Identifying and Managing Technical Debt in Complex Distributed Systems

M.A. de Vos

TU Delft

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 scientific 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. The platform suffers from a complex architecture, unintuitive user interface, an incomplete, unstable testing framework and a significant amount of unmaintained code. A new simple, flexible and component-based architecture that readies Tribler for the next decade of research, is proposed and discussed. We lay the foundations for this new architecture by implementing a flexible, convenient RESTful API and a new graphical user interface.

Additional work includes paying off various kind of technical debt by the means of a major refactoring in the testing framework, several heavy modifications within the core of Tribler and improvements to the infrastructure to make it more usable and robust. With the deletion of 12.581 lines, the modification of 765 lines and addition of 12.429 lines, we show that we contributed to the increase of several important software metrics and paid off a huge amount of technical debt. Raising awareness about the accumulated debt is of uttermost importance if we wish to prevent the deterioration of the system.

Our experiments will demonstrate that the performance of Tribler has not significantly degraded due to the invasive modifications as performed in this thesis work. We perform some static analysis to verify the usability of various components in the system and propose future work for the components that require more attention.

Preface

There are some people that contributed to this thesis I would like to thank. First, I would like to thank Johan Pouwelse for his supervision. Next, I would like to thank Elric Milon who taught me much about Python, Twisted and Linux and provided me with the necessary infrastructure. I would like to thank Laurens, Hans, Ernst, Niels and the rest of the Tribler development team for their help and support during this thesis by providing feedback and advise and by making this thesis a much more enjoyable experience.

Last but not least, I would like to thank my friends and family for their support and motivation.

M.A. de Vos
August 16, 2016

Contents

1

Introduction

The resources, budget and time frame of software engineering projects are often constrained[28]. 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 software. Research has confirmed that there exists a correlation between the amount of design flaws and vulnerabilities in a system[39]. A high amount of technical debt also leads to a greater likelihood of defects, unintended re-engineering efforts[33] and increased development time when implementing new functionalities.

The term technical debt was first introduced by Ward Cunningham in 1992 as writing "not quite right" code in order to ship a new product or feature to market faster[21]. Since then, the term has gained progressively more attention in the software engineering research and agile community. Effective management of such debt is considered critical to achieve and to maintain an adequate level of software quality. In 2007, Steve McConnell created the technical debt taxonomy where he refined and expanded the definition[4]. 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' the 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[6]. 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, together with some examples.

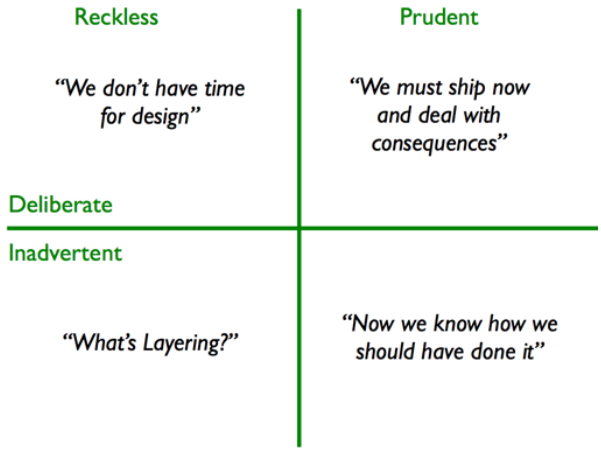


Figure 1.1: The technical debt quadrant, as proposed by Martin Fowler.

Technical debt has several interesting properties, explored and defined in the work of Brown et al[17].

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 to other developers. The value of 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 traced back to strategic decisions taken earlier in the development process. Otherwise, the debt could be accumulated in a more unintentional way, either due to recklessness or a lack of knowledge. Finally, we 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?

From the perspective of end users, technical debt can be both invisible and visible[32]. Examples of invisible technical debt includes code smells, coding style violations, low internal quality and high code complexity, issues developers should deal with. Visible debt is expressed in defects that are affecting users but can also be identified by user-unfriendly, cluttered Graphical User Interfaces (GUIs) that has been subject to various reckless modifications: decisions to extend and evolve the user interface with new visual elements, can lead to a high amount of technical debt and a poor user experience. In more extreme cases, the price of technical debt has to be paid with human lives as is painfully illustrated by a Toyota car accident due to unintended acceleration: after a review of Toyota's software engineering process and the source code for the 2005 Toyota Camry car, it was concluded that the system was defective and dangerous, riddled with bugs and gaps in its failsafes that led to the root cause of the crash[1].

The term itself is borrowed from the finance domain[27]. There is however one important distinction between financial debt and technical debt: when dealing with financial debt, the costs that the debtor has to pay is usually clear. This is not always the case with technical debt since there might be some situations where no technical debt is incurred. For instance, if it is known for a part of the system to never be updated or maintained in the future, development time and budget can be saved by not updating the related documentation. Software engineers need to be carefully with the consideration what technical debt they wish to incur and when this debt will be paid off at specific points in time.

There are several causes that contribute to the amount of accumulated technical debt during the software development process[35]. Time pressure can cause developers to think reckless about their design and architecture. Uncertainty in decision making during an early stage of development might lead to higher technical debt later on. Finally, in an agile environment, software requirements might change more often, causing the underlying architecture and code base to change to a certain extent. Not properly managing such changes can lead to significant technical debt.

Technical debt often becomes a noticeable problem in large systems that are built and maintained by many contributors. Tribler is an example of such a system: the software is the result of ten ongoing years of scientific research in the area of decentralized network technology and has incurred a serious amount of technical debt, both visible and invisible for users. The software 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*, offering a way to introduce the notion of trustworthiness inside a decentralized network and *Wikipedia*, allowing collaborative editing of content. These components are depicted in Figure 1.2.

Anonymity by the utilization of a Tor-like protocol has been added In 2014 by the work of R. Plak[40] and J.H. Tanaskoski[50]. In 2015, the protocol has been extended to support anonymous seeding of torrents[44]. The graphical user interface of Tribler is shown in figure 1.3.

This thesis work will be centred around identification and management of the accumulated technical debt within the code base of Tribler. With this in mind, we formulate the following research question:

How can we track, manage and prevent technical debt within Tribler?

This question can be divided into several sub questions:

1. What are the adequate tools to identity the amount of technical debt within Tribler?
2. What is the right approach to pay off each identified kind of technical debt?

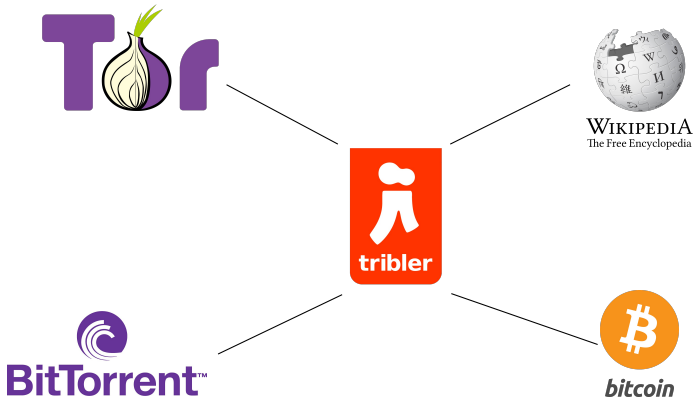


Figure 1.2: The four disruptive technologies as integrated in Tribler.

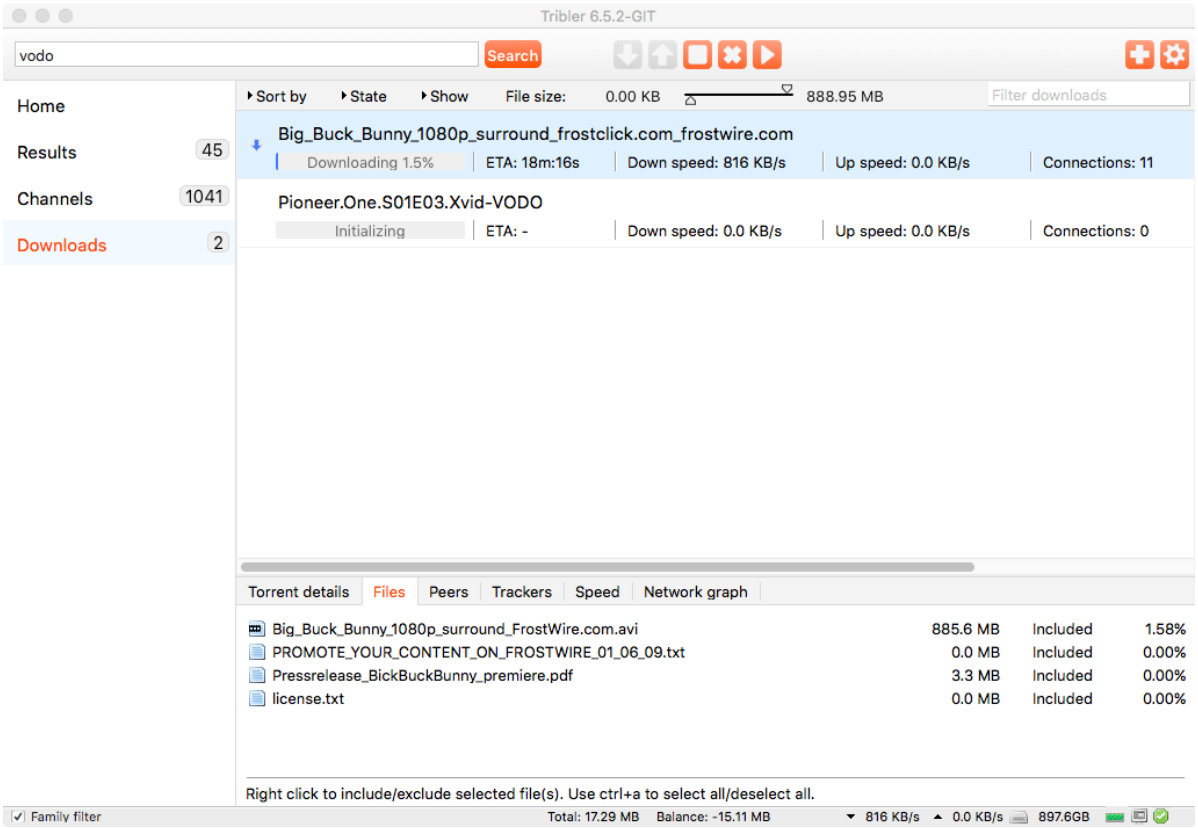


Figure 1.3: The graphical user interface of Tribler v6.5.2.

3. What are the adequate requirements in the software development process to prevent decisions that are leading to a high amount of debt later in the development process?

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 design and code base. In Chapter 3, the evolution of Tribler over the past ten years will be presented and we lay the foundations for a next decade of scientific research with Tribler by proposing a new future-proof and robust architecture. The first efforts towards a realisation of this new architecture will be discussed in Chapter 4 by the implementation of a RESTful API and a new user interface. Next, in Chapter 5, we will focus on the question whether we should pay off the debt in the core of Tribler and how we can do that. We will discuss efforts to improve code quality, architecture, infrastructure and the testing framework. The performance of Tribler after our 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 8.

2

Problem Description

The goal of this thesis project is to help Tribler mature from an experimental research prototype into production-level code with 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 reduction 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 more than 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, deferred infrastructure update decisions and feature pollution in the graphical user interface. This is illustrated by the fact that there even is a dedicated file in the Tribler code base, called *hacks.py* that facilitates various 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 the continuous integration environment and installers. 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 is 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 that the lack of student enthusiasm for fixing bugs, writing documentation and the absence of a code review policy are the root causes of current state of the code base.

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

2.1. A large code base

Tribler consists of a large and complex code base. This makes Tribler an unattractive open-source project for external developers to work on since the process to get familiar with the code base takes a considerably amount of time. Figure 2.1 illustrates the number of commits per months over the past ten years of Tribler software engineering. The evolution of the amount of source lines of code (SLOC) is shown in Figure 2.2. The magnitude of the project is also presented by Figure 2.3. From these Figures, it becomes evident that Tribler has continued to grow to a project with an unmaintainable amount of code. According to the basic software cost estimation model COCOMO[31], the established costs of the project is $2,371,403 with an estimated effort of 43 person-years.

This continued 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 a anonymous download mechanism, a credit mining system or an adult filter to hide explicit content in the user interface. After delivery of these features, the student leaves the project and knowledge about that specific part of Tribler he or she contributed to, is lost. Afterwards, that part of Tribler transits to an unmaintained state, due to lack of knowledge and manpower.

Continuous expansion of a system inevitably leads to feature pollution. During the lifetime of Tribler, no single effort has been made to do a proper clean up of the code, leading to 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, leading to much wasted development effort.

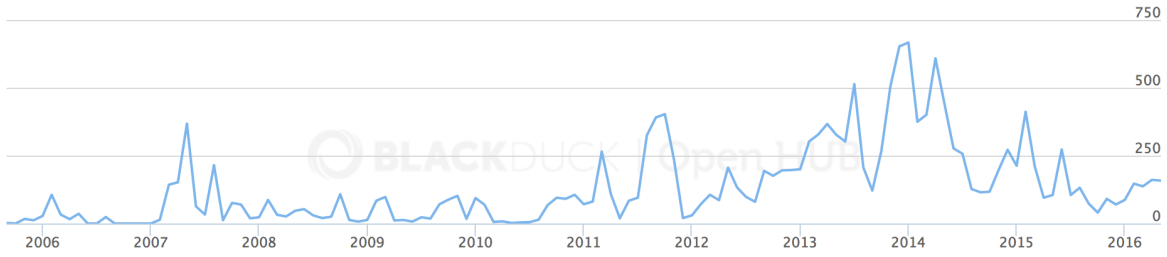


Figure 2.1: A history of commits per month on the Tribler project, as reported by Open Hub.

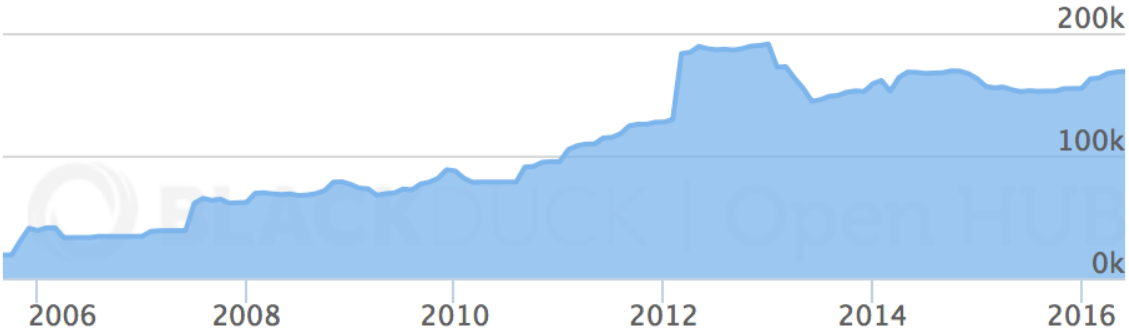


Figure 2.2: The evolution of lines of code in the Tribler project, as reported by Open Hub.

	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 enabled due to malfunctioning which could be removed.

The lack of maintenance is clearly visible in the messaging system of Tribler, called *Dispersy*. Dispersy is

a platform to simplify the design of decentralized communities and is mostly designed and written by N. Zeilemaker and B. Schoon[54]. After these developers left the project, knowledge of the Dispersy system disappeared and the system transited to an unmaintained state where the process of fixing new defects is being deferred.

Most researchers contributing to 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 open issues on GitHub, 100 of these issues are older than one year.

2.3. Architectural Impurity

During the lifetime of Tribler, the architecture has been subject to numerous minor and major modifications, leading to high amounts of *architectural debt*. Started as a fork of *Another BitTorrent Client (ABC)*, a torrent client based on *BitTornado*, Tribler has evolved to a platform that allows users to discover, manage, share and download content. The evolution of the Tribler architecture will be explained in-depth in Chapter 3.

On the highest level of the code base, two main components can be identified: the module with code that is responsible for the graphical user interface and the code that contains the implementation of core functionalities in Tribler. These modules have a mutual dependency on each other which is considered bad design since the core of Tribler should never be dependent on code realising a user interface. We consider breaking this dependency a high priority issue since it significantly impacts testability and modularity of Tribler.

Overall, the code base feels like a bunch of glued together research works where every developer has applied 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 broken or unused. After more analysis of the core module, we managed to identify various other issues, mostly related to code and design debt in the form of undesirable dependencies, code style violations and code smells.

The code related to the GUI is of very poor quality and plague with an astounding amount of cyclic dependencies. Having two files being dependent on each other, makes testing of the classes that these files contains, in isolation significantly harder. To get a better idea about the location of the main problems inside the package, we created an import graph of the GUI code base, presented in Figure 2.4 where a red edge indicates that this import dependency is part of a cycle. 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 files 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 user interface is built using code that's unmaintained and hard to understand. Many features and visual elements in the GUI are unnecessary and unintuitive, contributing to (visible) technical debt. Finally, the user interface has been written with the *wxPython* framework which is unmaintained 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 due to incompatibilities.

2.4. Unstable and incomplete testing framework

Testing is the responsibility of every developer that contributes to Tribler. Over the past ten years, this responsibility has been completely neglected by the majority of contributors, leading to a significant amount of *testing debt*. This is clearly visible in Figure 2.5 where we plot the ratio between the amount of code lines in the test suite (TLC) and the number of code lines related to production code (PLC). Tribler has a structural lack of well designed (unit) tests. Currently, around 100 tests are present that cover 71,2% of the source lines of code located in the Tribler core. Many of these tests are taking over half a minute to complete and are depending on a running Tribler session. Only a small fraction of the test suite has characteristics of unit tests. Having tests that are doing a broad range of operations, inevitably leads to undesired side-effect and failing tests. As far as we are aware, no attempt has been made to mock components of the system to simplify testing and to focus on the specific part of the system that has to be verified.

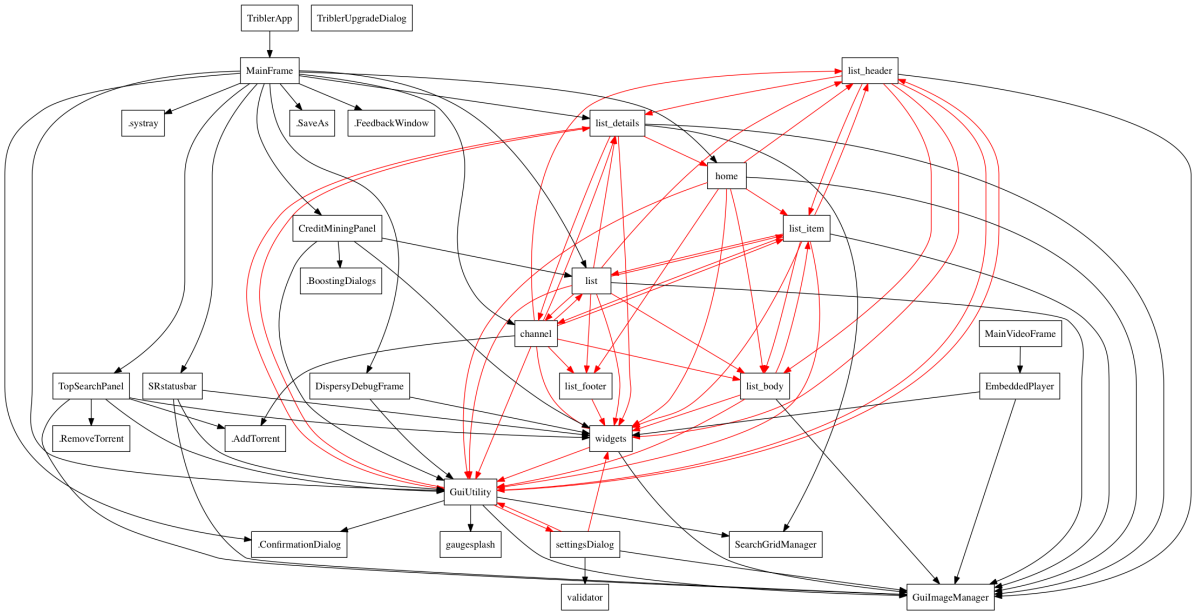


Figure 2.4: A generated import graph of the GUI code base. A red edge indicates that this edge is part of an import cycle.

There is an additional flaw that contributes to the instability of the current test suite. A significant part of the test suite is depends on external network resources, ranging from torrent trackers and seeders to other peers in the decentralized Dispersy 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 on multiple platforms, the 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[15]. Attention should be given to make our test execution multi-platform.

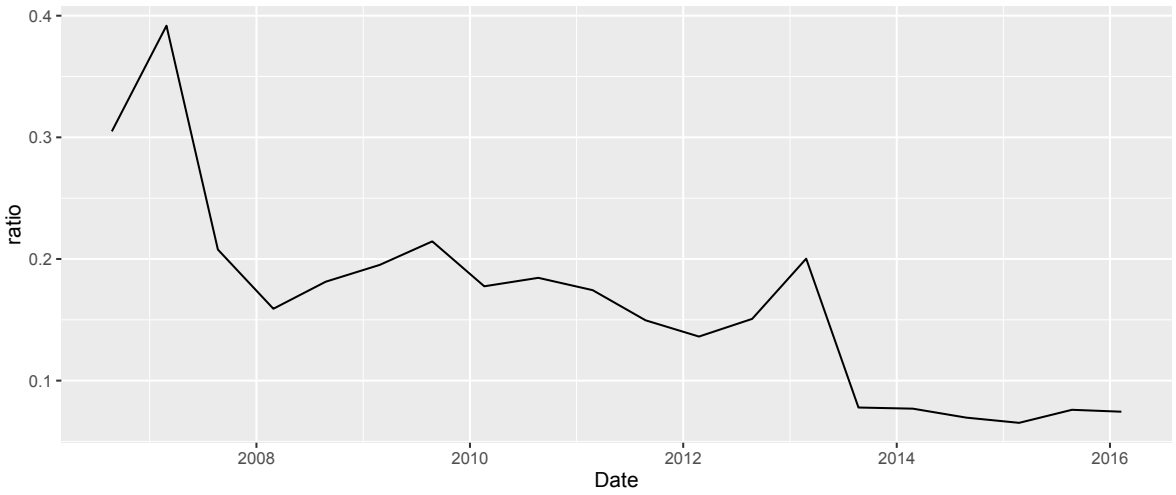


Figure 2.5: The ratio between the lines of code in the tests package and other lines of code over time.

3

Architecture and Design

Before we reach the point of proposing a new future-proof, modern architecture, we first focus on the evolution of the Tribler architecture throughout the last decade of research. Better understanding of the architectural and design decisions that have been taken in the past, will help us to shed light on the question what contributed to the current state of the Tribler system.

According to the Open Hub tool[11] which accumulates statistics about many open source projects available on the Internet, Tribler received code contributions from 111 unique contributors so far. This list is most likely not exhaustive since some work of contributors might have been finished by other members of the Tribler team or has never been merged into the main code base. A query for *Tribler* in the repository of Delft University of Technology[1], yields a total of 66 search results, 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) research work.

The remainder of this Chapter will present a historical view of the evolution of the Tribler platform, starting in 2007 and concluding with the proposal of a new, robust and scalable architecture that is ready for the next decade of research.

3.1. Tribler in 2007: A social-based peer-to-peer system

In April 2005, Tribler started out as a fork of the *Another BitTorrent Client (ABC)* application, an improved BitTorrent client. ABC is based on *BitTornado* which extended from the *BitTorrent* core system, originally written by Bram Cohen. ABC was shipped with an user interface and a variety of features to manage BitTorrent downloads. The software makes use of the BitTorrent engine, at that time completely written in Python like the rest of ABC. This is most probably the reason why Python is used for Tribler development.

In 2007, the first major research paper was published, describing Tribler as a social-based peer-to-peer system[42]. The key idea as described in the paper 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. Free-riding is a phenomena that is widely observed in the regular BitTorrent network. The system architecture of Tribler as described in the work of Pouwelse et al. is presented in Figure 3.1. We will now highlight the most important parts of the given architecture.

