# Attack-resilient media using phone-to-phone networking

P.W.G. Brussee


TU Delft

# Attack-resilient media using phone-to-phone networking

by

P.W.G. Brussee

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday November 30, 2016 at 10:30 AM.

| | | |
|---|---|---|
| Student number: | 1308025 | |
| Project duration: | December 1, 2015 – November 30, 2016 | |
| Thesis committee: | Associate prof. dr. J.A. Pouwelse, | TU Delft, supervisor |
| | Prof. dr. C. Witteveen, | TU Delft |
| | Assistant prof. dr. C.C.S. Liem, | TU Delft |

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

TU Delft

# Contents

# 1 Introduction

Modern media and news distribution is shifting from traditional media to social media. Modern media production and information distribution is shifting from traditional outlets. Also media consumption is shifting from traditional outlets to digital and mobile devices. From con-sumer to pro-sumer. News is not bound to expert opinion or an editorial news desk. The opinion of the individual can be broad-casted without the need for a professional studio and equipment. No office is required anymore to link a mobile eye-witness to a mass audience.

Given this context mobile devices are increasingly important and smart-phone in particular. A smart-phone has the unique property of being a ubiquitous device that is highly mobile and extremely connectible. Most smart-phones even have one or more cameras to produce multi-media content that can be shared immediately from the device. Finally the entire world has a smart-phone. Even or especially areas without tradition infrastructure they are ubiquitous.

In crisis situations, like natural disaster or unrest, the smart-phone becomes particularly important, exactly for the earlier mentioned properties, as under such conditions utilizing centralized infrastructure is undesired (censorship) or physically impossible. Decentralized can work in these situations. Censorship and large scale monitoring is difficult in decentralized networks.

Tribler is a fully decentralized video-on-demand platform. [5, 8, 10] It is autonomous, attack-resilient and self-organizing. [1, 7] It uses network overlays called communities to offer features like keyword search and managing contributions to channels for discover-ability of content. It offers privacy through layered encrypted tunnels similar to the TOR network.[2, 3, 11]

So far Tribler only supports desktop and server versions of Linux, Mac and Windows. The necessity of moving to mobile devices calls for Tribler functionality to be enabled to run on these resource limited devices. In this thesis the first prototype is presented that has all Tribler functionality fully enabled on mobile devices. The prototype is build for Android OS since Android dominates the smart-phone market.

Sharing opinion and discussion is enabled on social media. A global dialog is possible through social media like Twitter, Instagram and Facebook. People receive local and global news perceived relevant to their group on their social media feed. Young people in particular are shifting to social media as their number one source. [4] As shown in this age distribution graph, figure 1.1. Not only is social media more and more becoming a major distribution channel for news, it also starts delivering input for news and story creation. This way social media has become both the source and outlet for investigative journalism.

A smart-phone has the unique property of being a ubiquitous device that is highly mobile and extremely connectible. Figure 1.2 shows that world-wide 1,4 billion smart-phones were sold to end-users last year. And this number keeps rising. Even or especially areas without traditional infrastructure they are ubiquitous.

The capabilities and versatility of smart-phones enable it to be used for production and consumption of news and social media. The users themselves are turning from con-sumers into pro-sumers. With regards to production most smart-phones have one or more cameras to record multi-media content that can be shared immediately from the device. Eye-witnesses often have smart-phones at hand to immediately record an event with and post it on social media. No news desk or professional equipment is necessary to relay news directly from eye-witnesses to the masses anymore. People with camera phones can reach millions of people with multi-media in a very short timespan, becoming social journalists. The ease of reaching a global audience by an individual with a smart-phone diminishes the role of an expert curator handling incoming information.

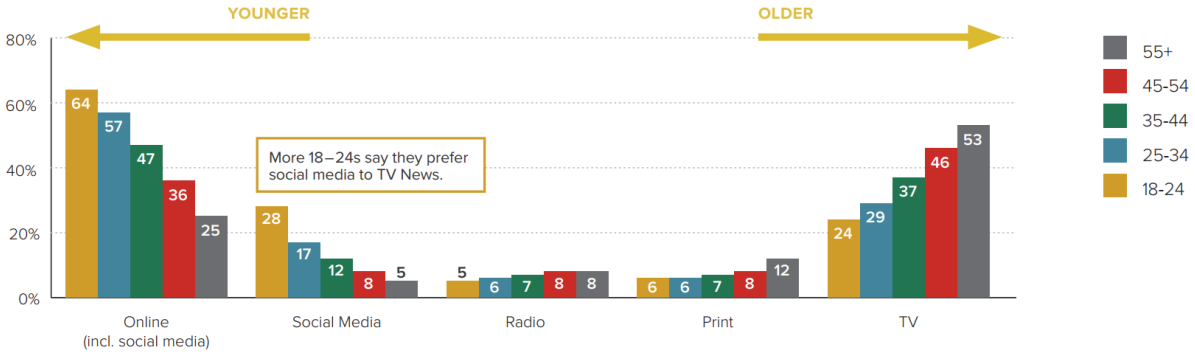


Figure 1.1: Main news sources split by age [4]

