implementation of quick workarounds is one example of that. Mandatory code reviews by other team members helps to improve one's code and to get a more critical attitude towards favouring short-term decisions over long-term agreements. We also propose that it is the responsibility of every developer to write code that is covered by the right amount of tests. By forcing a strict increasing code coverage policy, the code coverage metric can be controlled and gradually improved over time.

Additional future work proposed is the implementation of the trust walker, which will become the foundations of the Tribler platform. This way, the Dispersy framework could be removed, allowing the deletion of much legacy code. Although not the main subject of this thesis, the amount of accumulated debt in Dispersy is also rather high. From a performance perspective, research should be conducted to see how the performance on low-end embedded devices could be improved. Since our main performance bottleneck is the limited amount of CPU power in a specific core, one of the proposed solutions is to increase utilization of multi-core architectures. This can be achieved by splitting the architecture of Tribler in separate components that can independently run on several cores, however, this requires one to think very carefully about the final design and communication structure.

A

Gumby scenario file commands

As described in Chapter 6, a standalone Tribler runner that uses a scenario file has been created. The scenario file allows developers to specify commands at specific points in time after Tribler has booted. This Appendix describes the implemented commands and usage.

Command	Argument(s)	Description
start_session	-	Start a Tribler session.
stop_session	-	Stop a running Tribler session.
stop	-	Stop the experiment and write the gathered statistics.
clean_state_dir	-	Clean the default state directory of Tribler.
search_torrent	The search query and optionally the minimum number of peers required in before the search is performed.	Perform a remote torrent search.
local_search_torrent	The search query.	Perform a local torrent search in the database.
get_metainfo	The infohash of the torrent to be fetched.	Fetch meta info of a specified torrent from the DHT.
start_download	The URI of the download.	Start a download from a torrent specified by a given URI.
subscribe	The Dispersy channel identifier of the channel (can also be *random*).	Subscribe to a random or specified channel.

Table A.1: An overview of commands for the Gumby scenario file when performing experiments with Tribler.

Bibliography

[1] Technical debt. http://www.construx.com/10x_Software_Development/Technical_Debt/, 2007. Accessed: 2016-07-22.

[2] Tribler: A next generation bittorrent client? https://torrentfreak.com/tribler-a-next-generation-bittorrent-client/, 2007. Accessed: 2016-07-23.

[3] Technicaldebtquadrant. http://martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009. Accessed: 2016-07-22.

[4] History of tribler. http://blog.shinnonoir.nl/2011/08/history-of-tribler.html, 2011. Accessed: 2016-07-28.

[5] Production code vs unit tests ratio. http://c2.com/cgi-bin/wiki?ProductionCodeVsUnitTestsRatio, 2012. Accessed: 2016-07-27.

[6] Github issue 2508 - prioritization of the pieces is not correctly applied when seeking in a downloading video. https://github.com/Tribler/tribler/issues/2508, 2016. Accessed: 2016-08-02.

[7] Openhub - tribler project summary. https://www.openhub.net/p/tribler, 2016. Accessed: 2016-07-29.

[8] Production code vs unit tests ratio. http://twistedmatrix.com/documents/current/core/howto/reactor-basics.html, 2016. Accessed: 2016-08-01.

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

[10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[11] Mihai Capotă, Johan Pouwelse, and Dick Epema. Decentralized credit mining in p2p systems. In *IFIP Networking Conference (IFIP Networking), 2015*, pages 1–9. IEEE, 2015.

[12] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[13] M.A. de Vos. Vlc player in pyqt5. https://github.com/devos50/vlc-pyqt5-example, 2015.

[14] MA De Vos, RM Jagerman, and LFD Versluis. Android tor tribler tunneling (at3): Ti3800 bachelorproject. 2014.

[15] Roy Fielding. Fielding dissertation: Chapter 5: Representational state transfer (rest). *Recuperado el*, 8, 2000.

[16] Yuepu Guo and Carolyn Seaman. A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 31–34. ACM, 2011.

[17] Apache JMeter. Apache software foundation, 2010.

[18] K Sparck Jones, Steve Walker, and Stephen E. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information Processing & Management*, 36(6):809–840, 2000.

[19] Chris F Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987.

[20] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6), 2012.

[21] Antonio Martini, Jan Bosch, and Michel Chaudron. Architecture technical debt: Understanding causes and a qualitative model. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 85–92. IEEE, 2014.

[22] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4): 308–320, 1976.

[23] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

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

[25] Johan A Pouwelse, Pawel Garbacki, Jun Wang, Arno Bakker, Jie Yang, Alexandru Iosup, Dick HJ Epema, Marcel Reinders, Maarten R Van Steen, Henk J Sips, et al. Tribler: A social-based peer-to-peer system. *Concurrency and computation: Practice and experience*, 20(2):127, 2008.

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

[27] WF Sabée, N Spruit, and DE Schut. Tribler play: decentralized media streaming on android using tribler. 2014.

[28] K Sollins. The tftp protocol (revision 2). 1992.

[29] Risto JH Tanaskoski. *Anonymous HD video streaming*. PhD thesis, TU Delft, Delft University of Technology, 2014.

[30] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[31] Niels Zeilemaker, Boudewijn Schoon, and Johan Pouwelse. Dispersy bundle synchronization. *TU Delft, Parallel and Distributed Systems*, 2013.