3.1.1. Collaborative Downloads

The BitTorrent engine provides tools to download and seed files in a decentralized way using a BitTorrent-compatible protocol. In addition, the module allows usage of the *collaborative downloader* feature which significantly increases download speed by exploiting idle upload capacity of online friends in the network. The implemented protocol to facilitate these collaborative downloads is called *2Fast* and it uses social groups where members who trust each other collaborate to improve their download performance.

[1] http://repository.tudelft.nl

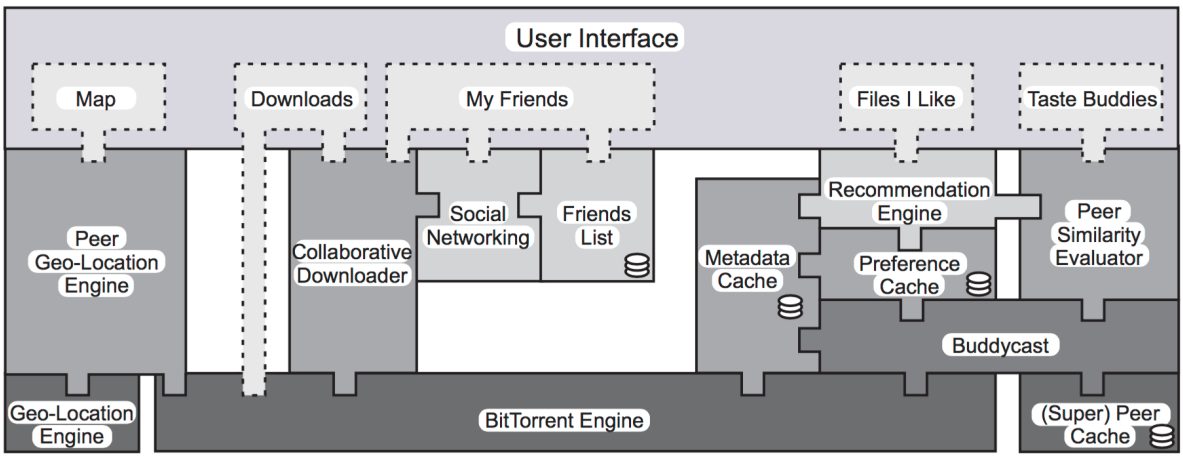


Figure 3.1: The system architecture of Tribler as described in [42].

The protocol works as follows: 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 whereas a helper is a peer that is recruited by a collector to help downloading that file. Both types of peers start downloading a file using the regular BitTorrent protocol and the collaborative download extensions. However, before a helper tries to fetch a file piece in the network, it first asks the collector for approval which is granted when no other helpers have downloaded or are downloading the file piece in question already. Afterwards, the helper peer sends the piece to the collector. For ADSL and ADSL-2 internet connections, the maximum achievable speed-up is 4 and 8 times respectively.

3.1.2. Geo-Location Engine

In the left side of the architecture in Figure 3.1, we notice the *Geo-Location Engine*, *Peer Geo-Location Engine* and the map component in the user interface. The *Geo-Location Engine* is used to determine the location of other peers in the torrent swarm, using an open API[2]. The *Peer Geo-Location Engine* has been built on top of this module, providing the primitives to display the location of peers on a map in the user interface. This feature stems from the goal to ease the process of visual identification of potential collaborators.

3.1.3. Content Discovery and Recommendation

The *BuddyCast* algorithm is designed to serve recommendations to users 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 content preference lists and a number of random peers, void of any information about their content preferences. Periodically, BuddyCast performs an *exploration* or *exploitation* step: when an exploration step is executed, the peer connects to one of its taste buddies. When an exploitation step is performed, the peer connects to a random peer in the network. When the connection with the other peer is successful, a *BuddyCast* message is exchanged, containing the identities of a number of taste buddies along with their top-10 preference lists, a number of random peers, and the top-50 content preferences of the sending peer. The age of each peer is included in the message to help other users know about the "freshness" of peers. After the BuddyCast messages are exchanged, the received information is stored in the local database of each peer, called the *Preference Cache*. Information about discovered peers are stored in the *Peer Cache*. To limit the exchange of redundant information, each peer maintains a list of recently contacted peers.

The BuddyCast mechanism interacts with the user interface in two different ways. On the *Files I Like* page in the interface, each peer indicates its preference for certain files expressed as a score between 1 and 5. Initially, this list is filled with the most recent downloads of the peers. Second, the user interfaces displays similar taste buddies and facilitates a content browser where each item is annotated with an estimated interest indicator

[2] http://hostip.info

for that user.

3.2. Tribler between 2007 and 2012

The first version of Tribler in the 4.x series, Tribler 4.0, is released in 2007[5]. Many features from the 3.x release cycles are untouched and some new functionality have been added, most notable in the user interface. With a new embedded video player, users can play videos (while being downloaded) directly from within the user interface. This video player is powered by the popular *VLC* library[3] and bindings that facilitates video management and playback in several popular user interface libraries. Tribler 4.0 allows users to remotely search for content inside the Tribler network but also searches content available on YouTube and Liveleak. These search results are presented to the user in a YouTube-like thumbnail grid. The user interface of Tribler 4.0 is displayed in Figure 3.2.

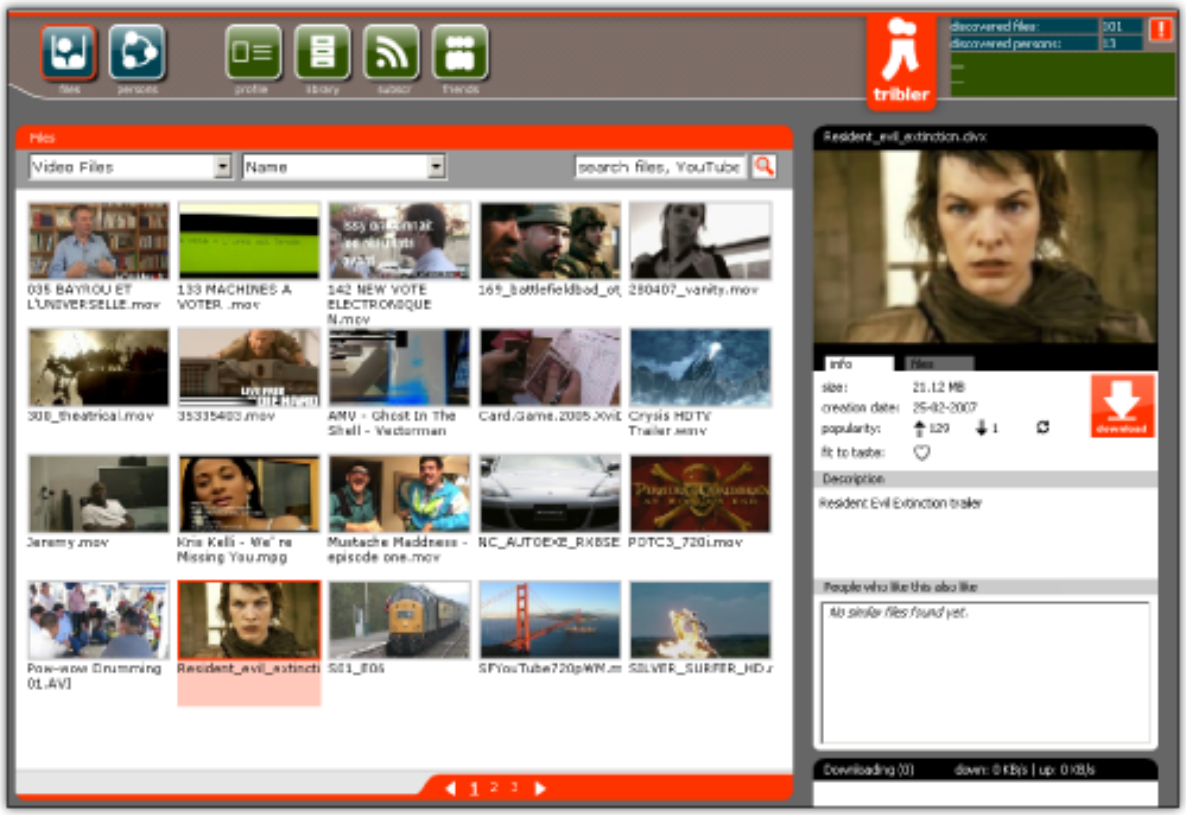


Figure 3.2: The user interface of Tribler 4.0.

The development of Tribler continued with the release of version 5.0 in 2009[7]. The user interface has been subject to a complete redesign, introducing a more dark theme, which was replaced by a white theme shortly after release. The focus of Tribler 5.0 has been on the stability and performance of remote content search and the download mechanism. The thumbnails have been dropped in favour of a paginated list.

Tribler 5.1 contained some major improvements to the user interface, thanks to the feedback of the community. A new, novel addition in Tribler 5.2 is the concepts of channels, similar to YouTube. One goal of the organization of content into channels was to prevent spam inside the network by favouring content present in more popular channels. Whereas custom widgets with an own look-and-feel has been used in this version, they all got replaced in Tribler 5.3 by native buttons to creating a more natural feel on each supported platform. Additionally, a tag cloud with popular keywords has been added to the home page of Tribler to help users determine which content they possibly want to look for. The paginated list was replaced by a single, scrollable list of items. In the next release, Tribler 5.4, a *magic search* feature has been implemented where

[3] http://www.videolan.org/vlc/

similar search results are collapsed using text similarity functions and digit extraction. The usefulness of this feature is apparent when searching for content that is split into many parts such as a sequel of books or a television show. This idea is that this feature leads to a much cleaner and comprehensive results list when searching for content.

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, providing benefits to the compatibility with *Network Address Translation (NAT)* firewalls. Tribler adopted the *Peer-to-peer streaming peer protocol (PPSPP)* protocol[14], implemented in *libswift*, PPSPP provides download capabilities over UDP, thus removing the TCP layer from the BitTorrent engine.

The architecture around the time of Tribler version 5.5 is visible in Figure 3.3. We notice that this architecture is significantly more complex compared to the design as presented in Figure 3.1. The depicted architecture excludes the user interface architecture which is equally complex. This model is the result of gradually adding and modifying smaller components to Tribler that have been developed during research, such as 2fast, BuddyCast and the *Secure Overlay*, providing a high-level communication mechanism. We notice four different threads that need to work together, contributing to the complexity of the code since developers need to be aware of context switches (jumping between different threads during execution of the application). Whereas the project contained around 45.000 lines of code in 2007, this number has increased to over 90.000 in 2010.

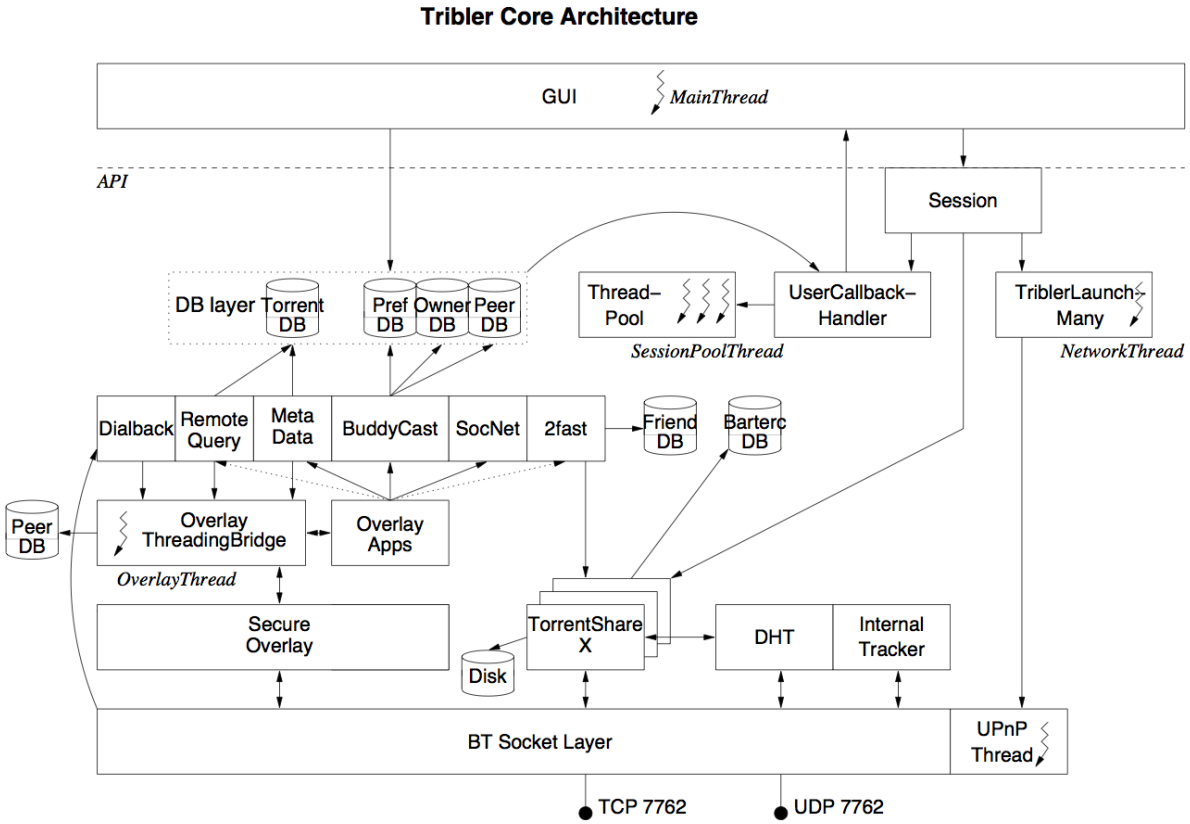


Figure 3.3: An overview of the core architecture around the time of Tribler 5.5.

3.3. Tribler between 2012 and 2016

Shortly after the release of Tribler 5.9, version 6.0 is released where a complete new user interface has been implemented and the PPSPP protocol implementation is replaced by the *libtorrent* library, written in C++[4]. This release also contained some minor bug fixes that increased performance and usability in general. After the release of version 6.0, several smaller releases (6.1, 6.2 and 6.3) followed. The focus of the Tribler

[4]https://github.com/arvidn/libtorrent

Community name	Purpose
AllChannel	Used to discover new channels and to perform remote channel search operations.
BarterCast4	While currently disabled, this community was used to spread statistics about the upload and download rates of peers inside the network and created as mechanism to prevent free-riding in Tribler[37].
Channel	This community represents a single channel and is responsible for managing torrents and playlists inside that channel.
Multichain	The Multichain community utilizes blockchain technology and can be regarded as the accountant mechanism that keeps track of shared and used bandwidth.
Search	This community contains functionalities to perform remote keyword searches for torrents and torrent collecting operations.
(Hidden)TunnelCommunity	The (hidden) tunnel community contains the implementation of the Tor-like protocol that enables anonymity when downloading content and contains the foundations of the hidden seeder services protocol, used for anonymous seeding.

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

platform shifted toward anonymous downloads and end-to-end encryption, designed and implemented by Plak, Tanaskoski and Ruigrok in 2014 and 2015[40][50][44]. The Tribler 6.4 release provided an experimental anonymous download mechanism and hidden seeding services implementation. Additionally, the release also introduced the *Trivial File Transfer Protocol (TFTP)*, a simplistic version of the popular *File Transfer Protocol (FTP)*. TFTP in Tribler is utilized to exchange torrent files between peers in Tribler when searching for content. The next release, Tribler 6.4.1, contained some major security fixes after an external code review by a member on the Tor mailing list[10].

3.3.1. Dispersy

With the release of Tribler 6.1, *Dispersy* got introduced. Described in [54] and mainly developed by Zeilemaker and Schoon, Dispersy lies at the foundations of the messaging and synchronization system in Tribler and is designed to deliver messages reliably in unpredictable networks. It provides a NAT traversal mechanism to improve the extent to which users are connectible in the network. Dispersy provides tools to quickly create distinct overlay networks, called *communities*, that peers can join and where messages be exchanged. The communities as utilized by Tribler, together with a short description, is presented in Table 3.1. While being a major dependency of Tribler, a throughout 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[5]. Twisted allows programmers to write code in an asynchronous way. The utilization of Twisted has been motivated by the presence of callback mechanisms in Tribler as can be identified in Figure 3.3. The library provides a simple model for handling callbacks and events. At the heart of Twisted, we find the *reactor* which is the implementation of the event loop[12]. The event loop is a programming construct that waits for and dispatches events or messages in an application. The threading model as available in Tribler 6 has been illustrated in Figure 5.10. In most applications that are using the Twisted library, the reactor operates on the main thread of a Python application. In Tribler, the reactor runs on a separate thread since the main thread is occupied by the event loop of *wxPython*, the library used to built the user interface. This means that code performing operations with the user interface such as refresh operations of lists, should always be executed on the Python main thread. Operations that are using Twisted constructs however, should be scheduled on the reactor thread to function correctly. The Twisted threadpool provides a pool of additional threads to dispatch work to and can be utilized for longer-running operations that should not block the main or reactor thread. To make context switching more easy to implement, several method decorators have been implemented by Tribler developers, visible in Figure 5.10.

[5]https://twistedmatrix.com/trac/

Developers should always 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 architecture reduced the number of threads if we compare it to Figure 3.3, we are still stuck with a dedicated thread for the reactor and complex thread switching patterns.

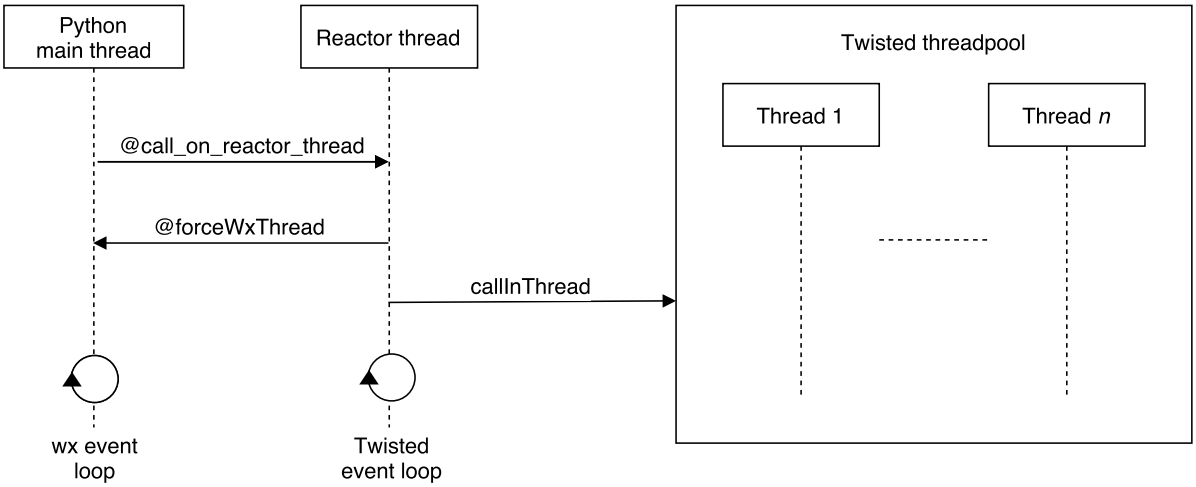


Figure 3.4: The threading model used by Tribler 6, together with the primitives to schedule operations on different threads.

3.4. The roadmap of Tribler

In the previous Section, the evolution of Tribler has been discussed. We have illustrated the increased complexity over time in terms of the architecture, design and threading model. We now turn our attention to the future of Tribler and propose a new architecture where we address some of the design flaws introduced in previous development iterations of Tribler. This new architecture should prepare Tribler for another ten year of research. We start by defining three requirements that our new architecture should meet:

- *simplicity*: we wish to shift towards an architecture that has a better learning curve for new developers. The current architecture is hard to learn, prone to errors and has a complex threading model, increasing the time for new developers to get familiar with the code. By modifying the architecture to make it simpler, we can save developer time and increase the chance for contributions from external developers outside the Tribler organization.

  Creating a simplified architecture inevitably leads to decisions to remove unnecessary, unused or broken components. The current architecture has various features for which we believe that the maintenance costs outweighs the benefits of that particular feature for end users. Making decisions to remove components and thus code, can lead to a significant reduction of technical debt.

- *flexibility*: by introducing a substantial level of flexibility in the system, developers can focus on the development of individual components when contributing to Tribler. While we can identify many different modules in Figure 3.3, there still are numerous interdependencies between them, making it hard to modify and test parts of the system in isolation. We propose a methodology based on component-based software engineering, where we provide interfaces between components to communicate with each other. The user interface should be implemented as a separate component, in comparison to the current architecture where the core and user interface cannot be used as separate modules (this will be elaborated in more detail in Chapter 5.5.1).

- *performance*: Tribler contains various time-critical components that should work reliably and fast such as the download engine, the anonymous overlay and the content discovery mechanisms. Several development tasks have been conducted, focused on performance optimization of individual components or the system as whole.

The new architecture should be designed with future performance engineering in mind. An unclear and unstructured architecture can cause much overhead for developers when boosting performance as is for instance illustrated by the numerous thread switching necessaries to implement a specific feature that utilizes both the Tribler core and user interface. By considering performance engineering as early in the process, we can build our architecture to allow for performance modifications.

We propose the architecture depicted in Figure 3.5 which design follows a layered, component-based approach. The remainder of this Section will discuss the components in the architecture in more detail and highlight decisions that have been made during the design process.

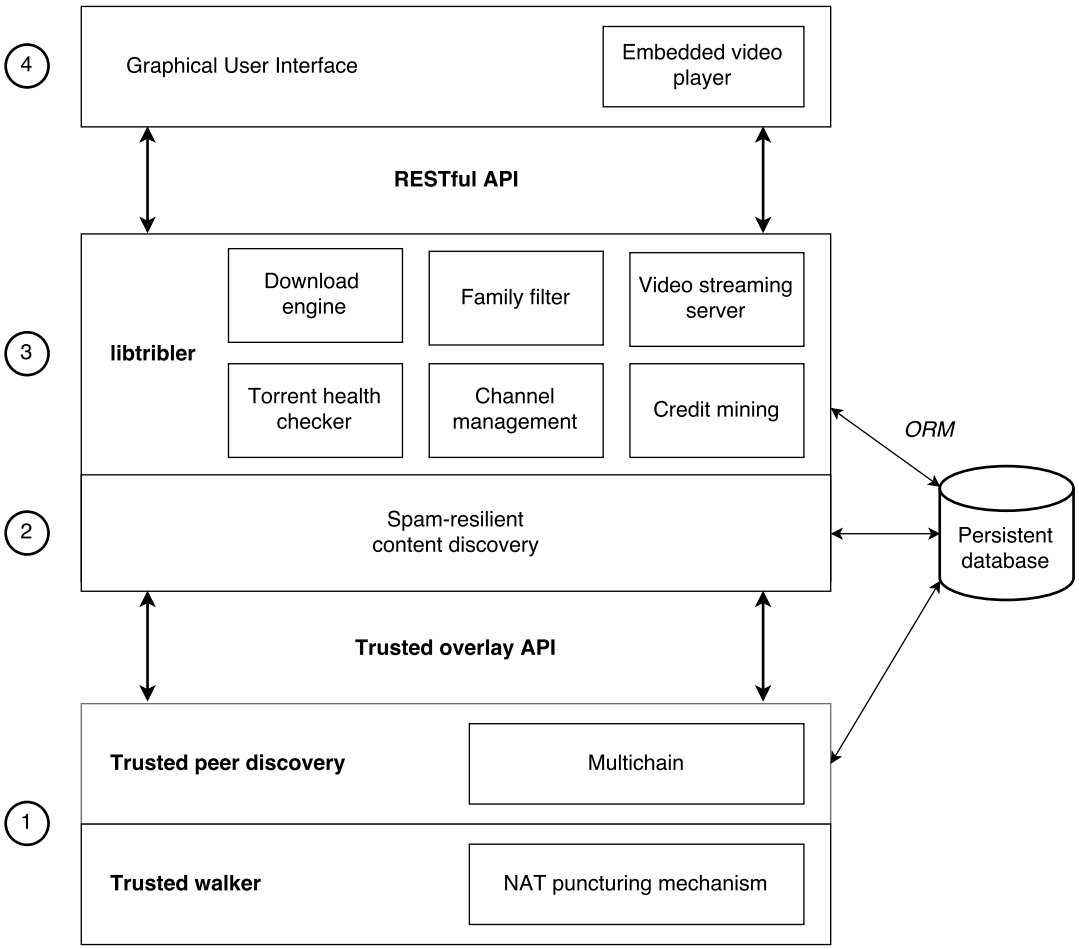


Figure 3.5: The proposed architecture of Tribler 7, consisting of a trusted overlay (1), a content-discovery mechanism (2), *libtribler* (3) and a user interface (4).