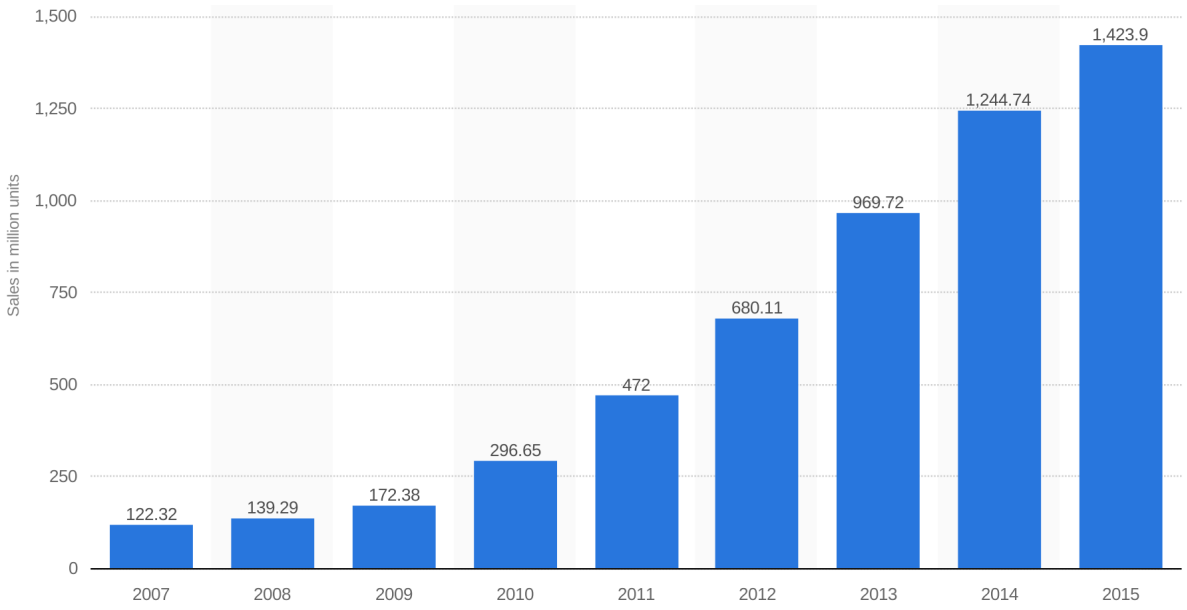


Figure 1.2: Number of smart-phones sold to end users worldwide from 2007 to 2015 (in million units) [9]

Also with regard to the consumption medium mobile devices increasingly replace the role of traditional outlets like TV, newspapers and other physical media.

In crisis situations, like natural disaster or unrest, people need to communicate and coordinate their efforts to restore safety. In this context the smart-phone becomes particularly important because it is often carried on person and provides connectability

In recent calamities people could mark themselves as safe on social media, effectively broadcasting that information to all their family and friends on social media instead of contacting them one by one or not at all due to congestion in the communication channels.

However, several natural disasters have taken out the necessary infrastructure on numerous occasions for a prolonged period of time.

Therefore we require a solution to enable social media on smart-phones that does not require infrastructure.

The main research question is: how feasible is it to run all Tribler functionality on mobile devices?

Secondary question is: using the unique properties of mobile devices, what features can be added or enhanced?

# 2

# Problem description

## 2.1. Privacy and censorship

Pervasive monitoring of digital citizens by Internet providers on behalf of governments to enforce censorship laws raises severe privacy concerns. The lack of anonymity becomes a problem when the users privacy is being invaded. Revealing personal information can be deduced from search queries for example, or associations on social platforms. When this information can be used for targeted advertising it becomes very valuable, and creates an incentive for the parties that have access to this information to sell it to third parties. Social media companies use targeted advertisement as part of their business model. Information considered private by users of social media is actually used to broker targeted advertisements. Subsequently users can be confronted with their information being misused in various ways beyond their control. This lack of control over your own privacy can lead to arbitrary interference as defined in UDHR article 12. Integration of social media on regular websites makes every page-view and click on these websites traceable to an individual, directly benefiting the business model of targeted advertisements

In fact the business model of social media appears to be serving targeted advertisements to its users on behalf of third parties. What's even worse is social media integrated into regular websites to de-anonymize and track the whereabouts of users even outside of the social media realm. Whenever users lose control over their privacy it becomes a serious problem.

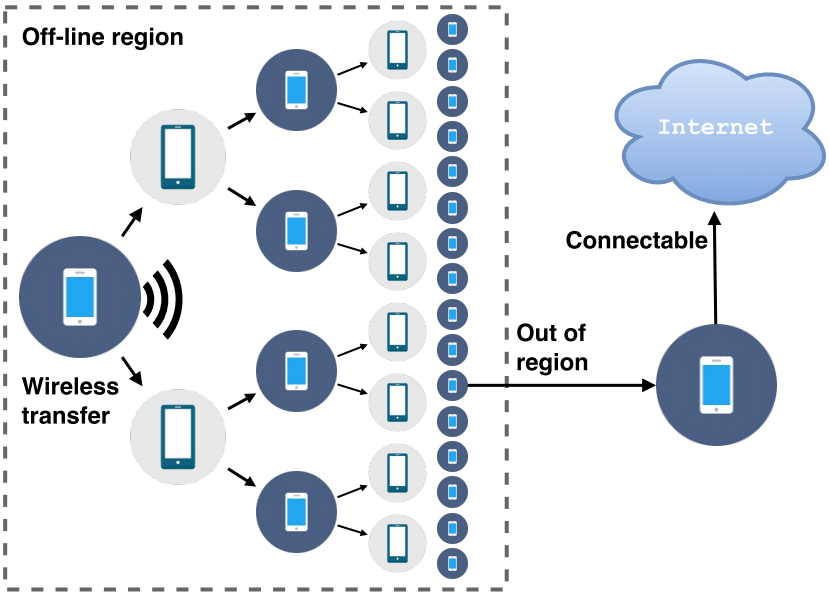


Figure 2.1: Viral spreading from one device to another within an off-line region

The incentive to de-anonymize the user, not only causes a lack of privacy, but also a potential lack of freedom of expression, as it hands key information to a censor: who is expressing dissent and who is associated

with this person on-line. Cyber-suppression has become a reality when you no longer can be associated with opinion-makers or foreign journalists on-line.

Internet exchange (IX) infrastructures are among the central components in the inter-network architecture that are also vulnerable to monitoring, censorship and Internet kill-switches. As such, not everyone has unrestricted access to the Internet due to censorship and surveillance. In fact a significant part of today's Internet users is affected by these attempts to hide or distort reality. This interference directly affects the universal right to freedom of opinion and expression as stated in article 19 of the Universal Declaration of Human Rights (UDHR).

For example: Arab Spring. Kill switches are real. [**?** ]

What are kill-switches?

Large portions of the global dialog on social media is uncontrolled by traditional media or governments.

Utilizing infrastructure is undesired because of censorship.

The sophistication of censorship techniques is pushed forward by the drive to stay ahead of attempts trying to circumvent it. Increasingly though, Internet traffic is put under surveillance and obfuscation techniques are targeted by restrictions.

### 2.1.1. Adversary model

From the Arab Spring scenario we know Internet kill switches are real, so we must assume the existence of a powerful adversary. The following threats [6] have been identified for similar circumstances:

- The adversary can observe, block, delay, replay, and modify traffic on the underlying network. Thus end-to-end security must not rely on the security of the underlying network.
- The adversary has a limited ability to compromise smart-phones or other participating devices. If a device is compromised, the adversary can access any information held in the device's volatile memory or persistent storage.
- The adversary can choose the data written to the transport layer by higher protocol layers.
- The adversary cannot break standard cryptographic primitives, such as block ciphers and message-authentication codes.

We assume the adversary cannot eavesdrop, jam, delay, replay, modify or spoof wireless communication between smart-phones. The adversary cannot compromise smart-phones or other participating devices.

To ensure that no controlling party can exercise censorship authority must be distributed over all users. If all information is located in one or a few places, the parties in charge of that location will still have control over it, so information must distributed all users as well, creating a *communication* system.

Then if all users want to use this system to share, order and appreciate each others information, in other words the essence of social media: social interaction, with everyone being able to interact in the same way, we need to distribute functionality over all users, creating a *cooperation* system.

## 2.2. Distributed solutions

Fully distributed systems capture these characteristics. To render the effect mute of the Internet kill switches in existence shown to be used we propose a distributed solution. Without any central component in the system it is no longer susceptible to censorship without everyone participating. Leveraging the properties of mobile devices, like smart-phones as stated above, together with the features of Tribler we see a perfect match. Because censorship and large scale monitoring is difficult in decentralized networks our proposed solution can work in these situations.

Networks that are not fully decentralized can be disrupted by taking down a limited number of nodes, less than the total number of nodes. Thanks to a server-less design we can say: The only way to take Tribler down is to take the entire Internet down.

Mobile devices are essential to take the next step in the face of censorship. If we can use smart-phones to create a fully decentralized social network we can say: The only way to take Tribler down is to take everybody's smart-phones away.

The importance of mobile devices in this context is crucial , if we want to do even better in spreading information, which is not been done before, we surmount than next hurdle.