3.4.1. Trusted Overlay

The trusted overlay is the lowest layer in Tribler that provides primitives for discovering and picking new trusted peers in the network. At the lowest level of the trusted overlay, we identify the trusted walker, the central component for discovering other peers. Currently, the Dispersy framework is responsible for discovering new peers within the Tribler network, using a gossiping protocol[54]. This discovery mechanism is illustrated in Figure 3.6 and executed at fixed time intervals. It works as follows: 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 *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 firewall of node *A*, making sure that node *A* can connect to him. This algorithm both provides a NAT-puncturing mechanism and allows a node to discover new peers.

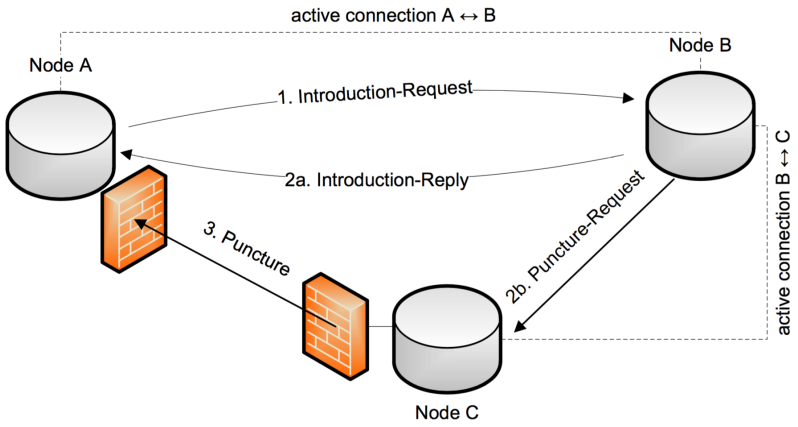


Figure 3.6: The discovery and NAT puncture mechanism as implemented in Dispersy.

The described mechanism of discovering new peers using Dispersy will be replaced by a trusted walker that uses accumulated reputation in the *Multichain*, the accountant mechanism that keeps track of shared and used bandwidth, providing more reputation when a user provides upload capacity to help other users. The key idea is that *sybil* nodes, forged identities in the network, and free riders are ignored and not considered as a trusted peer since their reputation is low and are not likely to be selected by the trusted walker.

New peers in that network that have not accumulated any reputation yet, start out by creating some random interactions with other nodes while learning about the network and the amount of reputation of other users. With an interval, every node runs an algorithm to calculate the reputation of their 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 and thus a higher trustworthiness. This leads to a trust network where each node knows about other trusted peers with which they can exchange content.

Interaction with the trust overlay can be realised by using an higher-level Application Programming Interface (API) which provides the facilities to perform operations regarding the discover of new trusted peers and management of known ones. By providing a trusted overlay API, the component allows for easy reuse which is beneficial when the module will be published as an open-source project.

3.4.2. Spam-Resilient Content Discovery

Discovering content is a key feature of Tribler. The spam-resilient content discovery component allows users to discover and search torrents, channels and playlists in Tribler using peers are managed by the trust overlay. The current implemented mechanism for information exchange in Tribler where messages are exchanged within segregated communities works well enough for this purpose, except for the exchange of more exotic meta data such as content thumbnails. Providing users with a visual preview of content in the form of thumbnails is a good way to make the user interface more appealing, however, a robust implementation in a complete decentralized network is challenging due to the fact that the content can be present in a huge volume, thus increasing the total size of the 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. These features will be considered future work and not be discussed in the remainder of this thesis.

3.4.3. libtribler

libtribler provides the primitives to developers to make use of the above described components and contains the implementation of a RESTful API that is used to communicate with libtribler. We will now discuss the components which together account for this layer.

Download Engine

The download engine is one of the most crucial parts in Tribler: before facilitated by the BitTorrent and libswift libraries, we currently use the open source *libtorrent* library to facilitate decentralized downloads. libtorrent is written in C++, however bindings for various other programming languages such as Python, Go and Java are available. *libtorrent* uses an alert mechanism to notify the application that is using the library about events, such as download state transitions, peer discovery in the torrent swarm or completion of a meta info lookup in the *Distributed Hash Table (DHT)*. There are no solid arguments for replacing the current download engine with another library that allows for decentralized and anonymous downloads. Moreover, the current way *libtorrent* is used in Tribler requires minimal changes to adhere to the proposed design, except for some optional refactoring of the current code.

We should note that the method to fetch peers from the DHT in libtorrent is private and not accessible from Python code. Under normal circumstances, this method only invoked by libtorrent, however, we manually call this method when performing a DHT lookup on behalf of another peer in the hidden services protocol, described in more detail in [44]. To still make it possible to perform a lookup of peers in the DHT, we 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 extra complexity and load of the system. Effort should be made to make this desired method accessible in libtorrent so Tribler can get rid of the dependency.

Streaming Engine

The video streaming component streams the video data to a video player outside *libtribler* after or during a download. The implemented video server in the current architecture listens for and serves HTTP requests and is implemented using the *SimpleHTTPServer* library[6], a built-in Python module that can be used to implement a HTTP web server. The functioning of the video server is based on HTTP range requests, allowing the web server to serve a subset of a file when the user navigates to a random time position in a video.

The inner workings of the video player is illustrated in Figure 3.7 and works as follows: when the user starts playing a video in an external video player, the player performs a HTTP range request to the video server implemented in Tribler. This range request contains information in the header about the requested range of the video file. When the video server receives the request, it first checks whether the requested range has been downloaded by the download engine already. If so, the server returns the requested data to the client. If the requested range is not available, the video server notifies *libtorrent* that the bytes in the requested range should be prioritized for download, reducing the latency before the requested range is completely available. When all pieces are downloaded, libtorrent notifies the video server about this event and the video server completes the request by sending the data to the client.

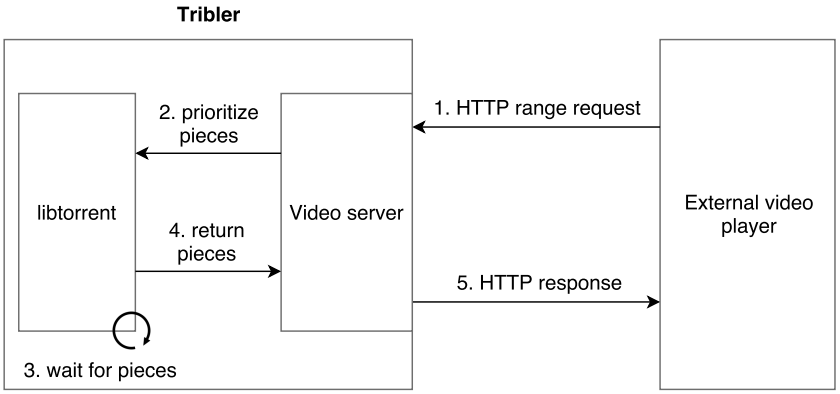


Figure 3.7: The flow when performing a HTTP range request when streaming a video using Tribler.

While the video server in the current form is functional, there is a performance improvement that can be considered: the video player runs on a separate thread and uses blocking calls to wait for the requested range to be downloaded. By integrating the video server inside the Twisted reactor thread, we can reduce the

[6] https://docs.python.org/2/library/simplehttpserver.html

complexity of the server and utilize all of the facilities that Twisted provides, for instance, managing incoming requests. An additional consideration could be 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.

Family filter

The freedom to upload any type of content in the network, comes with a price. The legal aspect of the available Tribler content can be disputable. While Tribler contains legal content such as pornographic material, that content might be undesirable for most casual users. A mechanism called the family filter is implemented in Tribler to filter out content that is not always safe for display. This filter is enabled by default and uses a list of keywords that can be associated with pornographic content. Discovered content gets classified by this filter, based on the torrent name, file names and other meta data. Unfortunately, this ad-hoc approach is not very effective since there are quite a few false positive classifications. While it provides some basic filtering tools, we noticed that the keyword-based approach can be greatly improved by using a more sophisticated classification approach. However, we would consider this as an enhancement rather than a defect that prevents a correct usage of Tribler.

Credit Mining

Ongoing work on a *Credit Mining System (CMS)* in decentralized systems has been extensively described by the work of Capotą et al[18] and is defined as the activity performed by peers for the purpose of earning credit. A possible purpose of the earned credits is to access exclusive content or receiving preferential download treatment in case of network congestion. Although the credit mining component is not enabled for end-users, a CMS has been implemented in Tribler, 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 CMS to take in consideration. Possible sources are Tribler channels, RSS feeds or a directory containing one or more torrent files. Next, the CMS periodically selects a subset of the chosen swarms by the user. Finally, Tribler joins the swarms and tries to maximize earned credits by downloading as little as possible and maximizing the amount of uploaded data.

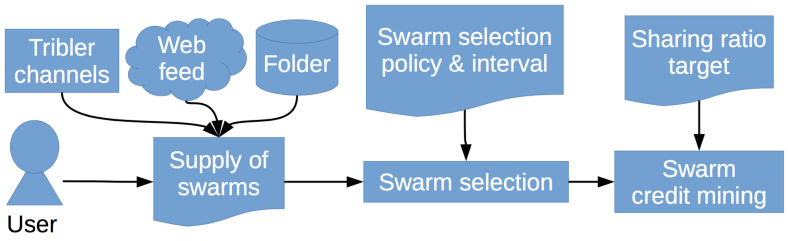


Figure 3.8: The credit mining system described in the work of Capotą et al.[18]

The CMS can apply different policies for swarm selection. The first policy is to selects a swarm with the lowest ratio of seeders to all peers (leechers and seeders). Intuitively, this boosts swarms that are undersupplied (having a low amount of seeders). The second policy is to select swarms based on the swarm age. The intuition behind this policy is that newer content is often better seeded so it might be more beneficial to boost older swarms. The final policy that can be used is a random policy that selects a swarm using an uniform distribution.

This credit mechanism is a convenient way for users to increase their reputation by supplying bandwidth to the community, requiring minimal intervention of the user. The CMS works in conjunction with the Multichain which can be used as accounting tool to keep track of downloaded and uploaded bytes.

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 is composed of a bundle of potential 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 should be facilitated by a *Representational State Transfer (REST)* API. The REST architecture was introduced and defined by Roy Fielding in 2000[25] and is used frequently when building APIs that operate on the World Wide Web. A service that conforms to the REST architecture, is called RESTful.

In a RESTful architecture, resources and collections, identified by a Uniform Resource Identifiers (URIs) are returned by servers to clients. Common HTTP verbs (*GET*, *POST*, *PUT* and *DELETE*) are used to manipulate or retrieve these resources and collections. Table 3.2 provides a summary of the most common operations that are used in a RESTful API.

Resource type	**GET**	**PUT**	**POST**	**DELETE**
Collection	Retrieve 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 single item.

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

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

3.4.5. Graphical User Interface

At the highest level of the Tribler architecture stack, we find the user interface. The user interface should be able to communicate with the Tribler core using the RESTful API as described in the previous Subsection. A 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, these bindings are not functional on MacOS 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 4.

3.4.6. Requirements Conformance

In Section 3.4, we defined three requirements that our architecture should meet: simplicity, flexibility and a focus on performance. After the proposal and discussion of the new architecture, we will now evaluate to what extent our proposed design meets the requirements we composed.

Simplicity

Our first requirement was simplicity. The architecture as depicted in Figure 3.3 is complex and has a steep learning curve for new developers. The threading model as present in Tribler 6, requires developers to be sufficient aware about the thread on which specific operations should be scheduled, possibly leading to confusing code. The proposed architecture is simpler by design and more divided into separate components,

increasing re-usability and testability. When modifying a specific component in a layer, developers should not have to care about the layers below the layer that is being modified. In this sense, the architecture meets our requirement that it should be more comprehensible and usable for developers.

Flexibility

The APIs that are facilitating communication between the layers in the architecture, increases the flexibility. As described in Subsection 3.4.4, the RESTful API allows a great amount of flexibility for developers. We strive towards an implementation where individual components can easily be toggled using a configuration file by developers. Some components should also be configured by users, such as the credit mining mechanism.

Performance

One addition reason to split the architecture into different components, is based on performance engineering efforts performed on the system. By having components that are communicating with each other through an API, we are able to more easily extract and refactor these parts of the system in separate processes later on, thus increasing performance since different processes can utilize more CPU cores.

4

Towards a new architecture

The discussion in Chapter 3 concluded with a proposal of a new future-proof architecture. Now, we will shift our focus to performed efforts on making this design a reality. This will involve work on components that can be found on higher levels in the proposed architecture, in particular the user interface, the REST API and libtribler.

4.1. REST API

As described in Chapter 3, communication between the user interface and the core of Tribler is facilitated by a RESTful API in our design. This Section explains the implementation details of the API in more detail.

The REST API has been implemented using facilities in the Twisted library. While there are plenty of Python solutions available that allow developers to create a web server in their application, we made the choice to use Twisted since it is already utilized to a great extent by Tribler. With the ability to integrate the REST API into the main application flow, we avoid having to create special constructions to execute the API on a separate thread like we are doing now with the video server. The API internally in Twisted is represented as a tree of resources which is in accordance with the REST architecture where the URL of the request can be treated like a path in the resource tree. If we visualize the import graph of the API package, this tree-like structure is somewhat visible, see Figure 4.1. We will highlight and discuss some important files in the API package:

- *rest_manager.py*: this file contains the *RESTManager* class which is responsible for starting and stopping the API. In addition, it 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. Error handling in the API is discussed in Chapter 4.1.2.
- *root_endpoint.py*: this file hosts the *RootEndpoint* class which represents the root node of our resource tree. This class dispatches all incoming requests to the right sub nodes in the resource tree.
- *util.py*: this file contains various helper functions, such as conversion utilities to easily transform channel and torrent data from the database into *JavaScript Object Notation (JSON)* format that can be sent to the client that initiated a request.

4.1.1. Response Format

Most 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. However, in some occurrences we do not return JSON but instead, binary data. Examples of this are when exporting a torrent file where we return the content of the torrent file instead. While introducing some kind of inconsistency in the API, it allows for easier management of the response object since developers do not have to decode and parse the JSON object first.

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 asynchronous stream of data

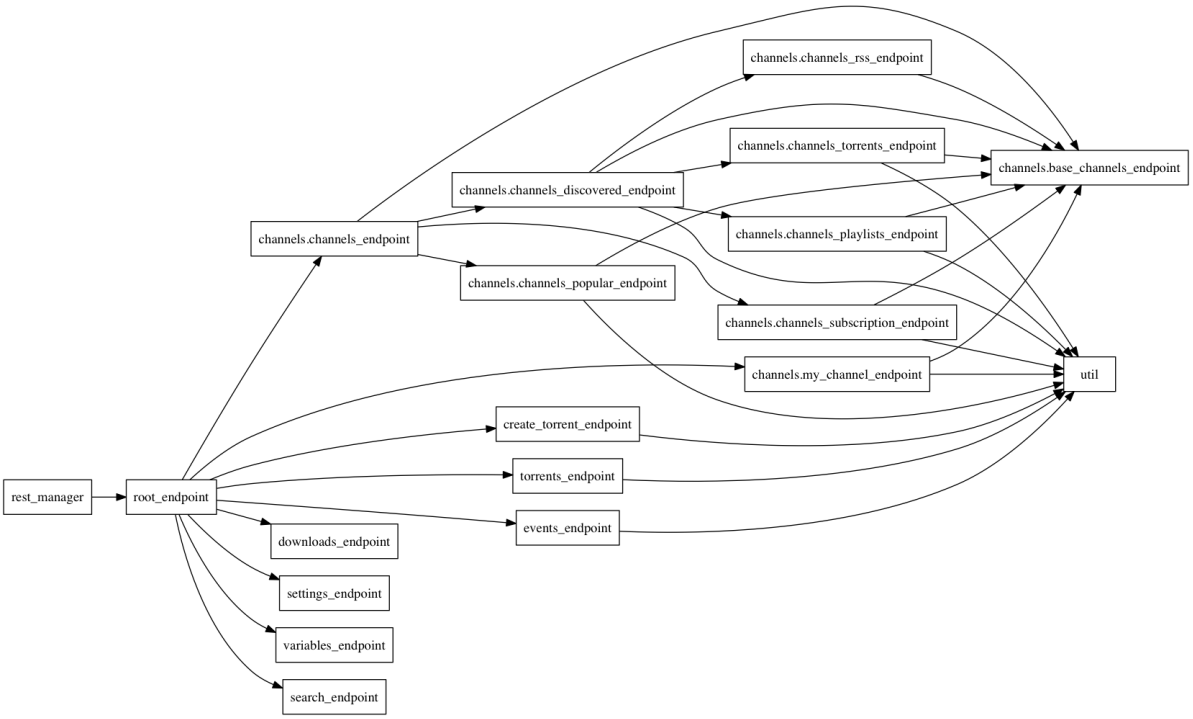


Figure 4.1: The import graph of the REST API module.

should be returned. For instance, this is the case when the user performs a search query. In some occurrences, 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. This motivates the design of an asynchronous events stream that notifies clients about interesting events in Tribler. This event stream has been implemented and all messages that are sent over the *events* connection are shown in Table 4.1. Developers can add new events with only a few lines of code.

4.1.2. Error Handling

A proper designed API should have a mechanism to notify users about any internal errors that occurred during requests. Our API returns HTTP response code 500 (*internal server error*) when we observe a Python exception during a request. Moreover, we return a JSON-encoded response that contains more specific information about the caught exception such as the name of the exception, whether the error has been handled by the core and if available, the stack trace of the exception. An example of an error response is displayed in Listing 4.1.

Listing 4.1: The response in JSON format returned when a Python exception is observed during the processing of an API request.

```
{
  "error": {
    "message": "integer division or modulo by zero",
    "code": "ZeroDivisionError",
    "handled": false,
    "trace": [
        " File \"/Library/Python/2.7/site-packages/twisted/web/server.py\",
        line 183, in process\n    self.render(resrc)\n",
        ...
    ]
  }
}
```

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 from the local database). The event contains the channel result data.
search_result_torrent	Tribler received a torrent search result (either remote or from the local database). The event contains the torrent result data.
upgrader_started	The upgrade procedure in Tribler started.
upgrader_tick	The status of the Tribler upgrader changed. This event contains a human-readable string with the status update, usable for display in a user interface.
upgrader_finished	The Tribler upgrader finished.
watch_folder_corrupt_torrent	The watch folder module has encountered a corrupt .torrent file. The emitted event contains the name of the corrupt file.
new_version_available	A new version of Tribler is available. The version number is part of the event.
tribler_started	Tribler has completed the start up procedure and is ready to serve HTTP requests on all endpoints.
channel_discovered	A new channel has been discovered. The events contains information about the discovered channel.
torrent_discovered	A new torrent has been discovered. The events contains information about the discovered torrent.

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

4.2. 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 to the user interface have been made, the code base has reached the point where it might be more beneficial to design and implement a complete new user interface. 29,3% of the Tribler code base, excluding Dispersy, is related to the user interface. We now will continue the discussion that has been initiated in Chapter 2 regarding the architecture of the user interface module. First, the structure of the current interface will be described. We will make the consideration between refactoring efforts of the existing user interface or creating and designing a new one. Additionally, encountered design decisions and implementation challenges are presented and discussed.

4.2.1. Analysis of the current GUI

The user interface of the latest version of Tribler, 6.5.2, is unintuitive and cluttered with unused and unnecessary visual elements. There are various spelling errors in buttons and the navigation through the GUI is complex. The interface feels uncomfortable for users that are using Tribler for the first time and there are no instructions or guides for these users. For instance, when users are starting Tribler for the first time, there is no information provided about the content discovery process in the background. This is one of the improvements to the user interface we can consider to make.

When focussing on the code base, we notice that it is full of undesired workarounds and bad coding practices (code smells). Much functionality that should be located in the core module, is present in the code of the user interface package, including a relevance sorting algorithm of search results, code to manage important configuration settings and a large part of the start up procedure. There is no documented structure to be identified throughout the code and we can think of several reasons underlying that. One of these causes is the mindset of developers that the code base of the user interface is subordinate to the code related to core functionalities of Tribler: while it is often true that minor defects in the GUI are less critical than errors in important core functionalities such as the download engine, developer should always strive to write maintainable and well-designed code to prevent technical debt in the long term, a responsibility which is clearly neglected by user interface developers of Tribler. The fact that the GUI has undergone dramatic changes throughout ten years of research and development is an additional reason that led to this unstructured code base. Making short-term decisions were favoured over decisions that benefit the longer-term development process, leading to many code smells and huge amounts of technical debt.

By taking a closer look at the structure of the user interface code base, several files with many class definitions can be found. We already presented the import graph of the user interface code base in Figure 2.4 where we identified many cyclic dependencies. While cyclic dependencies are not always undesired at the granularity of a class file, we are dealing here with a web of dependencies between files, each file possibly consisting of multiple class definitions. Automated testing of individual classes has become significantly more involved with these dependencies.

We now turn our attention to the layout of user interface, where we use the wxPython inspection tool to investigate the widget structure of the interface at runtime. We noticed here that the naming convention of widget elements is unclear and does not represent content that the widget should present. For instance, the *ActivityListItem* class is representing a list item in the left menu of Tribler, however, the name is confusing since the focus of this list item is not necessary a specific activity. We think that something along the lines of *MenuListItem* would be a more appropriate name.

We noticed that the developers of user have attempted to reuse widget elements, especially noticeable in list-related widgets such as headers, footers and list row items. However, we think this should be classified as a failed attempt since the amount of flexibility is too much: by making widgets adapt to many different use-cases in the user interface, the complexity of the class definition increases significantly due to conditional code that is only executed in a subset of all the supported use-cases. We consider this an addition reason for the bad architecture as displayed in Figure 2.4.

It is hard for developers to get familiar with the code base of the user interface and to modify it. We think the most appropriate metaphor of the user interface code base is a big ball of mud:

> *A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.*
> - Brian Foote and Joseph Yoder, Big Ball of Mud[26].

Due to the bad structure, huge amount of code smells and complexity of the code, we think that it is not worth the effort to refactor the code base of the current user interface and that it saves time to work on the design and implementation of a new user interface. However, we should think about a migration plan: while designing a new interface, critical issues should still be fixed in the current user interface code. By guaranteeing a minimal level of maintenance of the current GUI, we are not under time pressures to ship the new interface in a specific release of Tribler. Only when the new user interface is ready, stable and tested, we can remove the old interface.

4.2.2. Choosing the right tools

Before designing and implementing the new interface, we should first decide which library we would like to use. Because our proposed architecture in Figure 3.5 is communicating to *libtribler* using a RESTful API, this library does not necessary has to support the Python programming language. However, to make reuse of code easier and to maintain a consistent system which used the same programming language for all components, we will implement our new user interface in Python, There are plenty of libraries that are suitable. Below, several GUI libraries are summarized, together with a small description.

- *wxPython*[43]: this is the current GUI library utilized in Tribler. *wxPython* is built upon *wxWidgets* and provides the Python bindings to this latter library. The library is cross-platform and one possibility is 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, the incompatible video player on MacOS and the lack of a visual designer, requiring us to specify the complete layout in Python code.

- *Kivy*[47]: the cross-platform library Kivy has been used in the past by Tribler developers, particularly in past attempts to run Tribler on Android[23][45]. A decision to make use of the Kivy library for a new user interface enables us to reuse the interface logic already written for the Android app. The layout of Kivy can either be implemented using *.kv* files or specified in code. While the library is rather new, it has gained significant attention and adoption in the Python community.
- *Tkinter*[34]: the *Tkinter* library is built upon the Tcl/Tk framework and is considered the de-facto GUI library for Python. Like wxPython, Tkinter 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*[49]: *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, well-maintained state. The library has a large documentation base and provides many different 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, widely used when styling websites.