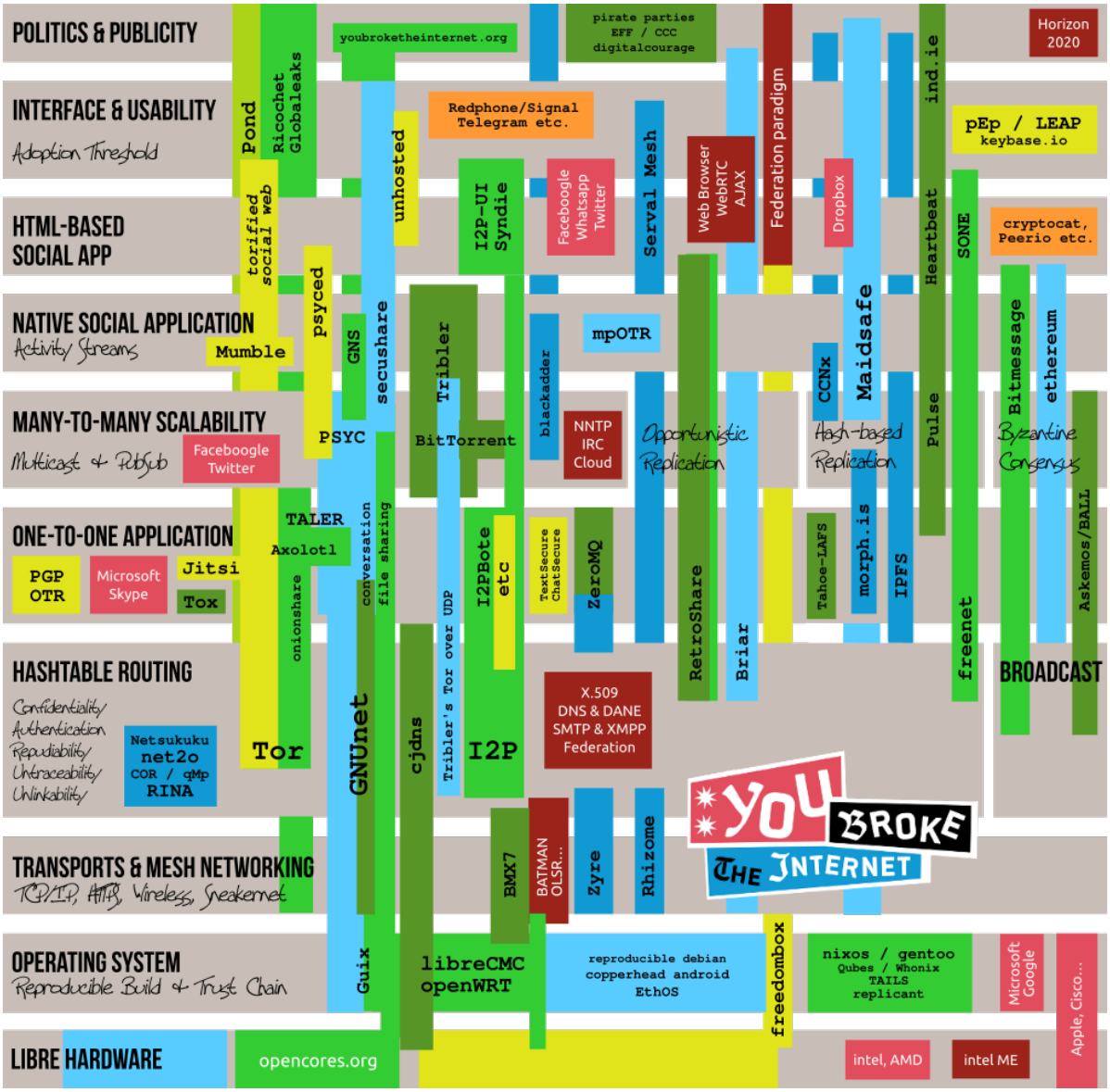


Figure 2.2: Source: www.youbroketheinternet.org

Peer-to-peer communication technology is essential for a server-less distributed system. Mobile devices typically do not require infrastructure to exchange information, like those equipped with Bluetooth or capable of ad hoc Wi-Fi. Smart phones are ubiquitous everywhere in the world and used to access social media and retrieve information from the Internet. Fortuitously these are also the type of mobile devices that can communicate peer-to-peer.

Example: Arab spring [**?** ]

Various initiatives have been started to deal with one or both of these problems. [**?** ]

Figure 2.2 shows a mapping of projects that are or have been working on that. The fragmentation is clear from the figure. There are a few big players, as can be seen from the figure, but none of them provide a full solution.

There is no de-facto solution available on mobile platforms. [**?** ] Previous research has shown that there is no solution that solves the problem entirely in a sustainable way. Tribler is our attempt to solve this problem entirely in a sustainable way. To see if it is feasible to apply it in a mobile context as described above, we need to port it.

The goal of Tribler is to become an information sharing platform / video-on-demand platform that protects the privacy of its users and is resilient to attacks while not relying on existing infrastructure. The goal of

this thesis project is to enable all Tribler functionality on mobile Android devices.

Previous attempts failed to deliver all functionality. [**? ? ?** ] Maintainability issues with earlier designs were a large part of the reasons why. We changed the architecture of Tribler for our approach.

*The main contribution of this work is the application of privacy-enhancing and attack-resilient technology to mobile devices and utilization of their unique properties in the context of censorship and communication during crises. The scientific contributions of this work are the experiments to verify the feasibility of Tribler on mobile devices and the ability to perform research with Tribler fully geared towards mobile devices in the future.*

*The main contribution of this thesis is making Tribler available on mobile devices. By doing that all research with regard to the problems that Tribler tries to solve becomes possible on the mobile platform. And the resilience of Tribler against attacks on the Internet is greatly increased.*

# 3

# Tribler functionality

## 3.1. Video-on-demand

Tribler introduces a server-less video-sharing platform with privacy enhancing technologies that provides a Youtube-like social media experience. Video-on-demand means that users can simply click and play videos in a streaming fashion, so without waiting for the entire video to be present on the device. You can search from within the application and browse for videos in channels rated automatically by popularity. Simply clicking on a video and watching it while streaming is supported via the BitTorrent-aware Tribler video server and integrated video player VLC.

## 3.2. Self-organizing

The BitTorrent protocol is used to download and upload the content. A BitTorrent swarm can use the "distributed sloppy hash table" (DHT) of a specific torrent to discover peers and gather meta-information. Also part of the DHT protocol is the self-organizing behavior of maintaining a routing table of known good nodes. Tribler uses these features to coordinate the exchange of videos and meta-data fully automatically. Users do not have to manage any files or configuration manually at all to be active on the platform.

## 3.3. Autonomous operation

New content discovery can be discovered automatically via a "Channel"-community that users can subscribe to. Communities are network overlays used by Tribler to offer functionality like search, add, remove and comment on content. To discover all channels in existence Tribler is subscribed to the so called "AllChannel"-community and "Search"-community by default. These are used to exchange information about what channels and torrents are out there and who likes them and knows about them. Each channel has its own community that rules permissions and meta-data of content. These communities operate autonomously and are transparent to the user, who only sees channels and search results show up in the GUI.

## 3.4. Attack-resilience

The server-less technique of Tribler is resistant to large scale monitoring and censorship, because there is no central point that can be controlled to gather or block information easily. To monitor or censor the network effectively on a large scale you need control of a significant number of the communication links. Censorship does not have an effect if the majority of users does not cooperate with the censor. The beauty of a fully distributed design is that communicating directly between peers, or via a local network, works without the need for external communication links. This is why the server-less technique of Tribler is resistant to Internet kill-switches as well because, even if the attacker can block all communication-links, users can always connect off-grid. Such kill-switches are typically deployed for the purpose of censorship, but won't stop a connectible device, like a laptop, from physically moving. No network infrastructure is required for viral spreading of the entire video platform.

These properties will ensure social media with resilience against Internet kill switches, natural disasters and censorship.

## 3.5. Anonymity

Tribler can protect the privacy of users by hiding their identity. To connect to others on the network anonymous connections are created on behalf of downloaders and uploaders. By routing the network traffic over a circuit of multiple hops it becomes difficult to trace the origin and destination. This way the privacy of users remains protected while they actively participate on the platform

Multiple encrypted tunnels are layered such that every consecutive tunnel from the initiator to a relay is going through the previous tunnel. Every relay works on behalf of its predecessor so no relay knows the identity of the initiator save for the first relay. Since all communication is properly encrypted no relay can perform a successful man-in-the-middle attack.

This is similar to how Tor works, except Tribler uses UDP rather than TCP for performance reasons.

The capability of hiding your identity is greatly advantageous to the user if his or her human rights are violated, like free speech.

However, this privacy feature requires a lot more bandwidth of the network than without anonymity: a ratio of 13 GB for every anonymous 1 GB of data. Since bandwidth is limited and transitory it can be beneficial to exchange unused bandwidth for a promise of bandwidth in the future, or another reward. The research group behind Tribler is currently building a fully decentralized accounting system and open exchange market using block-chain technology with the purpose of building trust on-line and creating the Internet of Money.

## 3.6. Towards Tribler on mobile devices

In the last chapter we discussed the usefulness of Tribler functionality running on desktop and server machines. To bring that to mobile devices will give access to the usefulness of Tribler for people on the move. Expanding the Tribler network would also improve the usefulness of experiments on the live network for this large scale system.

Properties (positive and negative) that come from these features are transferred to mobile devices, which will add their own distinct properties to the mix.

How to create a *self-organising video-on-demand* platform that is *attack-resilient* and can *operate autonomously* on a *mobile device*?

Mobile device in the sense that it is low-powered and portable including the network interface and power supply.

### 3.6.1. Opportunities

What does mobile allow in addition to desktop? Mobile devices are inherently easy to move around and very portable. In case of a breakdown in communication infrastructure the mobile devices with wireless radio transmitters can still connect ad hoc and moved within range if necessary. Via WiFi a device can connect to the Tribler network via existing infrastructure or other peers via ad-hoc WiFi. Using NFC Tribler can start a Bluetooth connection and transfer the installation package peer-to-peer. Also via NFC Tribler can exchange channel ID's to subscribe to another Tribler channel peer-to-peer. The same method can be used to exchange bootstrap peers for the Tribler network. The peer-to-peer WiFi Direct file transfer would be much faster than Bluetooth.

### 3.6.2. Challenges

What does working on mobile phone mean? Mobile devices are typically equipped with batteries to operate without a power cord. Considering the size and weight the capacity is limited. Smart-phone batteries usually barely hold a charge that would sustain a day of heavy usage. Tribler could potentially drain the battery much faster than normal. Heavy encrypted network traffic not only demand constant radio transmissions, but also CPU processing. In case of hidden seeding building circuits of 3 tunnels with layered encryption quadruples the amount of crypto work. Because Multichain punishes cheating like double spending by a permanent ban, it must never loose information and flush everything to permanent storage before continuing. Mobile devices typically have flash memory with limited write-cycles compared to classic hard drives that are commonly found in desktop computers.