Since the GUI will be an important aspect of Tribler, we wish to use a library that is mature, future-proof, well-maintained, easy to use and offers a large amount of tools so we can reduce the SLOC count that has to be maintained. 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 a visual editor is an enormous advantage since this will mean that we have less code to maintain. In addition, this allows other developers that are not familiar with the Tribler code base to contribute to the GUI. The *Qt* visual designer also offers abilities for internationalization and translation of the user interface in multiple languages. Tasks like these are perfect opportunities for contributions in an open source project and can attract new developers. A screenshot of the visual designer in Qt is visible in Figure 4.5.

4.2.3. Designing the new GUI

Designing a user-friendly interface is a non-trivial task and creating a proper design together with mock-ups, is a thesis task itself. Since the design of the new user interface is important but should not be the main focus of this work, we decide to adopt various design principles of existing applications that contain somewhat similar use-cases of Tribler.

In 2008, two design studies have been conducted on the Tribler GUI by master students[2][3]. These studies have shown the Tribler user interface contains various inconsistencies and lack of visual feedback. Various users reported frustration because of features that did not function as they expected. This is a common phenomena when designing a user interface as a developer: developers tend to assume that users understand concepts integrated in the user interface, such as torrents, channels while this is often not the case. In fact, it is beneficial to have to user interface designed by a specialist, however, this is not possible due to lack of manpower and budget.

Most applications that provide torrent download capabilities have a similar interface: they present a list with downloads and a detail window with specific information about a selected downloads. The old user interface also follows this design when displaying the downloads and we see no reason for now to make significant changes to this. However, Tribler provides more abilities than downloading torrents: browsing through content and managing channels are also use-cases that should be taken into consideration.

We believe that YouTube[1] is an example of application that has a large feature overlap with Tribler, namely the browsing and streaming of videos, managing channels and creating playlists. The home page of the YouTube interface is visible in Figure 4.2. We copy the left menu as present in the current interface and use it in the new GUI, however, we modify it slightly: first, we add an option to open and close the menu by clicking the hamburger menu in the upper-left corner since the menu might clutter the interface and does not always need to be visible. Next, we make use of icons so users can faster identify what the idea behind each menu

[1] https://youtube.com

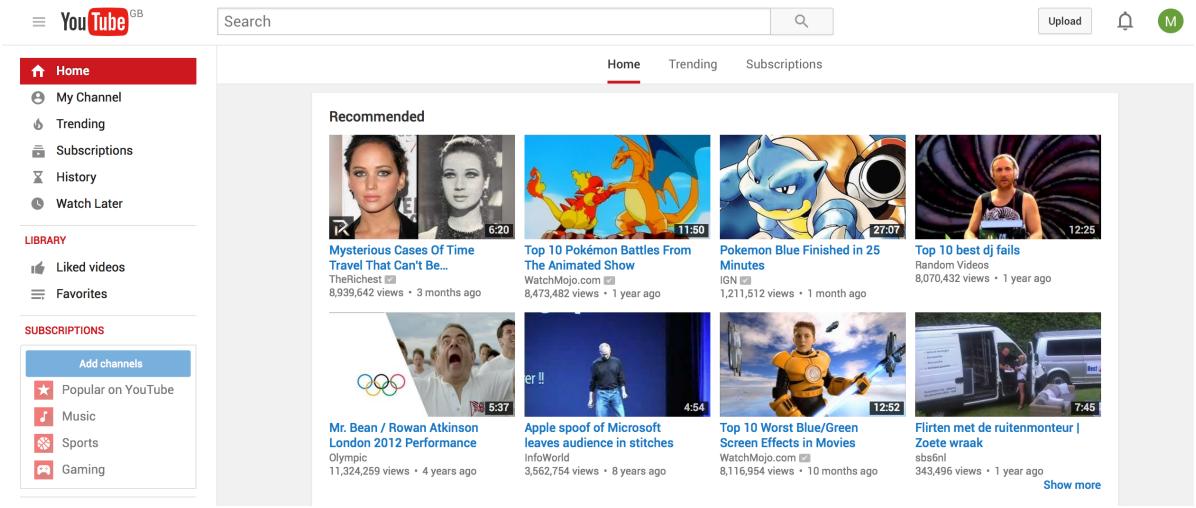


Figure 4.2: The user interface of YouTube.

option is. In general, we are using more icons in the new user interface since we believe that the old GUI contains too textual elements.

To make the user interface more appealing and less boring, we attach a thumbnail to each content item. There is however an issue here: the thumbnails as implemented in the current system are not working correctly. Since the implementation of a new thumbnail mechanism is outside the scope of this thesis, we use thumbnails that are generated based on the content. This is a technique also adopted by popular platforms such as Stack Overflow and Telegram to display a profile image when the user has not uploaded a picture yet.

Finally, we want to get rid of buttons that are only appearing when hovering over content like currently implemented in the old GUI. To make it more clear that a specific action can be performed with content, we do not conditionally hide and show buttons but instead, persistently display these widgets.

Due to time constraints, it is decided to not implement all features available in the old interface. Functionalities such as the debug panel, local content filtering and sorting of content will not be implemented in the first iteration of the new user interface. The focus of this new interface will be centred around searching and streaming content.

Our vision during development of this interface is that in essence, it should only display data and do as few processing operations on this data as possible. This means that various code that is now present in the user interface library should be moved to the Tribler core, such as the search results ranking algorithm.

4.2.4. Implementing the GUI

The result after implementation of the new GUI is visible in Figure 4.3. The majority of the new user interface has been built using the visual designer, part of *Qt*. The Python code that we had to write is responsible for handling requests to the Tribler core, displaying the right content in lists and to manage interface-related settings.

A key feature of the *Qt* library is the signal-slot mechanism which facilitates communication between code and widgets. Widgets in *Qt* can contain signals, events they want to broadcast. Some widgets have built-in signals, for instance, a button emits a *clicked* signal if the user clicks on this widget with the mouse. Other widgets or objects in Python can subscribe to these signals and perform specific actions when the signal is observed. Signals and slots can either be created using code, or designed in the visual designer. To keep the amount of written code to a minimum, we decided to specify our widgets connections in the visual designer as much as possible.

During development of the user interface, we encountered various interesting implementation challenges

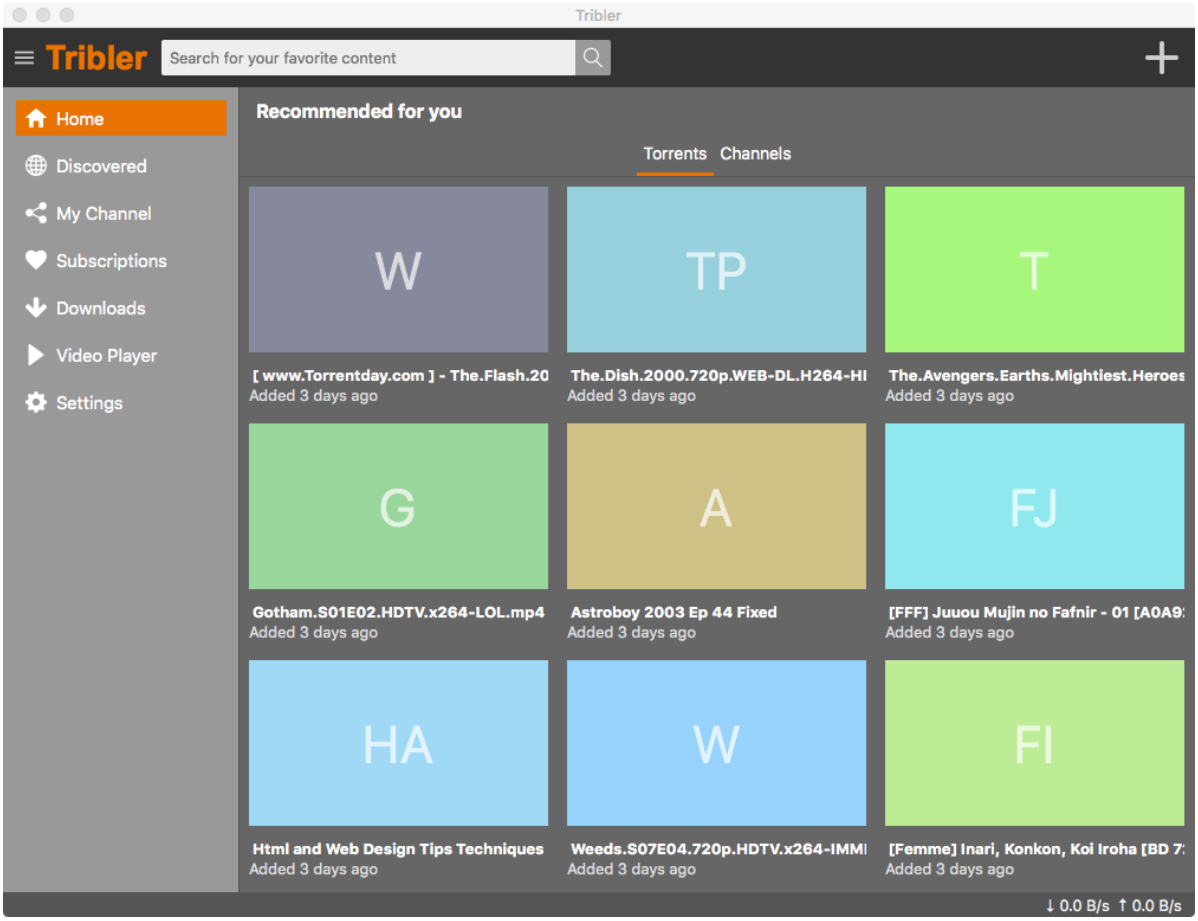


Figure 4.3: The home page of the new Tribler GUI.

that required some form of analysis. We will now present these issues and discuss them.

Scalability of list items

The *Qt* framework allows to display potentially many items in a simple list. The performance decreases dramatically if custom widgets are rendered in a list. Loading 1.000 of such list items takes over 22 seconds on a high-end iMac computer which is an unacceptable amount of latency when displaying a list with content. For each item in the list, the associated interface definition file has to be read, parsed and rendered, possibly many times per second. Channels hold potentially several thousand of torrents which should be presented in the GUI in a timely matter.

This scalability bottleneck has been solved by utilizing a simple technique: lazy loading. By taking advantage of a lazy loading approach where additional data is loaded in chunks (30 items in our new interface) when the user has scrolled to the end of a list, we can postpone and possibly avoid loading the whole list at once. This solution has also been implemented in the old user interface. 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, at the cost of a small delay when the end of the list is reached. The implementation of this lazy-loading solution is general enough to be reused for any type of list in Qt and is located in the *lazyloadlist.py* source file. This implementation however, still resulted in a significant period of waiting when the next set of items is being loaded, around one second. It turned out that loading and parsing of the interface definition file is a time-consuming operation. The solution to reduce this processing time is to pre-load the interface definition as soon the user interface starts. This has a minor effect on the total start-up time (around 40 milliseconds on a high-end iMac device). This the latency when loading additional items to several hundred milliseconds.

A functioning multi-platform video player

One of our main focusses of the new GUI is the streaming of videos. The embedded video player in the old user interface did not function correctly on MacOS due to incompatibilities with the wxPython library. The video player has been implemented using the popular VLC library[13], a free and open source cross-platform multimedia player and framework that plays most media files. We started the design of the new GUI 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 in different ways on each platform). The code for this prototype player has been used as basis for the implementation of the video player in the new user interface of Tribler and is available as open-source project on GitHub[22].

4.3. Threading model improvements

In Section 3.3.2, we discussed that the current threading model is complex and prone to implementation errors introduced by developers. The implementation of a new GUI and a RESTful API as described in the last sections, has led to a cleaner and stable threading model. Now that the user interface runs in a separate, dedicated process and thanks to the removal of *wxPython* from Tribler, we are now able to run the Twisted reactor on the main thread. This enables us to get rid of the confusing decorators to switch between the main and reactor thread since from now on, 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 switches. The new, simplified threading model is presented in Figure 4.4.

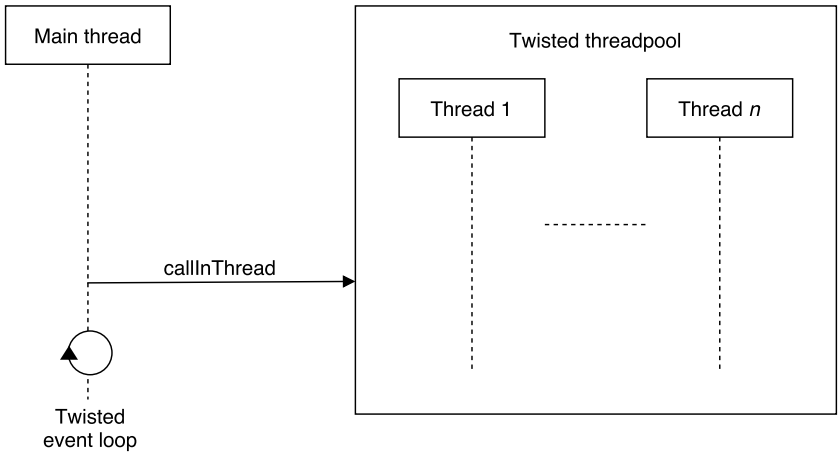


Figure 4.4: The new, simplified threading model in Tribler 7, together with the primitives to schedule operations on the threadpool.

4.4. 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.

wat ermee doen?

4.4.1. 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.

4.4.2. 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[29]. 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.

4.4.3. 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.

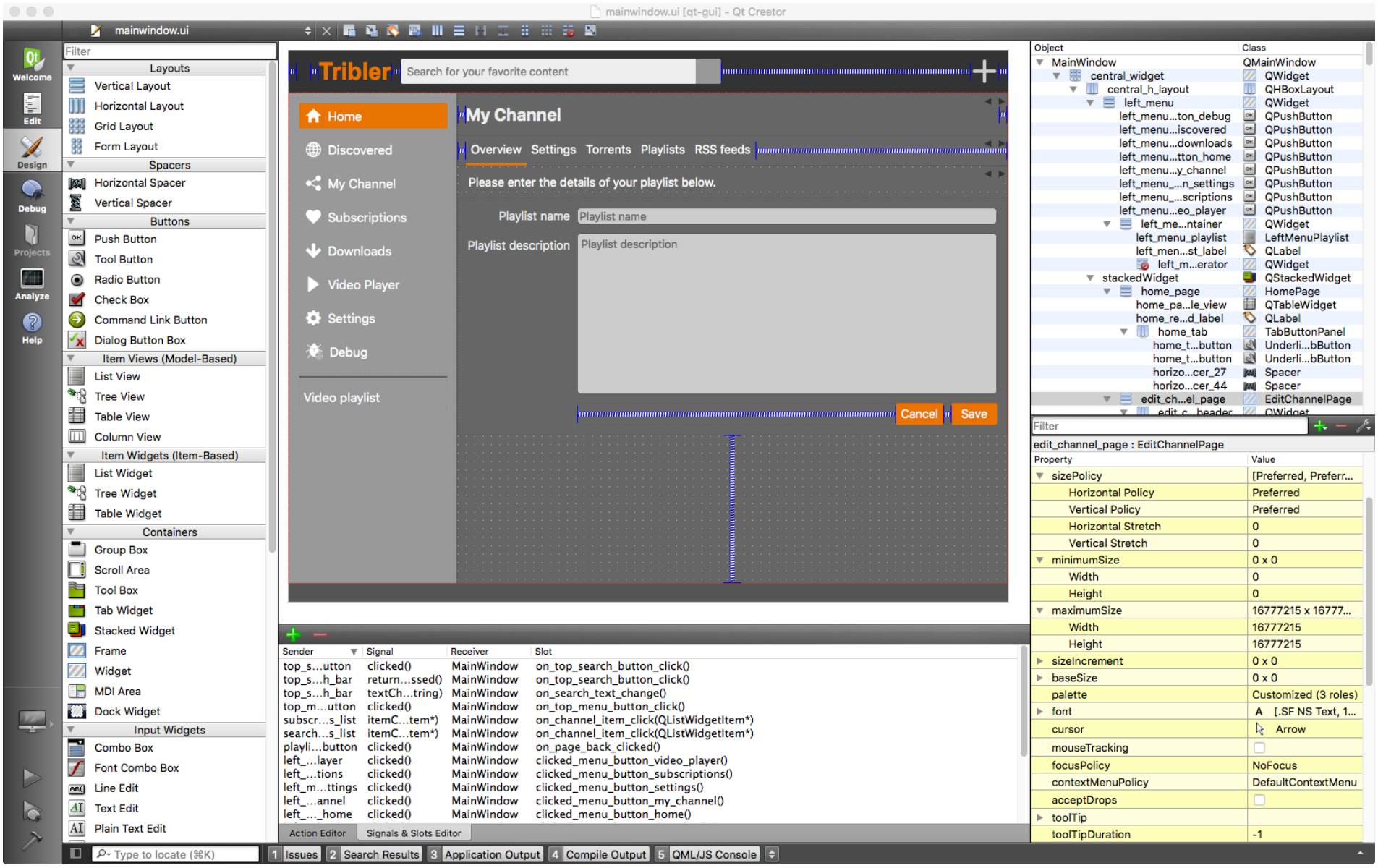


Figure 4.5: The Qt visualizer tool used to create the new user interface of Tribler.

5

Paying off the debt

In Chapter 2, the extraordinary amounts of technical debt Tribler has accumulated over the past decade has been highlighted. We presented the architectural evolution from a historical perspective and proposed a new robust and future-proof architecture in Chapter 3. The top-level components of this new architecture, the GUI and RESTful API to communicate between the user interface and libtribler, have been implemented and discussed in 4. We did however not focus yet on the libtribler component, which will can be considered as the core of the Tribler platform. Shaping libtribler to fit in the proposed architecture of Figure 3.5, requires some invasive refactoring efforts. Since this might be the most important component in our system, we will investigate libtribler in more detail and determine how we can identify and pay off the accumulated technical debt in the core. A summary of the re-engineering efforts conducted during this thesis work is displayed in Table 5.1.

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

Table 5.1: A summary of refactoring efforts as conducted during this thesis work, excluding the work on the new GUI.

Roughly, we can identify five different types of technical debt[46]: *code debt*, *testing debt*, *infrastructure debt*, *architectural debt* and *documentation debt*. Tribler contains symptoms for every of the summarized types of technical debt.

We argue that refactoring in a dynamic typed language like Python is harder than when using a statically type language: information about naming errors is not observed before runtime because of the lack of type information. This issue becomes apparent when for instance trying to rename a variable using a Python Integrated Development Environment (IDE): the application logic might miss various occurrences when performing the renaming operation and we become aware of this issue when either running Tribler itself or when executing the test suite. This shows the importance of having a solid, extensive test suite before we start to refactor major components in the system and we started with creating improvements to create a solid foundation for our refactoring efforts. Improving the test suite first has an additional advantage: by studying the test code, we can familiarize ourselves with the code base and review of the test code may help to us to understand the structure and detect code smells in the production code[52].

5.1. Investigating the debt

It is hard to get accurate numbers on the amount of technical debt. When a decision to pay off technical debt is made, developers might run into unexpected situations that take longer than expected, especially

if working with unfamiliar code. This makes estimations of the amount of work required unreliable. Several tools exist to monitor and estimate technical debt, the most prominent being CAST[1] (commercial) and SonarQube[2] [24] (open source). In this Section, we will use SonarQube to track and identify the amount of technical debt in Tribler by setting up a SonarQube server to identify technical debt, bugs and vulnerabilities in the Tribler project. The reported results are summarized in Table 5.2.

	SLOC	Code smells	Bugs	Vulnerabilities	Estimated debt
wxPython GUI	20.080	2.139	11	0	± 21 days
Qt GUI	2463	14	0	0	± 1 hour
Tribler core (before refactor)	15.732	365	6	0	± 4 days
Tribler core (after refactor)	15.700	117	0	0	± 2 days

Table 5.2: The reported software metrics by SonarQube.

The most interesting observation is that the wxPython GUI contains around five times more technical debt than the Tribler core and contains almost six times more code smells. To better investigate which files suffer from the most technical debt, SonarQube provides a bubble chart where the relation between the amount of technical debt, the number of code smells and the amount of code is visualized. For the wxPython GUI, this bubble chart is visible in Figure 5.1. In this chart, the size of the bubble is correlated to the amount of code smells. The files suffering the most under technical debt are annotated with the file name. By taking Figure 2.4 as reference, we notice here that the files that contain the highest amounts of technical debt, are also the files with the most dependencies.

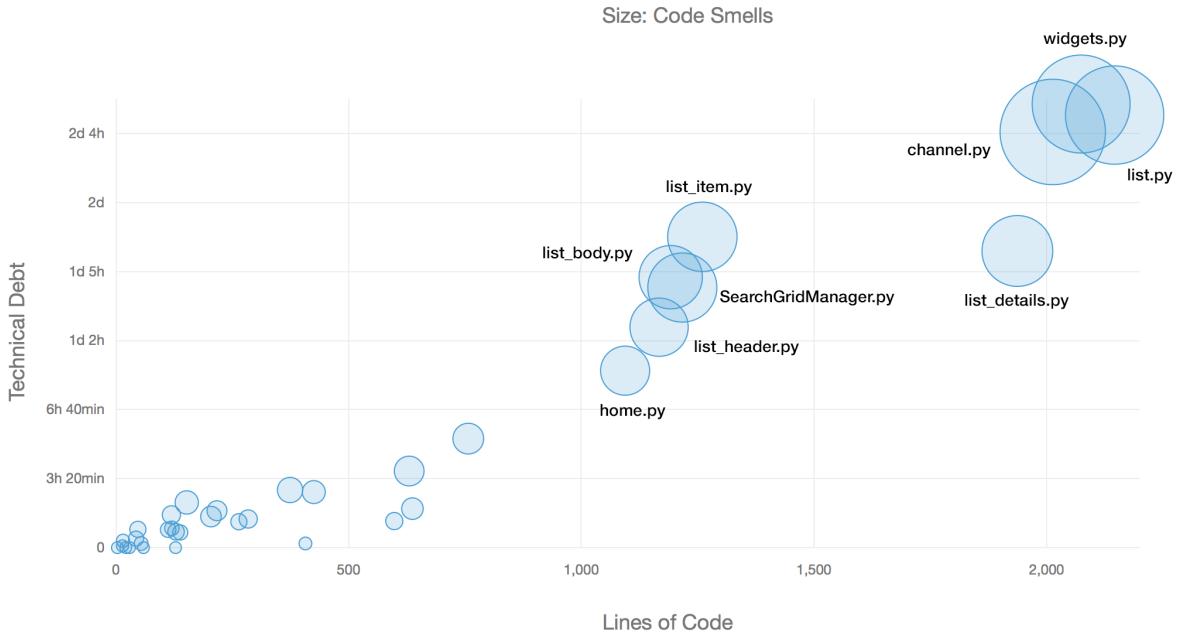


Figure 5.1: The amount of technical debt in the wxPython GUI reported by SonarQube.

5.2. Code debt

We now turn our attention to the core of Tribler and present the bubble chart associated with the core package in Figure 5.2. It is immediately evident that the *SQLiteCacheDBHandler.py* file urgently needs refactoring. This file contains various utility classes to perform common database operation such as retrieval of specific

[1] http://www.castsoftware.com
[2] https://sonarqube.com

channels and torrents. This file hosts the implementation of seven classes and our first effort consists of splitting this file into separate files where each file contains only one class definition.

However, we still identify many other files where technical debt is present in the form of code smells. We performed efforts to pay of this debt in the code base and will now summarize the most common identified code smells in the Tribler core package:

- The cyclomatic complexity of various methods is too high, indicating that the procedure contains many independent linear execution paths. This negatively impacts the testability of this method since more distinct tests are necessary to guarantee an adequate coverage of the method. During this thesis, we reduced the complexity of several methods by splitting them into separate parts.
- We identified various methods that could be static. Static methods are meant to be relevant to all the instances of a class rather than to any specific instance. The usage of static methods is beneficial for performance and readability reasons. SonarQube recommends to use static methods where possible and we changed as much methods to static as possible.
- Naming conventions were not following during the development process and this is most notable in the inconsistency between the usage of *CamelCase* practice and the usage of underscore notation. Since we wish to conform to the PEP8 styling guidelines[3], we should use the latter form. Part of efforts to pay off the identified code debt in the core includes work to rename of method, function, attribute and variable names to conform to the underscore notation. We should emphasize that some used libraries such as wxPython and PyQt are using the camelCase notation, leading to forced violation of this convention when overriding methods from this library.