# 4

# Design and architecture

We now present a design to address the challenges and utilize the opportunities of working on a mobile device as put forward in the previous chapter. First in terms of functionality and other requirements, second the overall system architecture.

## 4.1. Design principles

Previous attempts at designing a mobile version of Tribler failed to properly separate re-usable components. This resulted in unmaintainable code and increased difficulty in testing. We use a top down approach to come to a high level system architecture and in determining the reusable components or the current Tribler code. Some functionality of Tribler has been shown to work on Android before. Our design will fill in the gap of connecting the pieces of the grand puzzle to make Tribler fully work on mobile.

Recent refactoring of the Tribler code has divided functionality into core modules. The purpose of this was to increase cohesion inside modules and the core, and reduce coupling between modules where possible. This makes porting the core and individual modules easier. Our design also attempts to maximize re-use-ability with separation of components and clearly defined interfaces. An API between front-end and back-end is the industry standard of choice here. Having separated the GUI from the back-end, we can easily re-use the Tribler core and ditch the outdated wxPython GUI. The new GUI can concentrate on Android and smart-phone user interaction. Many established best practices, like Google's "material design", and support libraries that abstract from platform specifics can be re-used. Instead of re-implementing multi-media functionality and player interface, we should re-use an existing library or preferably an entire app. The flexibility of the design benefits from cohesive components with a clearly defined role. For example: being able to swap out components like the GUI makes Tribler much more adaptable to various mobile devices that may or may not have a display. The configuration of Tribler core is stored separately from the data and can be altered externally by users since it is saved in a human readable form.

All functionality of Tribler must be packaged in a single installable container. The GUI must be able to be tested separately from the back-end and vice-versa. Also the API must be able to be tested independently and be clearly documented for other developers.

All Android intents should be explicit. [**?** ] Expose / Export only Android activities and services that should be public

A distinction must be made between user input error and internal exceptions. The user must be given a chance to correct the input without starting over. If Tribler crashes it should resume normal operation by just restarting it and optionally ask the user to do so. The GUI should support other languages to be added without changing source code. The separation of front-end and back-end should not lead to resources being loaded into memory twice or anything of that matter. Make use of universally recognizable icons that are native to the platform UI. The responsiveness of the GUI must not be significantly impacted by the amount of data presented on screen. The responsiveness of the GUI must also not be significantly impacted by the amount of computational work, or otherwise, done in the background.

## 4.2. Functional requirements

Live production (screenshots) and dissemination Record video Create My Channel Add to My Channel Search results

Multichain is explicitly designed to not use a global state. Syncing a global state is less scalable in terms of storage and network bandwidth.

Start quickly / delay

JSON REST API

GUI

Nosetests CI

All functionality of Tribler must be usable directly from the mobile device, without the need or support for any other external device.

Because of the ubiquity of smart-phones with built-in camera one is always at hand and ready to record. Such mobile devices are much more manageable than for example laptops with built-in camera that Tribler has been capable of running on in the past. No opportunity has to be missed due to hassle of getting a dedicated camera and transferring the recording to a connectible device afterwards. Therefore the camera built-in the mobile device should be used to record video or photographs with Tribler. The usability of Tribler as a content generating tool is greatly improved by this.

Publishing content should be one simple step for the user to perform, especially right after creating new content. Every user must be able create his/her own channel in Tribler to publish their content, just like channels in YouTube, directly from the mobile device. Newly discovered channels and content may be added to the GUI without user interaction.

To easily share your own channel with others the id could be transferred via NFC from device to device. With just one-click-confirmation it should be send, and the receiving device must then be able to subscribe to this channel and start discovering the contents automatically right away. Because this may require both devices having Tribler installed, it also must be able to share the installation package via Bluetooth or WiFi without any prerequisites on the receiving device.

Self-created content must be stored on the internal memory and processed automatically on the mobile device itself in order to avoid dependency any external processing unit. This includes the creation of .torrent files required to share content via the bit-torrent protocol.

## 4.3. Non-functional requirements

All processing tasks should be performed in the background, so as to allow the user to continue working with Tribler or otherwise, while indicating about ongoing processing. The GUI must be responsive to the users' input at all times, even if the amount of data shown on screen is very large.

If an error occurs that is recoverable the app should automatically retry or ask the user to correct the input if applicable.

To keep the project maintainable the entire build tool-chain must be integrated in the continuous integration and build server of Tribler. All existing tests should also be performed on Android, with modifications for platform specific tests. Also all unit tests and integration tests must cover as much code as possible and should be executed on every build.

Search results must be near instant (less than 1 second) for local content.

Tribler must run on the most used mobile processor, compatible with ARMv7. If an exception occurs the user must be able to restart Tribler and continue using it as normal.

Tribler may crash but should always be able to return to a working state. The Python Tribler core has to be used without major modifications. The minimum API level of Android should support WiFi, Bluetooth and NFC features.

The Tribler core is written in Python so the platform must support that.

## 4.4. System architecture

Previous attempts at creating an Android version of Tribler failed with regards to maintainability. Poor separation of front-end and back-end meant re-use of the core modules was hard to maintain. The Tribler core patching for example meant that all patches would have to be re-applied ever change and that has proven to not be workable.

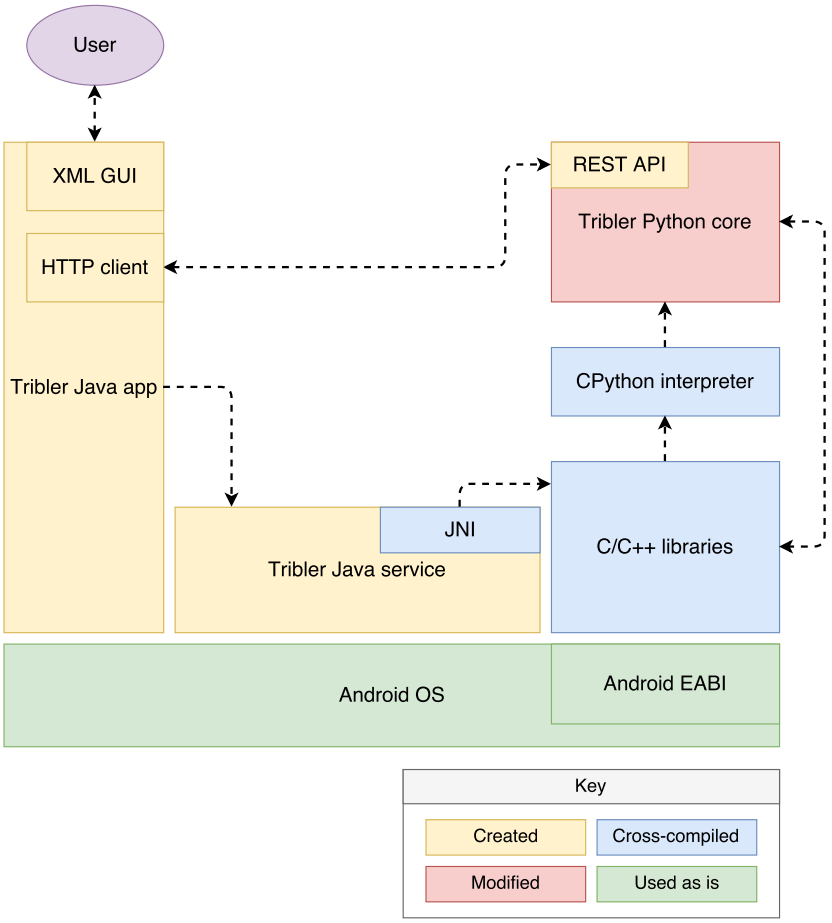


Figure 4.1: System architecture

The new architecture clearly separates tasks of the GUI and background operations. Figure 4.1 shows the distinct modules of the system architecture of Tribler for mobile. The Java app is considered the front-end and the Java service and other components are considered the back-end. The HTTP Client communicates with the REST API in JSON. The Java app starts the Java service with an Android intent. The Java service loads the native libraries and starts the CPython interpreter with a call over JNI. The Python code loads other Python and native modules directly.

# 5

# Implementation

Tribler To implement our design in chapter 4 we

## 5.1. Common Tribler core

Re-usability is an important design objective that can greatly improve maintainability as well. Therefore we aim to completely re-use the Tribler core, meaning everything but the GUI. We focus on implementing all the features of Tribler as described chapter 3. Tribler is written entirely in Python, but most of its dependencies are written in C/C++. To use these libraries on a mobile device they need to be compiled for the right embedded-application binary interface (EABI) including all nonstandard dependencies. Figure 5.1 shows the dependency tree.

Calling C code directly from Python is possible by using the Python ctypes module to load a native dynamic-link library (.so files on Android) or by using the Python/C API of CPython. This API enables a library to define functions that are written in C to be callable from Python. These Python bindings are the glue between pure Python and pure C code. SWIG can generate the boiler plate code for this. Libtorrent, one of Triblers' main components, uses Boost.Python to provide a standard C++ API on top of the Python/C API.

The Python/C API is actually so powerful it even provides access to the internals of the interpreter to mess with the global interpreter lock (GIL) which could be released during native C calls to improve the multi-threading performance of Tribler crypto.

### 5.1.1. REST API

Tribler uses the event-driven networking engine Twisted, which is also written in Python. Twisted allows you to build inter-process communication protocols and provides a HTTP server which was used to build the REST API.

Twisted uses a single thread to coordinate all others, called the reactor thread. If this thread is busy, the REST API can not receive incoming requests resulting in timeouts. We will investigate this in the next chapter.

## 5.2. Android platform

Due to time constraints we need to focus our implementation effort on a single platform. Of all mobile devices smart-phones fit our design requirements best. On smart-phones Android has the largest market share [**?** ]. We therefore aim our implementation at the largest group of users.

ARMv7 is currently supported by most Android phones and a large part of mobile devices are also compatible with this EABI, so it is the first to be supported by Tribler for now.