The bubble chart of technical debt identified in the core after refactoring efforts described above is visible in Figure 5.2. We emphasize the difference in scale on the vertical axis here compared to Figure 5.2. The variance of the code debt has decreased significantly. Notice that the database handler definition files (that were originally located in the larger *SQLiteCacheDBHandler.py* file) still contains some technical debt. However, there are many methods in these files that are unused when the new user interface will be deployed and could be removed at that point. Table 5.2 shows the statistics after our refactoring efforts. While we did not solve all code smells, we solve the most prominent occurrences of code debt and contributed to a more useful, maintainable and consistent code base.

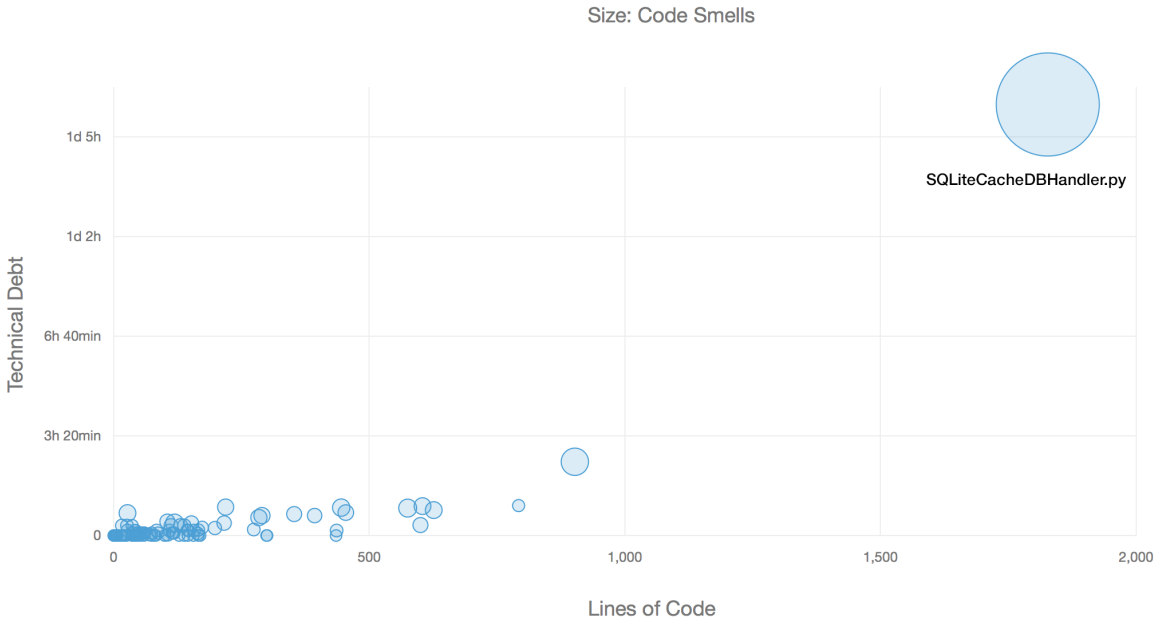


Figure 5.2: The amount of technical debt in the Tribler code reported by SonarQube before refactoring.

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

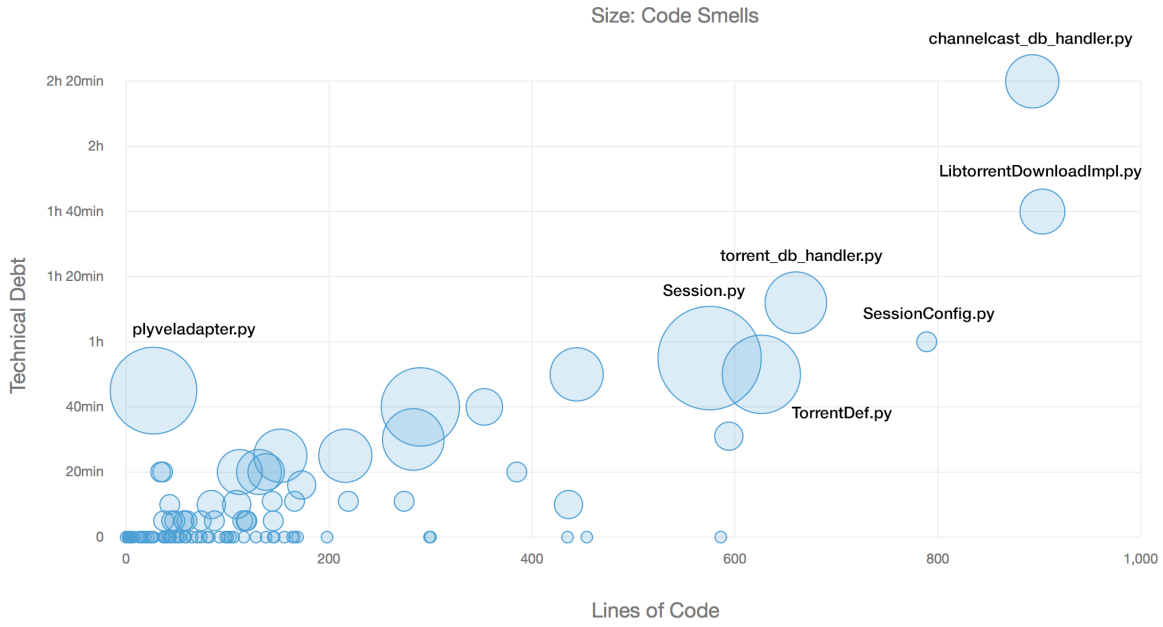


Figure 5.3: The amount of technical debt in the Tribler code reported by SonarQube after refactoring.

5.3. Testing debt

The most fundamental way to verify a correct functioning of software is by having an well-designed and stable test suite. A solid test suite leads a to high quality bar, thus introducing a mechanism to keep the amount of technical debt under control[19]. As pointed out in Chapter 2, the current test suite is plagued with unstable and non-functional tests. We will now focus on the performed work to strengthen and stabilize the test suite. A summary of the improvement of various metrics related to the test suite during this thesis is presented in Table 5.3. Notice that the number of unit tests has dramatically increased, while the average execution time per test and the total duration of the tests has decreased.

	November '15	**July '16**
Number of unit tests	80	676
Number of assertions	117	1205
Number of failed runs after 10 runs	2	0
TLC/PLC ratio	0.06	0.14
Total Linux test duration on Jenkins (sec)	448 (7 min. 28 sec.)	350 (5 min. 50 sec.)
Average execution time per test (sec)	18.90	0.85

Table 5.3: A summary of improvements to the test suite between November '15 and July '16.

5.3.1. Identifying code smells in the tests

As described in the work of van Deursen et al[53], 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 of the test suite of Tribler. This list is presented in Table 5.4 where each code smell is described and a solution is proposed.

Table 5.4 has been used as reference during the refactoring efforts of the test suite. We fixed most 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 in the tests and got rid of managing Tribler sessions as much as possible.

Code smell	Description	Solution
Dependencies on external resources	Various tests are using external resources outside the test suite, 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 Twisted after completion of a test.	Make sure that any delayed call in Twisted is correctly removed when the test is completed.
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 small unit in the system. Also implement a separate bundle of tests for the user interface.
Tests with a high execution time	There are some tests that are taking long to complete (sometimes over 30 seconds), negatively impacting productivity. This is an indication that the specific test has too much responsibilities.	Identify why the test takes long to complete and shorten the runtime i.e. by breaking the larger test into smaller parts.
Unclear assertions	Tests that consists of multiple assertion statements often do not annotate this assertion well with a clear and meaningful description	Add an annotation with the cause of the failure if an assertion fails so developers can determine the problem quicker.
Dependencies on a Tribler session	Some tests are starting a full Tribler session while only a small subset of the system is tested, leading to unpredictable circumstances.	Use mocking techniques to inject a mocked session or refactor the component so no session is required to test the component.
Resource writing to source code directories	Various tests are writing resources to source code directories. They might accidentally end up in the Version Control System (VCS) if developers are not noticing these files.	Temporary resources produced by tests should always be written to a temporary directory that is cleaned up after test execution.
Uncontrolled allocation of local network ports	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	Some tests are checking for 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 code comments	There are no code comments that are explaining the purpose and expected output of the test.	Tests should be annotated with comments to explain the purpose of the test together with the expected in- and output.
No directory structure in the tests package	There is no directory structure and a large amount of the tests are located inside the same directory.	Restructure the tests package and organise tests in different, logical named directories.

Table 5.4: 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*
Tribler core	71,2%	58,1%	81,2%	67,3%
REST API	-	-	99,4%	92,7%
wxPython GUI	65,8%	42,7%	-	-
Qt GUI	-	-	73,4%	50,4%

Table 5.5: The difference in code coverage between November '15 and July '16.

5.3.2. Improving Code Coverage

Code coverage is defined as the percentage of SLOC that is covered by at least one test. Our continuous integration (CI) environment offers tools to track the code coverage over time. After each 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 third-party libraries are included in the coverage report, such as the VLC bindings and *pymdht*, a library to fetch peers from the DHT. We are not responsible for the code coverage of these libraries so we excluded them from the report.

A summary of improvements of the code coverage metric during the span of this thesis are displayed in Table 5.5 where the SLOC and branch coverage is visible before and after this thesis work. Branch coverage is a metric that specifies how well conditional statements are covered and 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 dealing with code containing many nested conditional statements. The cyclomatic complexity as developed by McCabe in 1976[36] 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. The branch coverage is usually lower than the SLOC since it is either hard to cover a specific branch or the missing branch might be considered as not unimportant. For instance, this might happen when we have a branch that only logs an event.

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. Refactoring of the test suite has consequences on the code coverage in other locations in the code base. To elaborate, the smaller unit tests are not starting the old user interface anymore, leading to a lower coverage in that module.

Improving the coverage has been done by writing small unit tests where we use mocked objects to control the system we are testing. The increase in the amount of unit tests is displayed in Figure 5.4 where we annotated when this thesis has started. Using mocking is necessary since some components have many other dependencies that are hard to keep under control. Writing tests makes a developer more aware of the written code and it can be considered as a possibility to get familiar with an unknown code base. An additional advantage is that various bugs have been solved during the process of writing additional tests.

In Figure 2.5 the ratio between the number of code lines in the tests package and the amount of other code lines over time has been visualized. Together with the code coverage, this number can be a useful metric to developers. While one might argue that 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 possibly performing 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. In the case of Tribler when starting this work, the code coverage is reasonable but the TLC/PLC ratio is very low, indicating that most likely, the tests are not granular enough. This is in accordance with the discovered code smell that individual tests have too many responsibilities.

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[8] proposes an optional TLC/PLC

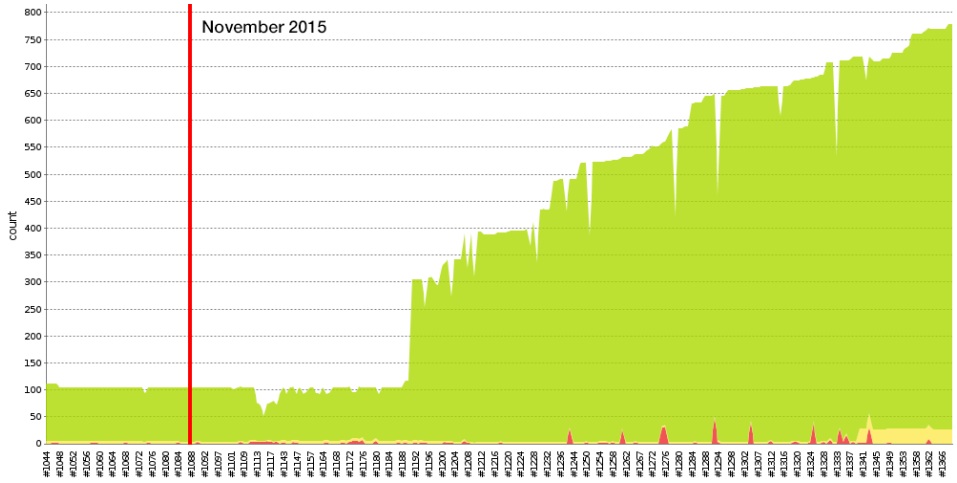


Figure 5.4: The number of tests over time until July 2016 (November 2015 is annotated).

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 of van Deursen et al.[53], a 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 is suitable for Tribler. Tribler differs from a commercial software engineering project in the sense that it is used for scientific research. When performing research, testing is considered a subordinate task and the main focus is gathering results that can be published. 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 an adequate TLC/PLC ratio for the Tribler project is between 1:2 and 1:4. With this ratio, we do not spent too much on writing tests while still maintaining a solid test base.

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 at least one test. While this check is not implemented by the author of this thesis, this is an effective way to keep the code coverage metric under control and to make developers more aware of the code coverage.

5.3.3. Testing the GUI

One of the issues identified in the tests package, was the lack of separation between tests that are testing the GUI and tests that are asserting core functionalities of Tribler. This is the main reasons that have led to big, individual tests in the old test suite. Since testing is an important aspect of this thesis work, constructing a solid test suite for the user interface code has been a prioritized task earlier in the development process.

User interface testing is a 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 and verification of the correct output of an action is often a non-trivial task. A common way of testing user interfaces is a Finite State Machine-based modelling where the user interface is modelled as a state machine that transitions when actions in the user interface are performed[20][16]. Another model to create a test script based on a genetic algorithms has been proposed by the work of Kasik et al[30]. While these models might lead to good results when dealing with large application, consisting of many pages and transitions, we think they are unnecessary to utilize when testing the Qt user interface at this point.

Testing the new Qt user interface uses the *QTest* framework. This framework provides various tools to perform non-blocking waits in the tests and to simulate mouse clicks and keyboard actions. A sample of a test written with the *QTest* framework is presented in Listing 5.1. This test has the following execution flow: after the interface is started, the test navigates to the home page, clicks on the *channels* button in the header and waits for items to be loaded. During the test execution, two screenshots are taken, 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 saved as a *jPEG* file and the name of the file is specified by the developer. In the sample presented 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 in our CI environment where the captured screenshots are archived and displayed in a grid. This allows developers to manually verify whether there are defects in the layout of the user interface. A part of the generated image gallery is displayed in Figure 5.5.

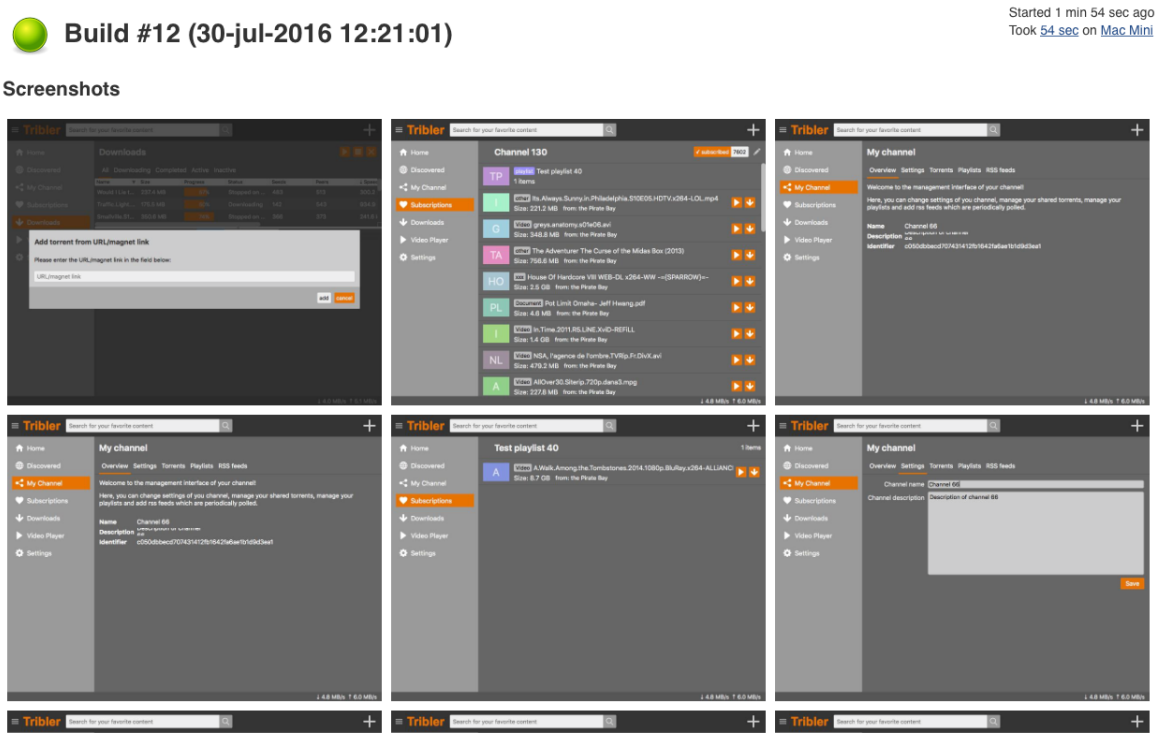


Figure 5.5: The generated image gallery after executing of the user interface tests, generated by Jenkins.

To avoid any dependency on core components of Tribler itself, we implemented a small piece of software that provides the same interface as the REST API implemented in Tribler. This 'fake' API is much simpler in nature and has a very simplistic in-memory data model. By utilizing this API, we are able to control API responses, significantly improving the predictability of the tests. The downside of this approach is that new endpoints have to be written twice, once in Tribler and once in this fake API.

A summary of several statistics related to these GUI tests is displayed in Table 5.6. We note that the average execution time per test is higher than the time presented in Table 5.3, however, during the tests we have various points where we have to wait for incoming data from the fake API provider.

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 identified code smells in Table 5.4 is the dependencies on external resources, leading to unstable and unpredictable tests. To elaborate, the test suite contains various tests where external torrent files are

Amount of unit tests	23
Total execution time on MacOS	1 min. 4 sec.
Average execution time per test	2.8 sec.

Table 5.6: A summary of statistics of the GUI tests.

fetched from the Internet, in particular, from the Ubuntu repository. While this repository guarantees a high availability, any downtime in this external resource leads to failing tests. The implemented solution for this 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.

A similar solution has been applied to solve the dependency on seeders in the libtorrent network by setting up a local session that seeds a torrent. A small number of tests makes assumptions on the availability of torrent pieces. This dependency makes tests fail if the machine that runs the test has an unstable internet connection. Again, this approach requires code to properly start and shut down the seeder session, thus increasing complexity of the test suite. However, the implementation is reusable to an extend that developers of tests can reuse the implemented solution with only a few lines of code.

Unfortunately, there are various external 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 ensure incoming search results. Due to time constraints, getting rid of this dependency is considered future work.

5.3.5. Instability of Tests

An unstable testing suite has a direct impact on the productivity of developers: when tests fails to reasons unrelated to the code that the developer contributed in a specific commit, developers have to execute the test suite again. One method to do this is by writing a comment on the Pull Request (PR) that says *retest this please* on GitHub. Every retest operation is "wasting" several minutes since developers have to wait for the completion of test execution before they have the necessary feedback about the stability of their PR. This is a structural problems that Tribler developers are experiencing since the utilization of continuous integration.

To further investigate this structural problem, we estimate the total time developers had to wait for retests by writing a small script that uses the GitHub API to analyse every opened PR and count the amount of retests required before the PR is merged. Before we present the results, we should note that we might miss some occurrences since it is possible to remove comments on GitHub. In addition, some retests might be related to failures in the continuous integration environment and are not caused by flaws in the test suite. In total, we counted 2.045 retests in 1.481 pull requests, on average, 1.38 retests for each merged PR. If we use an optimistic estimation where an execution of the full test suite takes six minutes in total, we spent around 204 hours retesting pull requests. We argue that we can stabilize the test suite in much less time so we need no retests anymore. To demonstrate that we are dealing with a structural problem here since 2013, the number of retests over time has been displayed in Figure 5.6.

Essentially, we are dealing here with a special kind of technical debt. Developers prudently made the decision to postpone fixing of the problems in the test suite by retesting the PR until the tests suite succeeds. The incentives to spent some time to fix test errors are not apparent and often, developers do not feel that they are responsible for failing tests since it might have been caused by code written by other developers. This makes it attractive to ignore test failures.

Well-designed tests should only fail if some new code is breaking existing functionality. If no changes are presents, the tests should always succeed, regardless of how many times it is executed. Reducing dependencies on external resources is not sufficient to guarantee this desired property. The structural problem of the tests is that the system is infected a great amount of race conditions. Race conditions can be hard to spot since they often occur in a very specific runtime setting, making the debugging process of these kind of errors frustrating. In fact, it is very easy to deliberately introduce a race conditions that is not noticed after the code is merged into the main branch. A complex architecture is of direct influence on this phenomena and leads

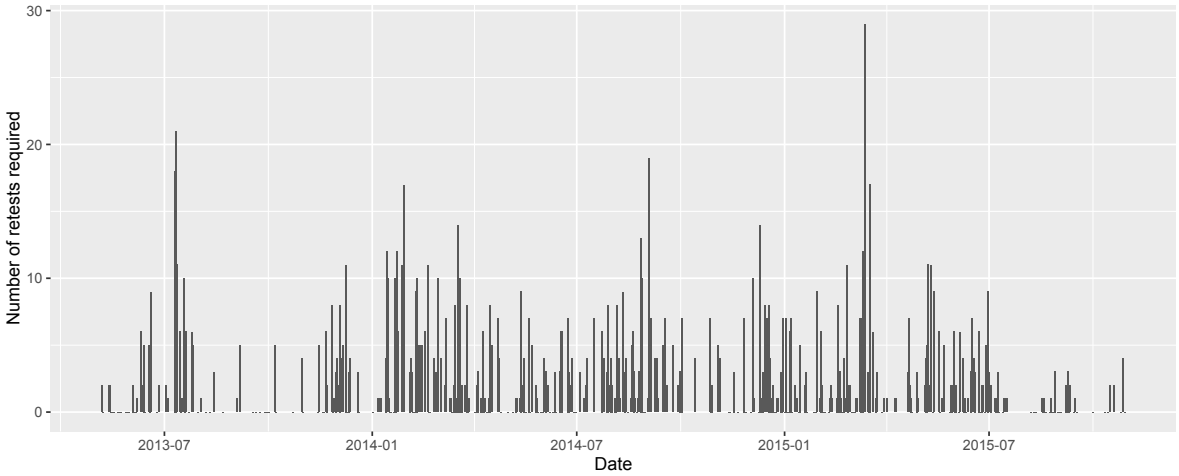


Figure 5.6: The number of retests required in PRs over time.

to more race conditions.

During this thesis, various 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 origin of the detected race conditions is addressed to delayed calls in Twisted. During the test execution, Tribler is started several times. If a developer leaves 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 Twisted is void of any delayed call is not straightforward: if one is not aware of scheduled calls in the system, the mistake is easily made.

5.4. Infrastructure debt

Tribler makes use of the popular CI platform Jenkins. Jenkins allows developers to define jobs which can be executed manually or when pushing a commit on the code base. This CI platform is responsible for running the tests, packaging Tribler and running research-oriented experiments on the DAS5 supercomputer.

We noticed that the test suite is only executed in a Linux environment. Beller et al[15] conducted research on CI adoption and usage and it turned out that for some languages, it might benefit to run tests in different environment. We strongly agree with this vision and since Tribler is shipped for multiple platforms, we think it is of uppermost importance to run the unit tests in different environment. An addition argument for this is the presence of some platform-specific workarounds in Tribler. To make sure that these statements are covered by the tests, we must run the test suite in a specific 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 set up should have a (small) positive influence on the code coverage metric.

The set up of the testing environments on Windows and MacOS is straightforward: new slave nodes to specify the Windows and MacOS test runners have been created. The tests on MacOS are executed on a Mac Mini, late 2014 model with 4GB of DDR3 memory and an Intel Core I5 1.4 GHz processor. In order to run the tests on Windows, two virtual machines using Proxmox[4], a server virtualization management platform, have been created, both with 32-bit and 64-bit environments. After the utilization of these additional machines, the tests are executed on four platforms: Linux, Windows 32-bit, Windows 64-bit and MacOS. So far, both the MacOS and Windows test executers have completed over 2.700 test executions. Each test runner generates a coverage reports and these reports are merged in the final analysis step in the build pipeline.

[4]https://www.proxmox.com

5.4.1. Future improvements

While this is certainly a step in the right direction, there are additional steps in the execution plan that could enhance the testing procedure. In Figure 5.7, we present what we think is the ideal test execution plan, together with the various stages in this pipeline. The dashed boxes are jobs in the pipeline that are not implemented yet. The pipeline starts by the execution of 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 and the pipeline aborts. This negative result is visible on GitHub.

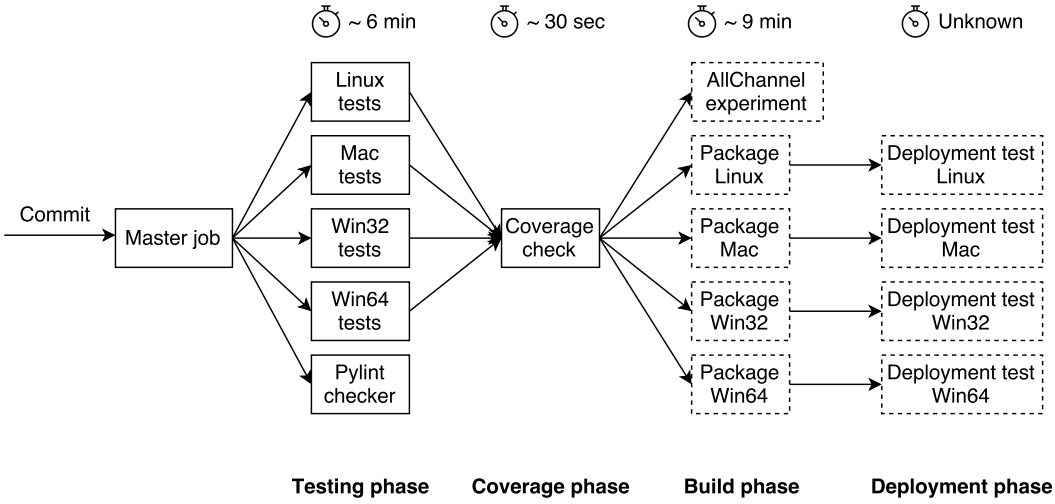


Figure 5.7: The desired test execution plan in our CI environment. Dashed boxes are not implemented yet.

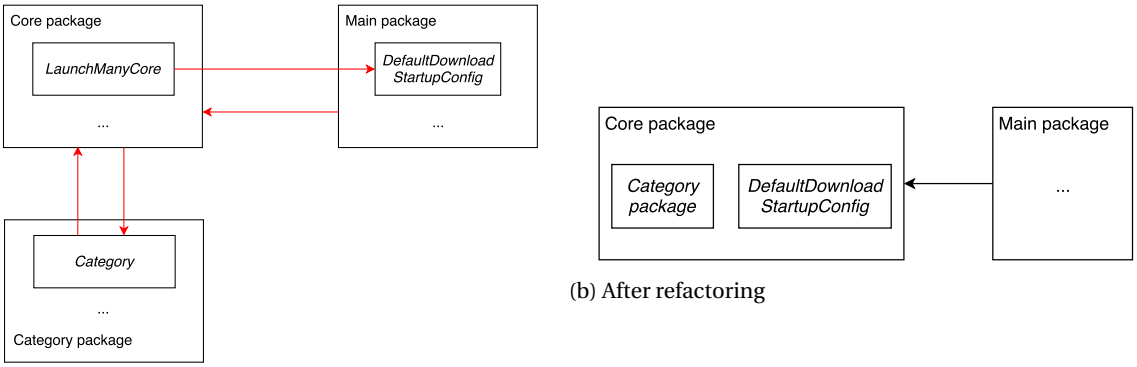
A static Pylint checker to check for code style violations has been available for a long time, however this only gave insight in the total amount of Pylint errors in the whole code base and did not stimulate to actually fix errors in the committed code of a developer. While not implemented by the author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

After the coverage phase has passed, the *AllChannel* experiment should be performed. This experiment is executed on the DAS5 supercomputer and starts 1.000 Tribler clients that are synchronizing torrent and channel information with each other. When the experiment is completed, various graphs are generated, providing developers insights in the performance of their modified code when Tribler runs in a more large-scale setting. For instance, the experiment can highlight introduced issues in the message synchronization between peers in the network.

In parallel with the *AllChannel* experiment, we should package Tribler for distribution to end-users. The purpose of this step is so we can execute deployment tests on various platforms to check whether Tribler works when distributed to end users. On Windows, an installer will be created that installs Tribler to the *Program Files*. On MacOS, we create a *DMG* file that contains an app bundle. On Linux, the required files are bundled in a *deb* archive. All these packaging and deployment testing jobs can be executed in parallel to shorten the time of the testing pipeline.

5.5. Architectural debt

Already indicated by Figure 2.4, Tribler is plagued with many dependencies that are leading to a highly coupled system where it is hard to reuse individual components. We aim for a low coupling to increase testability of packages. This Section will focus on identification and removal of undesired dependencies between packages.



(a) Before refactoring

(b) After refactoring

Figure 5.8: The dependencies between Tribler modules at the highest level.

5.5.1. GUI and core dependencies

As described in Chapter 2, the source code for the user interface and Tribler core code is interleaved to a large extent and there is no clear separation between these components. There are various instances where we identified code present in the GUI code base that should be moved to the core and vice versa. To realise a clear separation between *libtribler* and the user interface, we should make sure that we move code to the package where it belongs.

In the present code base, the *Core* package is dependent on the user interface which is undesired: we are unable to utilize the Tribler core without the GUI code being present, leading to higher coupling. The exact dependency is visible in Figure 5.8a and is caused by the *DefaultDownloadStartupConfig* class which is located in the *globals.py* file, part of the GUI package. This class is responsible for providing default configuration options when a download is being started, in case the user did not override default options like the destination of the downloaded file and the amount of anonymous hops used. Since the superclass of *DefaultDownloadStartupConfig*, *DownloadStartupConfig*, is already located in the Tribler core, it makes sense to move the *DefaultDownloadStartupConfig* class to the *DownloadConfig.py* file, which already contains the *DownloadStartupConfig* class. After we moved this class to the core and modified the references to point to the new location of the class, the core is completely independent of the user interface, displayed in Figure 5.8b.

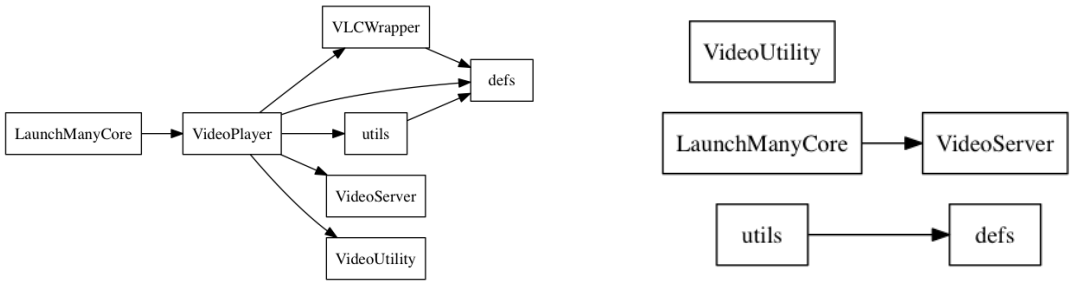
5.5.2. Category package

Referring to Figure 5.8, we note a cyclic import dependency between the *Core* and *Category* package. The *Category* package hosts the source code to facilitate the family filter. Obviously, the family filter is used by the Tribler core, however, the family filter also has dependencies on classes inside the Tribler core, causing the cyclic import.

In the architecture proposed in Figure 3.5, we specified the family filter as a component of libtribler. We think that the best solution to solve this dependency, is to move the *Category* package to the Core package so it's part of libtribler. This change is reflected in Figure 5.8b.

5.5.3. Video Player

We will now zoom in on the core package which contains some GUI-related code that should not be present in that package. The most obvious occurrence is attributed to management of the (embedded) video player in Tribler which is handled by the *VideoPlayer* class in the *Video* package. Figure 5.9a shows the import graph of the *Video* package before refactoring. The *VideoPlayer* class makes use of the VLC bindings for Python, however, in our design, the core does not need to have any dependency on VLC since managing the video player is a operation that should be performed on the user interface level. The *LaunchManyCore* class contains code to initialize all components available in Tribler, including the *VideoPlayer*. When initializing, this *VideoPlayer* creates a *VideoServer* that is responsible for the streaming capabilities of Tribler. The *VLCWrapper* class contains various utility methods to work with raw VLC data such as the time position within a video.



(a) Before refactoring

(b) After refactoring

Figure 5.9: The import graph of the *Video* package in the Tribler core before and after refactoring.

We performed refactoring work within this package and removed the *VideoServer* and *VLCWrapper* classes. The composition of the *Video* package after this operation is displayed in Figure 5.9b. We modified the code such that the *LaunchManyCore* class starts a video server instead of a video player. We point out that there are some classes that are unused now, such as *VideoUtility* and *utils*: these classes contains some helper methods to retrieve thumbnail images from a video file and is considered legacy code. Due to time constraints, we are unable to implement these features in the new user interface so for now, we keep these files as reference for future development.

5.6. Documentation debt

During the last ten years of development efforts on Tribler, the main focus of the project has been to deliver working code. The project has a severe lack of updated software artifacts, including documentation, code comments and architectural diagrams, leading to a huge amount of *documentation debt*. Some of the conducted research was documented on the Tribler wiki[5], however, this wiki contains many outdated pages and is not used or maintained anymore. After the migration of the project to GitHub, this platform was favoured for storing documentation over continued usage of the Tribler wiki archive.

Currently, there are various 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 repository. The ideal situation is to have one single location for all generated software artifacts during the process. Many Python projects are using *readthedocs*[6], 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 website can be generated from all the available documentation. Sphinx also provides tools for translation of documentation in other languages.

During this thesis, all available documentation of Tribler has been rewritten in RST format in conjunction with *Sphinx*. Moreover, the available documentation has been expanded with several guides, in particular, guides that help new developers to set up an 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 Tribler development.

The REST API in particular has been well documented. Since external developers should 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 in the source code. This documentation is parsed by the *autosummary* tool that is executed each time the documentation is built: it navigates through the API code base, extracts all doc strings and generates separate sections for each annotated method. The doc string can be attributed with *RST* syntax. This feature decreases chances that developers accidentally forget to write or update artifacts since the code and documentation is present in the same file instead of being spread across different distinct files.

[5]https://www.tribler.org/TitleIndex/
[6]https://readthedocs.org

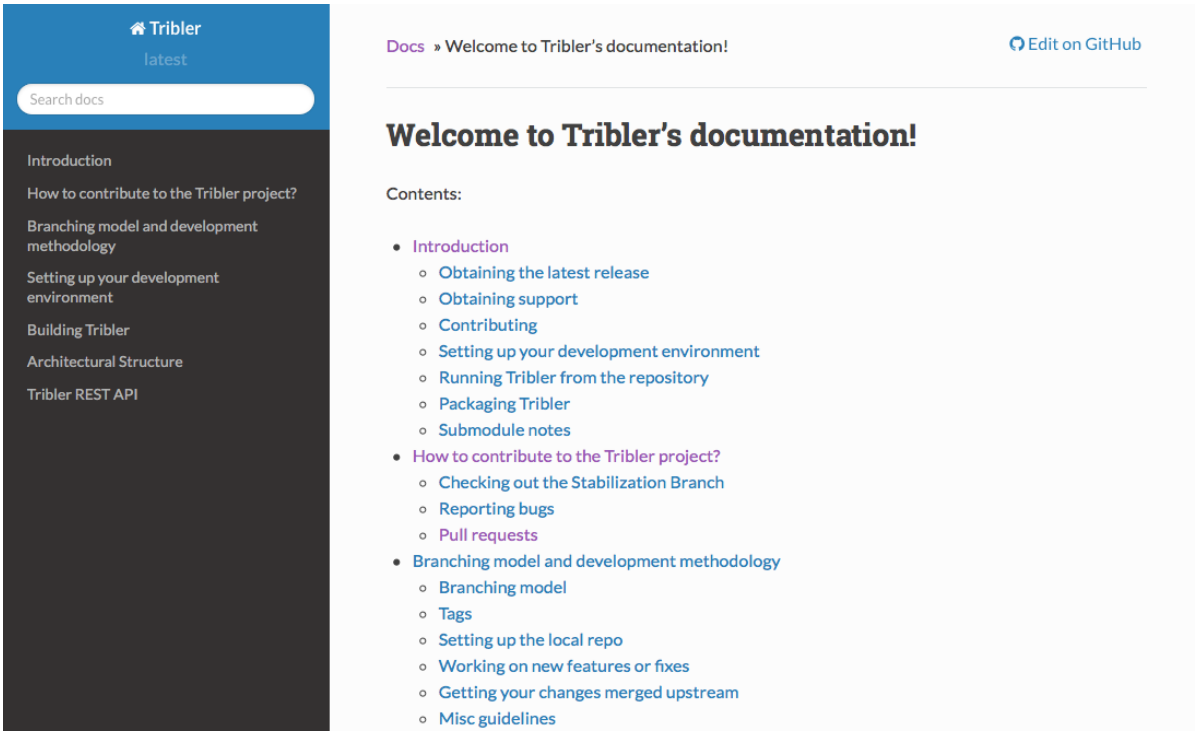


Figure 5.10: The new documentation base of Tribler, as available on the *readthedocs* website.

5.7. Preventing Technical Debt

Prevention is the best medicine and we must think about a mechanism that prevents technical debt in future development stages. Developers have never been aware of the long-term consequences of the Tribler architecture and their work. To stop the deterioration of the system, we must raise awareness of technical debt and the term needs to play a more profound role when making long-term development decisions. To realise this, we present the following list of implemented ways to raise technical debt awareness:

- Introducing mandatory code reviews of new PRs is an effective way of ensuring that problems in the code are detected as early as possible[9] but additionally, it helps developers to learn from their mistakes and to raise the quality bar of contributed code. The new policy introduced during this thesis requires each PR of developers to be reviewed by at least two other Tribler developers. This policy also helps developers to get more aware of ongoing work from other developers.

- Continuous integration and automated testing is an excellent opportunity to maintain a higher level of code quality and to catch bugs during the development process before end users are reporting them. The work as described in this Chapter, has matured the Jenkins and testing environment so it can be used reliably by the next generation of Tribler developers.

- Our CI environment already used static analysis tools to report violations in the source code which is an effective way to make developers aware of their introduced violations[38]. However, when starting this thesis, these analysis tools have been implemented as separate jobs and were not executed on every pull request. This has been changed so developers receive fast feedback when they push a new commit to GitHub. The implemented checks executed for every PR are visible in Figure 5.11. Besides the reports of the test execution on multiple platforms and the code violation reporter, the code coverage report fails if developers added or modified lines that are not covered by a test. This tool will definitely contributes towards an increase of the code coverage metric in the long run.

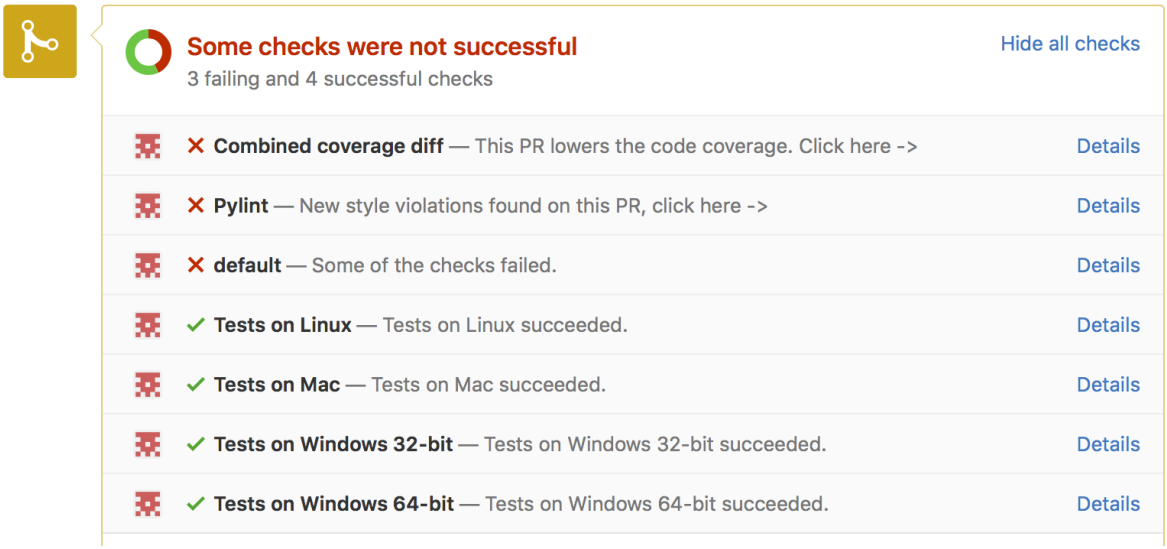


Figure 5.11: The implemented checks in Jenkins, executed on every new commit in a pull request.

6

Performance Evaluation of libtribler

Technical debt has a negative impact on product quality in terms of defects and other structural quality issues[51]: it is significantly harder to fix defects in a complex, unstructured system and more dangerous in a sense that one might introduce additional bugs when trying to solve one. Boosting performance is often achieved by minor or major refactoring efforts of system components to utilize another underlying model or structure. Modifications of the system is more involved when the system as whole is suffering from huge amounts of technical debt.

Now that we got rid of most of the technical debt identified in the Tribler core, the next step towards a stable, future-proof libtribler involves research efforts on the usability and performance of various components. We wish to quantify the performance of operation in libtribler to get an idea about the usability of the core in general. For various components in the core, we have no performance baseline to help us to make statements about usability. 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 show that the performance of the system did not degrade to an unacceptable extent due to our refactoring efforts. On the other hand, we use the performance measurements to create a baseline and to identify possible failures or issues that we will classify as future work.

6.1. Environment specifications

The experiments performed in this Chapter are executed on a *virtual private server (VPS)*. We wish to stay as close as possible to the specifications of a machine that an actual user could be using. A summary of the specifications of the machine used for most of the experiments described in this Chapter, is given in Table 6.1.

CPU	Intel Xeon CPU E5-2450 (2.50GHz, 4 cores)
Memory	8GB 1000MHz
Hard disk	50GB
Operating system	Ubuntu 15.10

Table 6.1: The specifications of the machine used for most of the experiments.

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 set up can be used to get more insights in the performance of Tribler from a user's perspective.

If not stated otherwise, the default Tribler configuration file values 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 the *BarterCast* community, are initialized. Dispersy, the HTTP API and the video server are enabled during the experiments. All experiments are executed without running the *wxPython* or Qt GUI.

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

Some of the experiments are built using a scenario file. In such a scenario file each line specifies a specific command of a peer at a specific point in time during the experiment. Our framework to run the experiments, Gumby, contains code to read scenario files, interprets the commands to be executed and to schedule these commands in Twisted. 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^2. Various Dispersy experiments are already using the scenario file framework, mainly in our *AllChannel* experiment that runs on the DAS5 supercomputer. Before we executed the experiments in this Chapter, we first extended the usability of the scenario files to run and manage a Tribler session and we improved the framework with the addition of various commands to support the operations that are executed in the performed experiments in this Chapter. An overview of all implemented commands can be found in Appendix A. The flexibility of these scenario files gives developers a robust framework to use when conducting performance analysis, benchmarking and other kinds of scientific research with Tribler.

6.2. Profiling Tribler on low-end devices

The implementation of a RESTful API gives developers a possibility to run and control Tribler from remote devices. For instance, one can run Tribler on a low-end, cheap devices such as a Raspberry Pi and use it to accumulate reputation in the Multichain by enabling the credit mining mechanism. 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[45][23]. 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 regular desktop or supercomputer.

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

Some preliminary exploration of the performance on the Raspberry Pi using the REST API has us suspected that the device 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 execution time of methods in Tribler. This profiler has been integrated in the *twistd* plugin and can be started together with Tribler by passing an option. 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*[3], a *callgrind* file visualizer. We start this experiment 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.

[2]https://www.r-project.org
[3]https://sourceforge.net/projects/qcachegrindwin/

The file created by the Yappi profiler provides a detailed overview of the execution time of methods in Tribler 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 could be used to locate the respective function in the Tribler code base.

If we analyse the breakdown, we notice 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 Tribler run time 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 a system.

To verify whether the system load when running Tribler decreases when we disable cryptographic verification of incoming messages, we measure the CPU usage of two different 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 in Dispersy. The CPU utilization over time of the two runs are displayed in Figure 6.2: on the horizontal axis, we show the time into the experiment and on the vertical axis, we display the CPU utilization in percentage. We emphasize that Tribler is limited to run on a single CPU core.

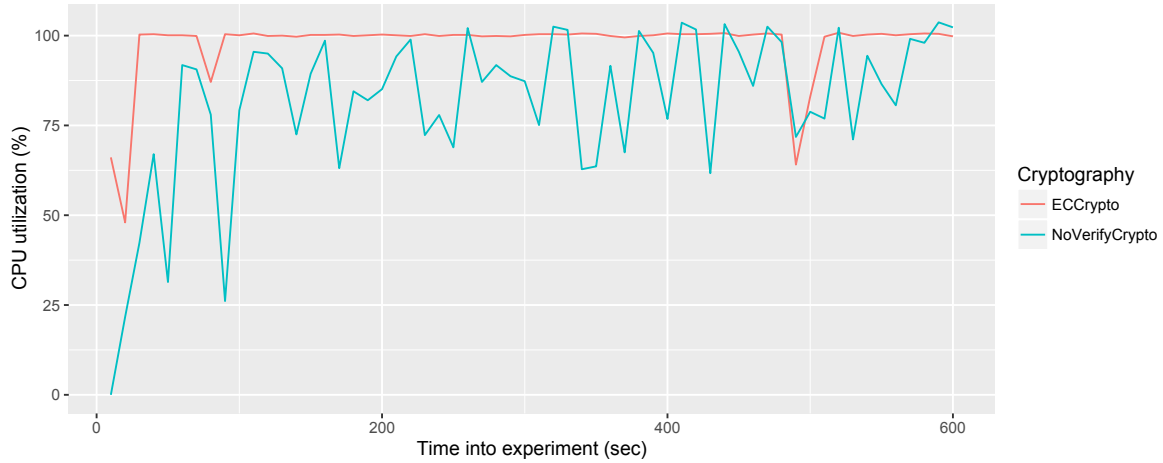


Figure 6.2: The CPU utilization of one core on a Raspberry Pi device when running Tribler with and without cryptographic verification of incoming Dispersy messages.

In Figure 6.2, some occurrences can be identified 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 cryptographic verification of incoming messages, the CPU usage is often 100%, leading to a non-responsive system. When we disable message verification, we observe a somewhat lower CPU usage but overall, this utilization is still relatively high. Unfortunately, disabling incoming message verification is not enough to always guarantee a more usable and responsive system.

To detect other performance bottlenecks, we sort the report that has been generated by the Yappi profiler on the *self* column to get insights in methods that are taking a long time to complete. This is visible in Figure 6.3. An interesting observation 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 resolve to a true value. 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, analysis and optimization of this method is considered future work and has been documented in GitHub issue 505[4].

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.

To summarize, we demonstrated how adequate usage the Yappi profiler can lead to the detection of performance bottlenecks present in Tribler and Dispersy. Integration of the profiler in the twistd plugin makes it convenient for developers to run and analyse Tribler sessions under different circumstances and on a broad range of devices.

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 in the user interface. For this reason, we wish to make the API serve requests as fast as possible. The purpose of this section is to assess the performance of the API with a particular focus on latency of request response times. 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 and the responsiveness of Tribler.

We make use of Apache JMeter[5] 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 within a fixed time interval. The performed (GET) request will be targeted to a specific endpoint in the REST 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 consists of a JSON-encoded dictionary of all channels that Tribler has discovered so far. The returned response by this request 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 query is performed to fetch all channels that are stored in the persistent SQLite database.

We perform multiple experiments with different time intervals between requests made and a fixed total amount of 500 requests per experiment. First, we conduct the experiment with one request every second and we expect that the system should 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 frequencies have been determined empirically 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. During the experiment, we are using a pre-filled database with around 100.000 discovered torrents, 1.200 discovered channels and a subscription to 20 channels. A summary of the experimental results are visible in Table 6.2 where we present various request statistics.

[4] https://github.com/Tribler/dispersy/issues/505
[5] http://jmeter.apache.org

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 6.2: A summary of the experimental results when measuring the performance of the RESTful API.

If we focus on the average request time (second column), the most interesting observation is that it appears that requests are served 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 this to be the other way around: when the frequency of requests is increased, the average request time is expected to increase since Tribler has more to process. The observed result is most likely explained by caching mechanisms data performed by the underlying database engine or Twisted.

The standard deviation of the request times in Table 6.2 (third column) is for all experiments rather high compared to the average request time. We suspect that this can be explained by the fact that Tribler is performing many different operations besides serving API requests. In particular, we think that Twisted is busy with processing other calls that have been scheduled earlier, causing the API calls to be processed later. To verify this, we ran the experiment again where we disable Dispersy, the component responsible for many calls in the reactor (as concluded in Section 6.2). We perform five requests per second and 500 requests in total for this experiment. The observed results are illustrated in Figure 6.4 where we display the time into the experiment on the horizontal axis in seconds and the request response time in milliseconds on the vertical axis.

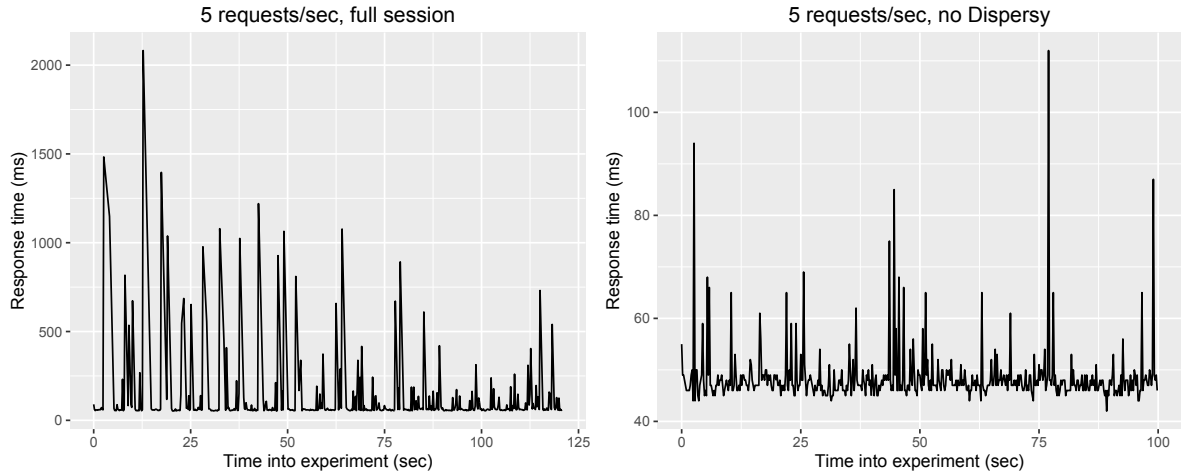


Figure 6.4: The response times of API requests, in a Tribler session both with Dispersy enabled and disabled.

In the left plot, the response times of the performed requests with a regular Tribler session is displayed (corresponding to the 5 requests/sec row in Table 6.2) whereas in the right plot, we display the response times of the run with a disabled 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 in Figure 6.4 is 48 milliseconds, significantly lower than the average of the response times when Dispersy is enabled, namely 123 milliseconds. While both plots are showing a spiky pattern, the standard deviation of the right plot is 5.73 milliseconds and the standard deviation in the left plot is 210.23 milliseconds. We conclude that the variation in response times is much lower in the right plot and that DIspersy is producing a huge amount of work, introducing considerable amounts of latency when performing API requests.

We identified a key issue here: the latency of methods to be processed in Twisted is high, causing the processing of incoming requests to be delayed. This is not only a situation that occurs in the REST API: the same

situation holds for Dispersy and the tunnel community where possibly many incoming connections have to be processed and served. A step in the right direction is to make sure that there are no big blocking calls scheduled in Twisted that are considerable amount of time to complete. When a method with a long execution time is executed, Tribler is unable to process other events during that period, leading to a less responsive system. Ongoing work is focussed on making the disk operations in Tribler non-blocking. This should reduce the latency of event processing and improve the responsiveness of the system in general.

Table 6.2 provides us with another interesting observation, namely that it appears that the bandwidth is reducing as the number of requests per second increases. This becomes more obvious if we plot the theoretical maximum bandwidth together with the observed bandwidth during the experiments, see Figure 6.5. In this Figure, we presented both the obtained bandwidth by running a regular Tribler session and one where Dispersy has been disabled. We assume that each request contains 613KB (627.712 bytes) of data in the response body. The theoretical maximum obtainable bandwidth is determined 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 initialize the HTTP connection in Tribler which we do not consider in our simple model. Figure 6.5 clearly shows the impact of a running Dispersy on the bandwidth. Whereas we almost obtain the theoretical bandwidth when we disable Dispersy, the gap between the theoretical maximum and observed bandwidths 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.

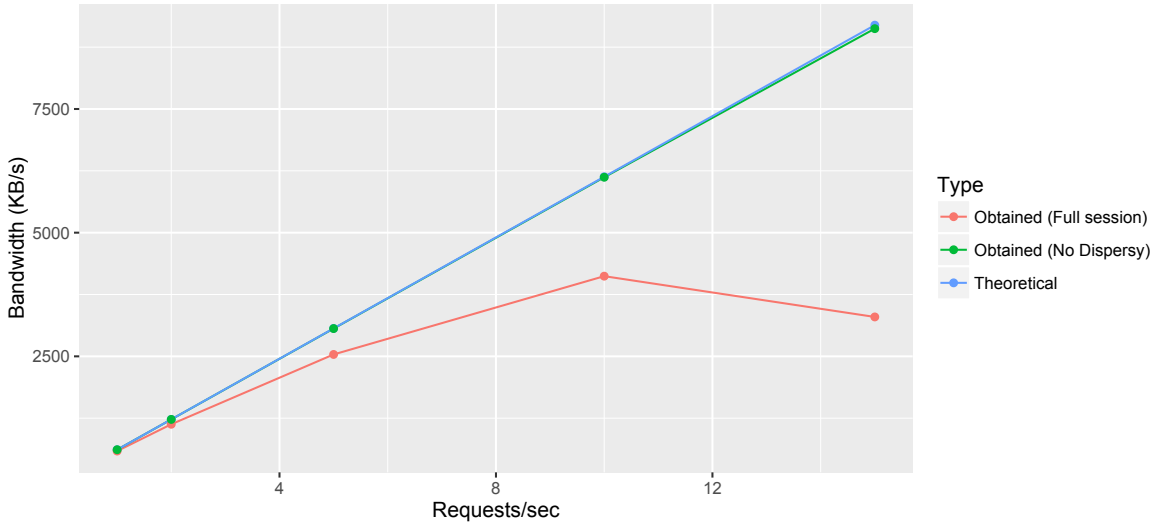


Figure 6.5: The theoretical maximum bandwidth compared to the observed bandwidth in the experiments (using a full Tribler session and disabled Dispersy).

We conclude this experiment with the argumentation that we can use the API response times as a benchmarking tool to measure the responsiveness of the Tribler core. Using the Apache JMeter application, we can easily build a stress test and verify whether performance has increased or decreased after a specific set of modifications. Implementation of a performance regression framework is considered future work.

6.4. Start-up experience

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

- The Tribler state directory where runtime data are stored, is created and initialized with necessary files such as the SQLite database, the Dispersy member key pair and various configuration files.
- A connection to the persistent SQLite database is opened and initialized.
- Dispersy is initialized and the communities that are enabled in the configuration file are loaded.

- Various Tribler components are created, including the video streaming server, the RESTful API, the remote torrent handler, responsible for fetching torrent information from other peers and the *leveldb* store, a key-value storage for torrent meta info.

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, we start Tribler 50 times in a row. The experiments are performed multiple times where in one experiment, the software is started for the first time, with no prior existing state directory. When starting Tribler with no prior existing state directory, a new one is created and the required files are initialized. In the other runs, a pre-filled database containing just over 100.000 torrents is used. This database is built by running Tribler idle for several hours, after subscribing to some popular channels to synchronize and discover as much torrents as possible. In both scenarios, there are no active downloads. A timer is started when the *start* method of the *Session* object is called and stopped when the notification that Tribler has started, is observed, allowing us to determine the total start-up time to a granularity of milliseconds. During the span of this thesis, there have been various changes to the start-up procedure of Tribler where code has been modified, removed and added. Since we would like to guarantee that our modifications do not significantly decrease the start-up speed and we make a comparison between the Tribler code in November '15 and July '16. The results are presented in Figure 6.6, where for each commit we compare, we present an empirical cumulative distribution function (ECDF) with the boot time in seconds on the horizontal axis and within each plot, the distribution of start-up times from a clean and pre-filled state directory.

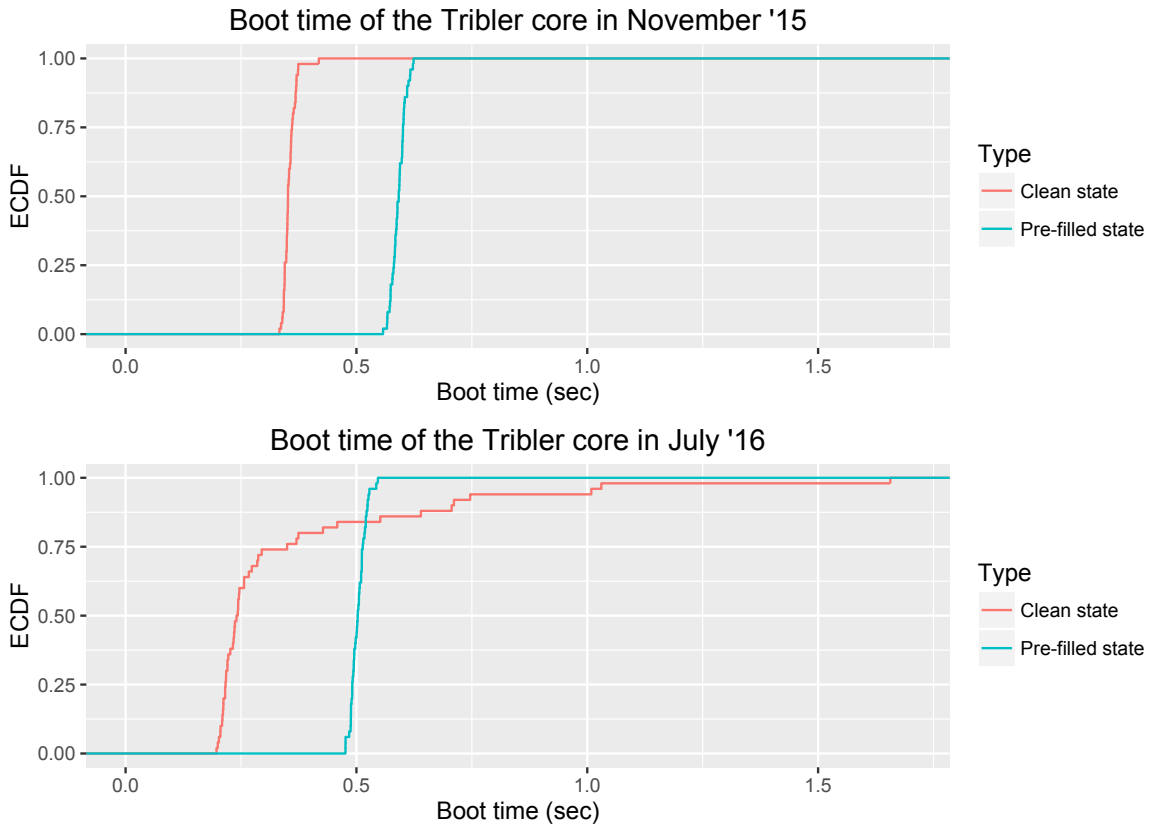


Figure 6.6: The start-up time of Tribler from a clean and pre-filled state using the code base in November '15 and July '16.

In both plots, It is clear that size of the Tribler database has impact on the time for Tribler to completely start. However, this impact is relatively minor since the system still starts 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 small 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 GUI starts Tribler

and shows a loading screen after the interface has started. However, the difference is that users are able to already perform some actions before Tribler has started, such as the browsing of discovered content.

Whereas the boot times of the experiments performed with the November '15 code are very constant, we notice a larger variation in the runs with the code base from July '16, indicating that there is a component that has a high variation in initialization time during the start-up procedure. Further analysis learns us that this variation can be addressed to Dispersy, possibly caused by the initialization 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 discover content they like, a remote keyword search has been implemented, allowing users to 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, however, for the experiments in this Section, we will focus on remote search for torrents since the amount of channels is rather small compared to the number of torrents available in the network.

Several experiments to verify the speed of a remote torrent search are discussed in this Section. A list of 100 search terms that have a high chance of triggering search results is used and each query is executed when there are at least twenty connected peers available in the *Search* community (this condition is checked every second). The time-out period of the remote search is 60 seconds, indicating that incoming search results after this period are not regarded. This experiment is focussed on two performance statistics: the time interval until the first remote torrent search result comes in and the turnaround time of the search request, meaning the interval until the last search response arrives. We should note that users performing a remote search might see results earlier since a lookup query in the local database is performed in parallel (the performance of local search is discussed in Section 6.6). The results of our experiment are visible in Figure 6.7 where we created two ECDF plots with the distributions of time until the first response and time until the last response. On the horizontal axis, the measured time interval in seconds is visible.

Overall, the remote torrent search as implemented in Tribler is fast and performs reasonable. On average, there are 61 incoming search results for each performed query where the first torrent result takes on average 0.26 seconds to arrive. As we see in Figure 6.7, over 90% of the first search results are available within a second. During our experiment, we always have the first incoming torrent result within 3.5 seconds. The plot displayed on the right 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.

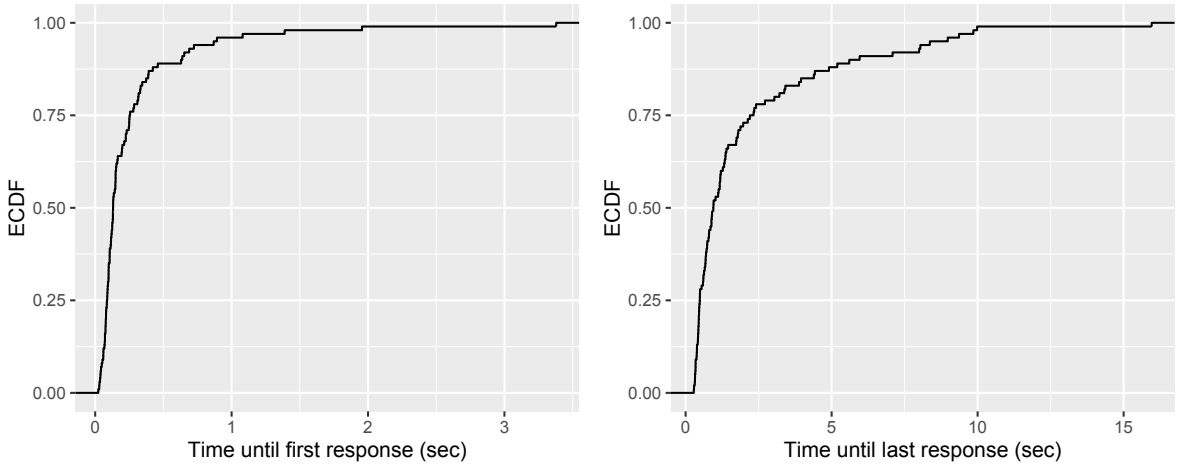


Figure 6.7: The performance of remote content search, expressed in the time until the first response and time until last response.

The same experiment has been performed in 2009 by Nitin et al. where 332 remote search queries have been performed. Their results are also presented in an ECDF in 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 input/output mechanism, causing messages to be exchanged at a faster rate. The observed average time until first response in 2009 is 0.81 seconds whereas the observed average time in our experiments is 0.26 seconds, more than three times as fast.

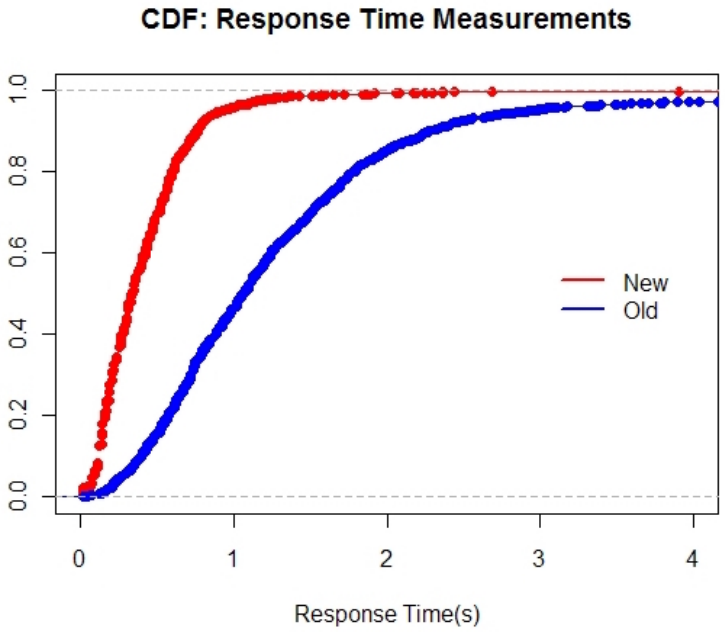


Figure 6.8: The performance of remote content search, performed by Nitin et al. in 2009. The new remote search had an improved input/output mechanism, causing messages to be exchanged faster.

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 local content search, which is considered more trivial than the remote search counterpart due to the lack of network communication. In particular, our goal is to quantify the performance gain or loss when switching to the new relevance ranking algorithm that uses a newer search engine, as described in Chapter 4.4.

The set-up of this 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 retrieved from the database.. In the code used in November '15, this post-processing step involves determining the associated channels that are containing a specific torrent result. This experiment is performed for the old method that uses the *Full Text Search 3 (FTS3)* engine and the new procedure that uses the more recent *Full Text Search 4 (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 of the experiments with the old and new local search logic are visible in Figure 6.9, presented in two ECDF plots with on the horizontal axis, the time of either the database query time (the red line) and the total time for the processing of results, including the query time (the blue/green line).

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 plots have moved closer to each other which means that the total time of post-processing of tor-

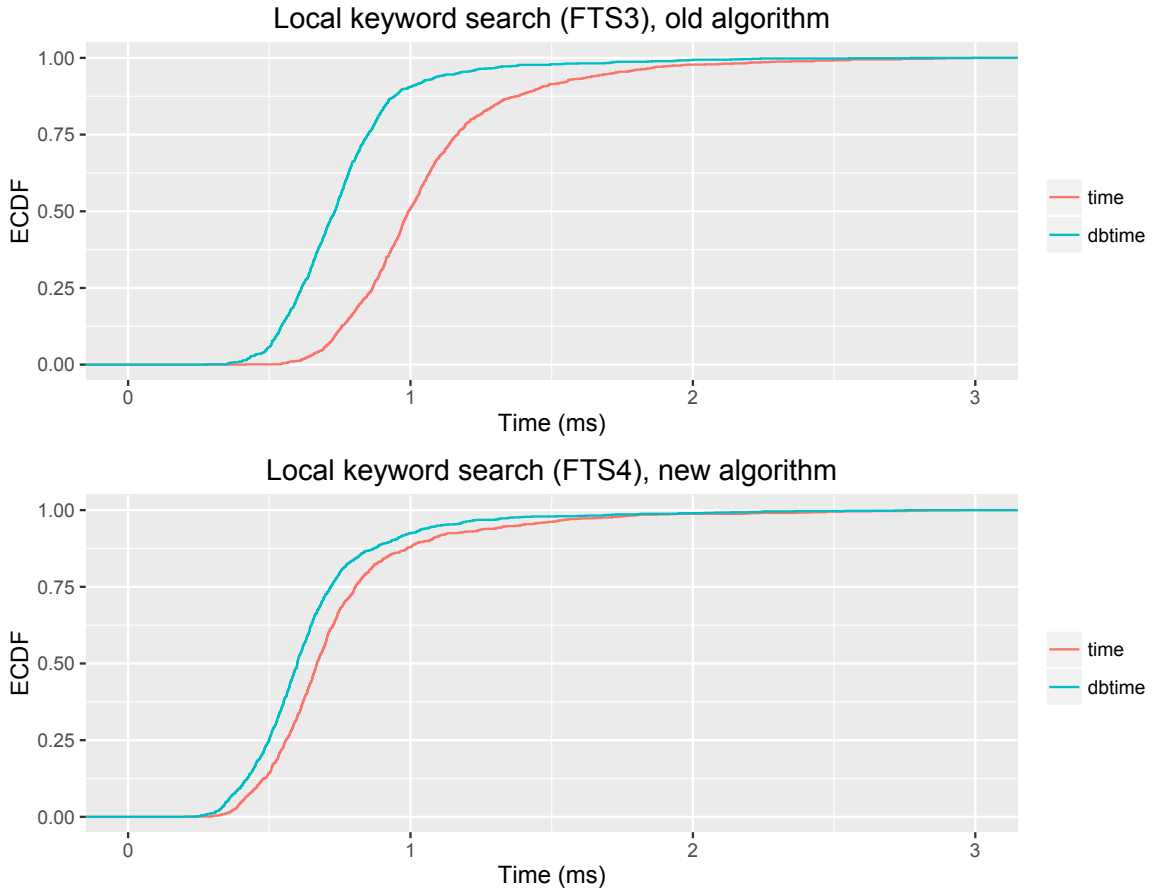


Figure 6.9: A comparison of the performance of local keyword searches between the old local search mechanism utilizing the FTS3 and FTS4 engine.

rent results has decreased. 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 very 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 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 a 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.

6.7. Video streaming

The embedded video player in Tribler allows users to watch a video that is being downloaded and the working is explained in more detail in Chapter 3.4.3. Video playback has been available since Tribler 4.0 and is implemented using the VLC library. One distinguishable feature is support for seeking so the user can jump to a specified time offset in the video. Video downloads have a special *video on demand (VOD)* mode which means that the libtorrent piece picking mechanism uses a linear policy mode. In this mode, pieces are downloaded in a sequential order. When the user seeks to a position in the video, the prioritization of the pieces is

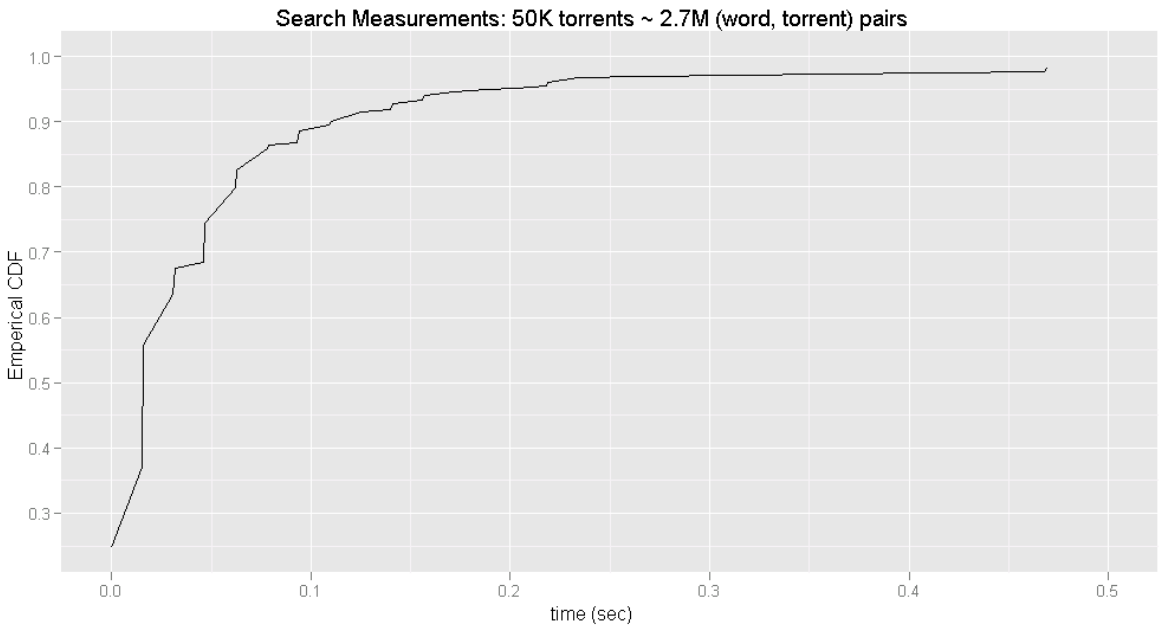


Figure 6.10: The performance of a local database query as verified by Nitin et al. in 2009.

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-compatible client using a HTTP stream. When Tribler starts, a video server is started if enabled in the configuration file. 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, possibly saving a huge amount of bandwidth. If some requested pieces are not available, the video server will wait until these bytes are downloaded before returning these bytes in the response.

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*[6] movie will be downloaded. The movie file has a size of 885.6 MB and has a duration of 9 minutes and 56 seconds. We will perform various HTTP range requests using the *curl* command line tool[7], 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 6.3 where we specified the first requested byte and the time until the request has been fulfilled and the response is received.

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 various anomalies 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. We found out that this is intended behaviour of the code since VLC needs the information embedded in the file header first. This file header provides information about the file type, file duration and encoding used. There are some video formats where this kind of information is present at the end of the video file.

conclusie??

[6] https://peach.blender.org
[7] https://curl.haxx.se

First byte	Time until request done (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 6.3: Performance of the video server when requesting bytes at different offsets of the video being downloaded.

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 messaging mechanism. When a user starts Tribler for the first time, there is no discovered content yet. We will verify the discovery speed of content after a first fresh 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 fifteen times. The results are visible in Figure 6.11 where we created an ECDF with a distribution summary of the discovery times of torrents (green line) and channels (orange line).

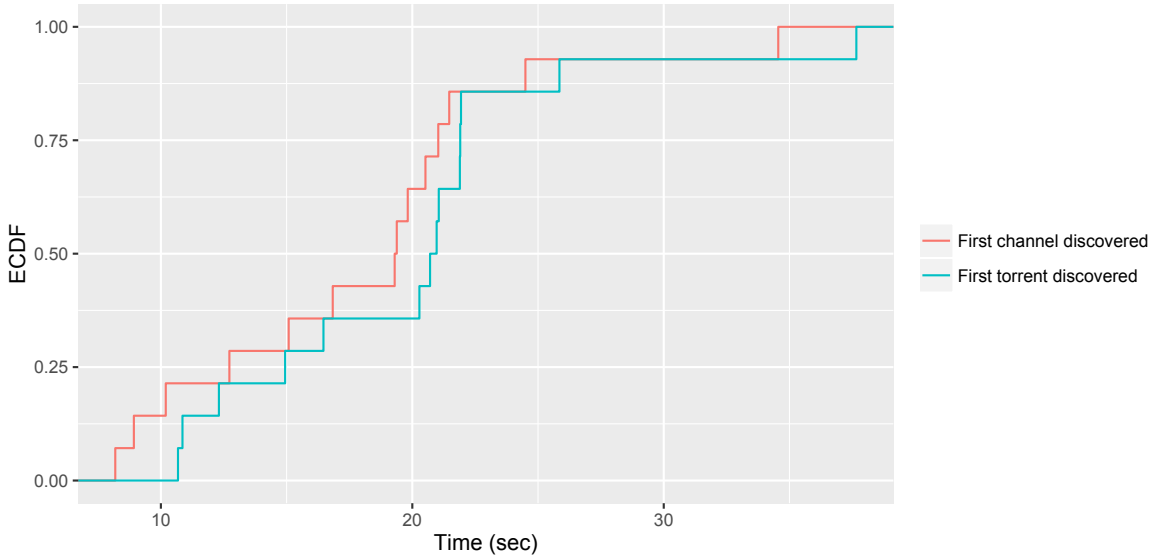


Figure 6.11: The discovery time of the first channel and torrent after starting first starting Tribler.

The delay of discovering the first channel is reasonably: this happens on average 18 seconds after start-up. In all runs, we have our first channel discovered within 35 seconds after Tribler starts. 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 after a while.

In the old user interface, users were presented with a blank screen with no feedback about content that is being discovered in the background. 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 is dismissed when there are five discovered channels after which the page with an overview of discovered

channels is presented to the user.

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 a channel (in the old user interface, this is referenced to as marking a channel as favourite). When Tribler discovers a new channel, users are able to browse 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* 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 (having the most subscribers) are determined. To get these channels, we have used 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 time interval between subscription to the channel and discovery of the first additional torrent in this channel. 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.

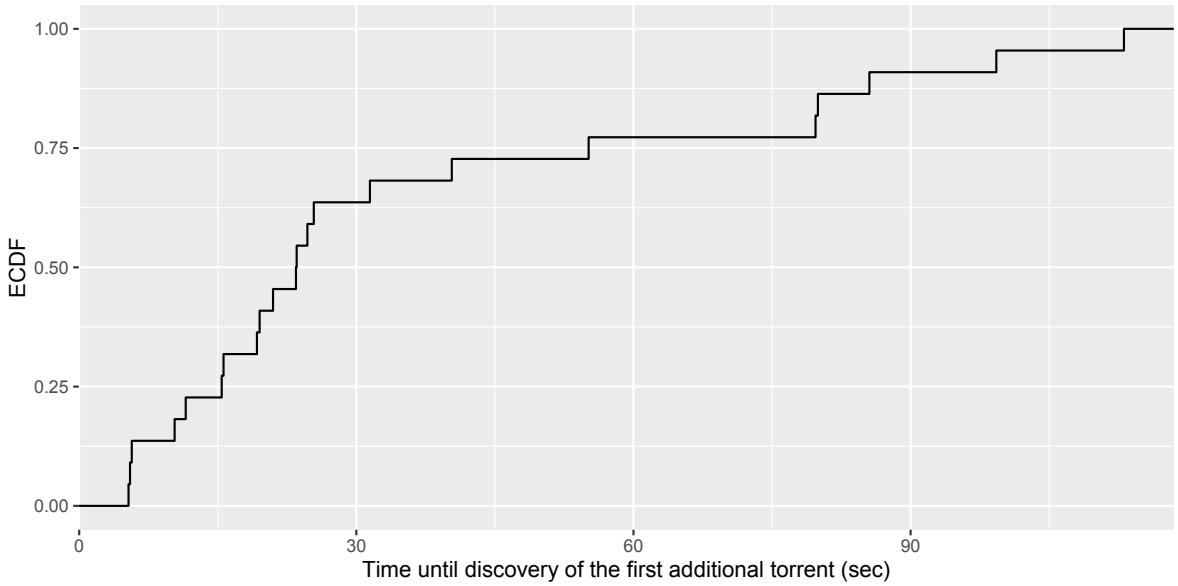


Figure 6.12: An ECDF of discovery times of the first additional torrent after subscribing to a popular channel.

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.

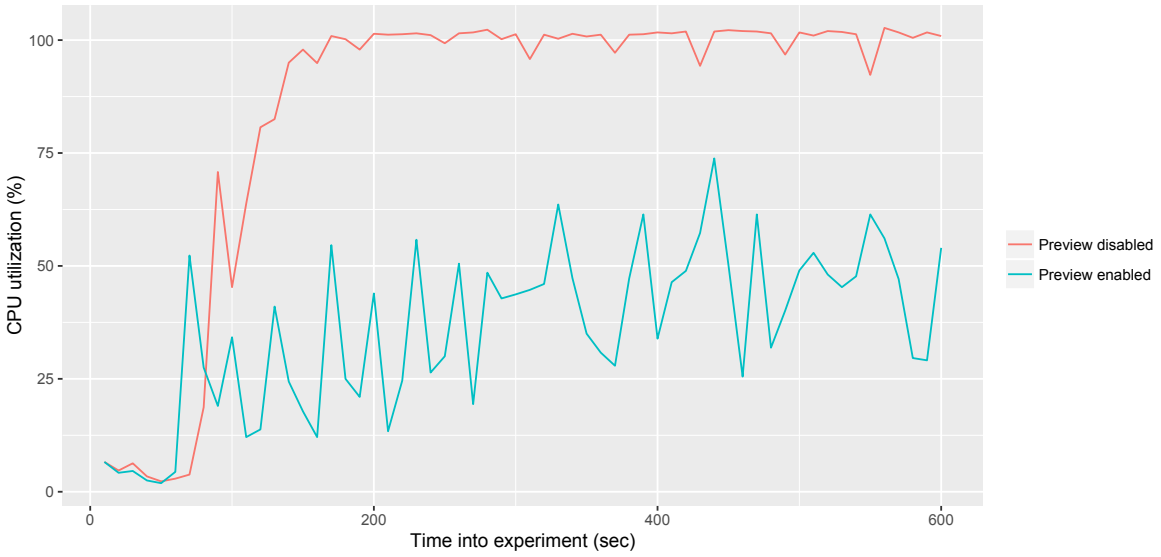


Figure 6.13: The CPU utilization of one core during a period of ten minutes with channel preview enabled and disabled.

Whereas the CPU usage of the normal run is around 45% on average, the CPU is quickly rising 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 available trackers and information about pieces. This meta info might 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. Trivial File Transfer Protocol Handler

When users are performing a remote torrent search, the first three incoming results are pre-fetched in the old user interface which means that the meta info of these torrents are fetched automatically. An incoming search result might contain information about remote peers (candidates) that have 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)*[48] which is a simplified version of the more sophisticated *File Transfer Protocol (FTP)*[41], 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 is located in the core package of 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. For every incoming result, we schedule a remote torrent lookup. After eleven minutes, we stop Tribler and gather the statistics of the TFTP sessions. The observed results are visible in Table 6.4.

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

Table 6.4: 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. First, some of the requests timed out, possibly due to the fact that various remote peers are not connectible. A solution for this kind of failure would be a robust NAT puncturing method. The second reason is that the remote peer might 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, a separate persistent database. 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 successful TFTP requests, presented in the ECDF in Figure **??**. Here, we notice the weird distribution of the turnaround times: we would expect that the total request times to fetch meta info using TFTP 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 eleven minutes. The most reasonable 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 6.4 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.

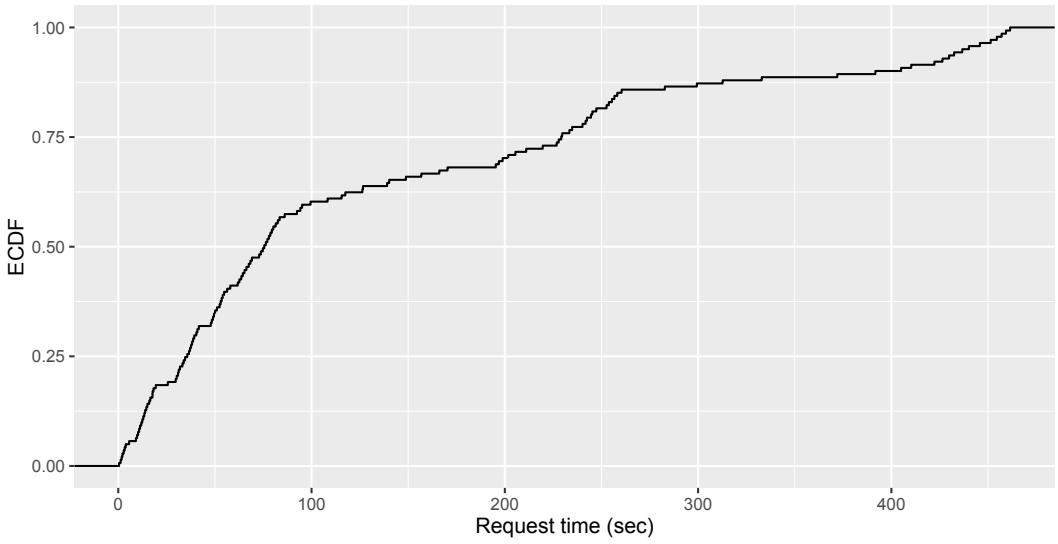


Figure 6.14: An ECDF of the performance of the torrent meta info download mechanism using TFTP in Tribler.

6.10.2. Distributed Hash Table

Another source of torrent meta info is the DHT. In the DHT we can lookup meta info of a torrent, identified by an infohash by invoking the *download_torrentfile* in the *Session* object. In this Section, we will perform an experiment to get insights in the availability of torrent files and the performance of lookup operations in the DHT. This experiment is relevant to the 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, including names and sizes of the files the torrent contains. This meta info should be available as soon as possible.

In the old wxPython user interface, the torrent file is fetched when the user single clicks on a torrent in a 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 there is a higher probability that the user might be interested in them. For this experiment, a popular channel with over 5.000 torrents is considered and a subset of 1.000 torrent infohashes in this channel is determined. We start Tribler from a clean state and 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 the ECDF depicted in Figure 6.15.

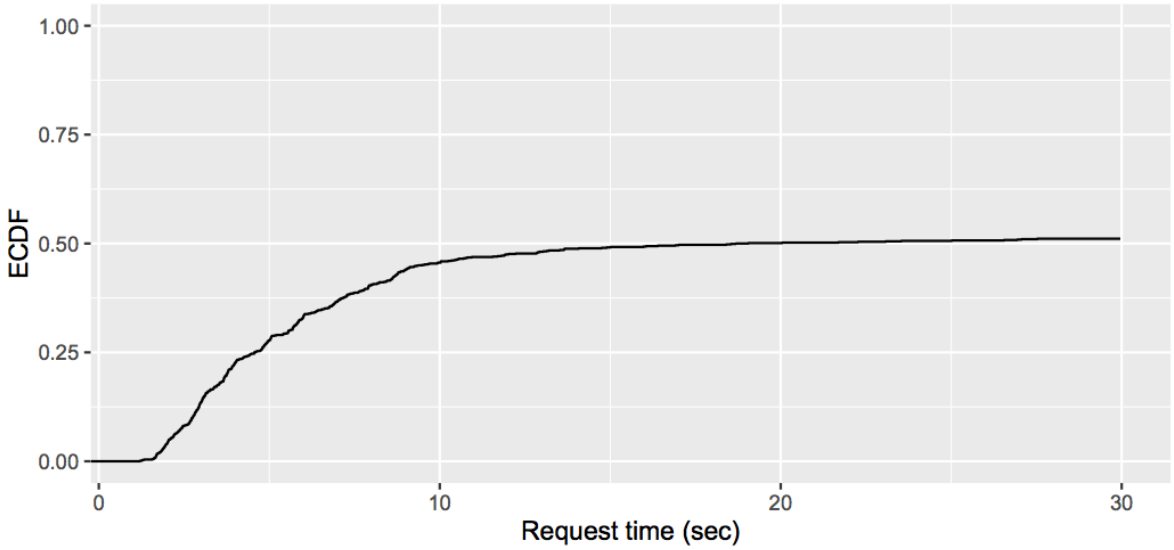


Figure 6.15: Test.

We immediately notice that the failure rate of DHT lookups is quite high: a little under 50% of the lookup operations are timing out and never succeed. 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 exposed in the DHT). The amount of failures might be even higher in a less popular channel since the content in these channels are probable to be 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 meta info lookups, 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

Testing Tribler at a large scale (concept)

In the previous Chapter, we performed many small performance measurements and quantified the usability and performance of various common performed operations in Tribler. While these experiments gave us insights in the performance of Tribler, we didn't chain together multiple operations to investigate a pipeline.

One of the main selling points of Tribler is the built-in anonymous overlay which provides anonymous downloading and seeding capabilities. In this Chapter, we will test a full pipeline of Tribler on a large scale, where we start Tribler, perform a remote keyword search, select a download and start this download anonymously. We will first discuss the setup of the experiment in more detail, after which we turn our focus to the results we observed during the experiment.

7.1. Setup of the experiment

The order of operations as performed in this experiment is visible in Figure 7.1. The experiment itself is executed on the DAS5 supercomputer where in every run, we allocate ten nodes and run one Tribler instance on every node.

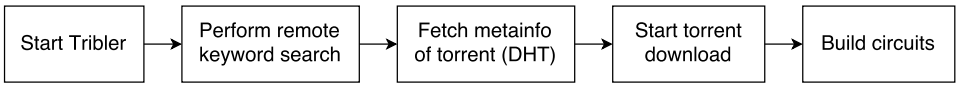


Figure 7.1: The setup of the experiment as described in this Chapter.

The experiment starts when Tribler is booted and we start with no pre-filled state directory. After Tribler has been started, we wait until we are connected to a sufficient amount of peers (30) in the *SearchCommunity* before we perform the remote search. In the experiment described in Section 6.5, we waited until we are connected to at least 20 peers. We increased this number since we are running ten other Tribler nodes inside the same network that might connect with each other. When these Tribler nodes connect to each other, they provide no remote search results to each other since there is likely to be no discovered content in the database yet. The check to see whether we are connected to a sufficient amount of peers is performed every second.

Once we have enough connections, we perform a remote torrent keyword search. For this purpose, we created a list of 1.000 popular keywords that have been constructed as follows: we analysed the database with just over 100.000 torrents, determined all keywords in the database, together with the frequency of each keyword and calculated a list of the keywords with the highest frequency in the database. In every run, we uniformly pick a random keyword from this list and perform a remote keyword search with the selected query.

We wait 30 seconds for incoming search results and we store the time of the first and the last incoming remote search results. If we have no search results within 30 seconds, we abort the experiment. We save all incoming torrent results and after 30 seconds, we pick five random, non-explicit torrent results and query a metainfo lookup in the DHT. We are using a timeout period of 60 seconds for the DHT lookup operation: if we did not

receive any response from one of the scheduled DHT lookup, we abort the experiment.

As soon as the first incoming metainfo is received, we start the download of this torrent, where we enable anonymous downloading with one hop and hidden seeding. After three minutes, we abort the experiment and clean the downloaded data. We keep track of the time until the circuits are build and we receive the first incoming bytes. Additionally, we keep track of the total number of downloaded bytes after these three minutes.

There are various failures that could lead to an interruption of the experiment, which we will summarize below:

- If we are not connected to at least 30 peers in the *SearchCommunity* after Tribler has started, we abort the experiment.
- If we do not receive any remote torrent result within 30 seconds, we abort the experiment.
- If we do not receive a response from any scheduled DHT lookup, we abort the experiment.

We also have various failures that influences our result but are not failing our experiment. An example of these failures is when we fail to build circuits within three minutes after starting the download. In this situation, we are still able to finish the experiment, however, our final results are influenced since we were not able to download any byte.

7.2. Observed results

ven

Todo

8

Conclusions

Technical debt is a recurring problem in almost all large software engineering projects. 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 careful analysis, we concluded that the code base related to the user interface suffers from architectural impurity and an unmaintainable amount of technical debt of all kinds.

To better understand the root causes of the technical debt, we discussed the architectural evolution of Tribler over its lifespan, concluding that Tribler benefits from a new architecture that meets three requirements in place to reduce the amount of technical debt in the future: *simplicity*, *flexibility* and *engineered for performance*. We started working towards this architecture by creating a new user interface, built using the mature and well-maintained Qt framework. Next, we created an extensive REST API that could be used to control Tribler by external developers. This provides migration possibilities, eventually leading to removal of the code base related to the old user interface.

While this solves the technical debt to a large extent, the core of Tribler is still suffering. During this thesis, we identified and solved different kinds of debt. Code debt has been identified with the SonarQube application and with some invasive refactoring efforts in the core, we reduced the amount of technical debt to just over one day. Testing debt has been handled by refactoring efforts in the testing framework mostly by splitting the tests into smaller parts while fixing various testing code smells. By updating and extending our CI environment with multi-platform unit tests and the improvement of Tribler installers, we reduced the amount of infrastructure debt. Architectural debt has been reduced by breaking dependencies in the Tribler core and by the implementation of the new user interface. Finally, we have set-up a new robust documentation structure, solving documentation debt.

Since technical debt has a negative impact on product quality in terms of defects and other structural quality issues, we identified where performance might have been lost due to the many architectural changes. We assessed the usability of various components found in libtribler and argued that our refactoring efforts did not degrade the performance of components. uitbreiden

Prevention is the best medicine. We should learn from the mistakes made during the past years. 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 is under control and can gradually be improved over time.

While this work is a step in the right direction to guarantee a robust platform for research in the area of decentralized networks, there is still much work to do. The performance of various components such as the video player and remote torrent handler can be improved to increase usability. The implementation of the trusted overlay located in the lower parts of the architecture in Figure 3.5.

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] Toyota unintended acceleration and the big bowl of “spaghetti” code. http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-\T1\textquotedblleftspaghetti\T1\textquotedblright-code. Accessed: 2016-08-14.

[2] Progress report group c2, client tribler. http://kayapo.tribler.org/trac/raw-attachment/wiki/userResearch/Tribler5.0_usability_test_C2.pdf,. Accessed: 2016-08-12.

[3] Progress report group c4, client tribler. http://kayapo.tribler.org/trac/raw-attachment/wiki/userResearch/Tribler5.0_usability_test_C4.pdf,. Accessed: 2016-08-12.

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

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

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

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

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

[9] Preventing technical debt. https://18f.gsa.gov/2015/10/22/preventing-technical-debt/, 2015. Accessed: 2016-08-09.

[10] Github issue 1066 - addressing the various security improvements. https://github.com/Tribler/tribler/issues/1066, 2016. Accessed: 2016-08-06.

[11] Open hub - tribler project summary. https://www.openhub.net/p/tribler, 2016. Accessed: 2016-07-29.

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

[13] Vlc media player. http://www.videolan.org/vlc/, 2016. Accessed: 2016-08-08.

[14] A Bakker, R Petrocco, and V Grishchenko. Peer-to-peer streaming peer protocol (ppspp). Technical report, 2015.

[15] 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.

[16] Fevzi Belli. Finite state testing and analysis of graphical user interfaces. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 34–43. IEEE, 2001.

[17] 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.

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

[19] Sumit Chhetri. How we use unit testing to tackle technical debt. github.io, 2016.

[20] James M Clarke. Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference. IEEE Press*. Citeseer, 1998.

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

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

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

[24] Davide Falessi and Andreas Reichel. Towards an open-source tool for measuring and visualizing the interest of technical debt. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 1–8. IEEE, 2015.

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

[26] Brian Foote and Joseph Yoder. Big ball of mud. *Pattern languages of program design*, 4:654–692, 1997.

[27] 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.

[28] Yuepu Guo, Carolyn Seaman, Rebeka Gomes, Antonio Cavalcanti, Graziela Tonin, Fabio QB Da Silva, Andre LM Santos, and Clauirton Siebra. Tracking technical debt—an exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE, 2011.

[29] 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.

[30] David J Kasik and Harry G George. Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 244–251. ACM, 1996.

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

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

[33] Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi, and Apostolos Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, pages 119–128. ACM, 2014.

[34] Fredrik Lundh. An introduction to tkinter. *URL: www. pyhonware. com/library/tkinter/introduction/index. htm*, 1999.

[35] 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.

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

[37] Michel Meulpolder, Johan A Pouwelse, Dick HJ Epema, and Henk J Sips. Bartercast: A practical approach to prevent lazy freeriding in p2p networks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[38] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[39] Robert Nord. Early software vulnerability detection with technical debt. `https://insights.sei.cmu.edu/sei_blog/2016/08/early-software-vulnerability-detection-with-technical-debt.html`, 2016.

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

[41] Jon Postel and Joyce Reynolds. Rfc 959: File transfer protocol. `https://www.ietf.org/rfc/rfc959.txt`, 1985.

[42] 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.

[43] Noel Rappin and Robin Dunn. wxpython in action. 2006.

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

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

[46] Carolyn Seaman and Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82 (25-46):44, 2011.

[47] Hugo Solis. *Kivy Cookbook*. Packt Publishing Ltd, 2015.

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

[49] Mark Summerfield. *Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming*. Pearson Education, 2007.

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

[51] Edith Tom, AybüKe Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[52] Arie Van Deursen and Leon Moonen. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002.

[53] 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.

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