Since API level 18 (Android 4.3 codenamed Jelly Bean MR2) loading native libraries dynamically from Python works properly (apparently). API level 18 also added support for Bluetooth Low Energy (LE), WiFi scan only mode, key store for app-private keys, hardware credential storage and automated UI testing. All very useful to Tribler. Earlier versions of Android already supported NDEF Push with NFC and Wi-Fi P2P (since API level 14), and initiate large file transfers over Bluetooth via NFC (since API level 16). 83.7% of devices on the Google Play Store run Android 4.3 or higher.

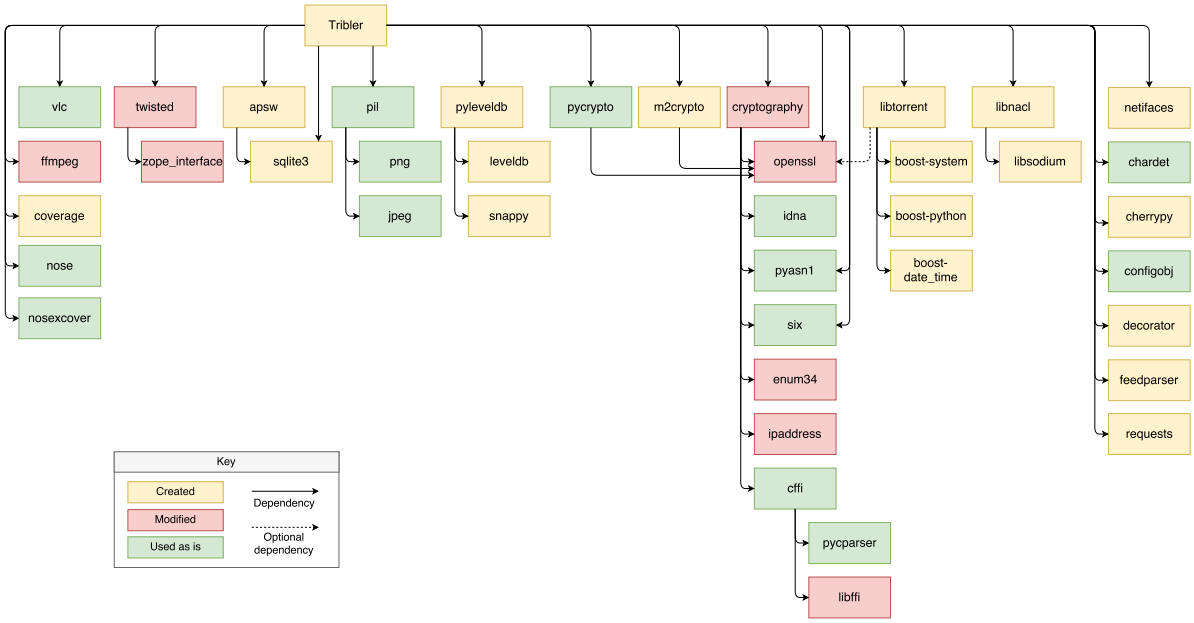


Figure 5.1: Tribler dependencies in terms of Python-for-Android recipes

## 5.3. Python on Android

What alternatives are there, besides P4A, to run python code on Android? 0. QPython: scripting, cannot build regular .apk 1. QtAndroid, inmiddels alternative, niet gereed toen wij hieraan werkte (juni 2016, qt 5.7 android service) 2. PGS4A: no longer in development 3. SL4A: no longer in development Why is P4A chosen? Because this project continued from previous work (legacy code) build on P4A. Even though the revamp version was build from scratch, it still is the best choice because it not only provides the Python interpreter, it also is a complete tool-chain to cross-compile native libraries with bindings and build a standalone Android app installation package (.apk).

P4A uses the Java native interface (JNI), between Java and C/C++ code, to launch the CPython interpreter from a thin Java Android application. JNI enables to define functions that are written in C to be callable from Java. On top of that the entire core of Tribler can run, containing a REST HTTP API module also written in Python. The GUI is created by a native Android Java application, which talks to the REST API module. This Java application contains an Android service that wraps the original P4A CPython launcher. I changed this behavior to avoid having the GUI and back-end running in the same Python process, hindered by the GIL. This approach is also superior to having two separate Python interpreters in distinct processes talking to each other, because that means using a very resource heavy Python GUI instead of the regular and lightweight native Android Java XML GUI. The latter has tools available for automated UI testing.

### 5.3.1. Build tool-chain

The Python-for-Android tool-chain uses recipes to cross compile the C/C++ libraries with Python bindings with the necessary build tools. These recipes are like a high level make file.

## 5.4. Native GUI

Traditionally applications with a user interface execute tasks in the background while showing the progress to the user. The design choice of separating the GUI from the back-end means no longer an unresponsive GUI, like older versions of Tribler are used to.

Due to the fact that Android targets mobile devices it is very optimized for low resource usage. Therefore memory is freed more aggressively and the application is often paused or stopped and restarted if the user switches to another app. Running two interpreters with Python Kivy front-end and Python Tribler back-end would be doubling memory usage for the interpreter itself and require an inter-process protocol as well. Native Java with XML front-end and Python Tribler back-end brings the user experience seamlessly in line with the native UI.

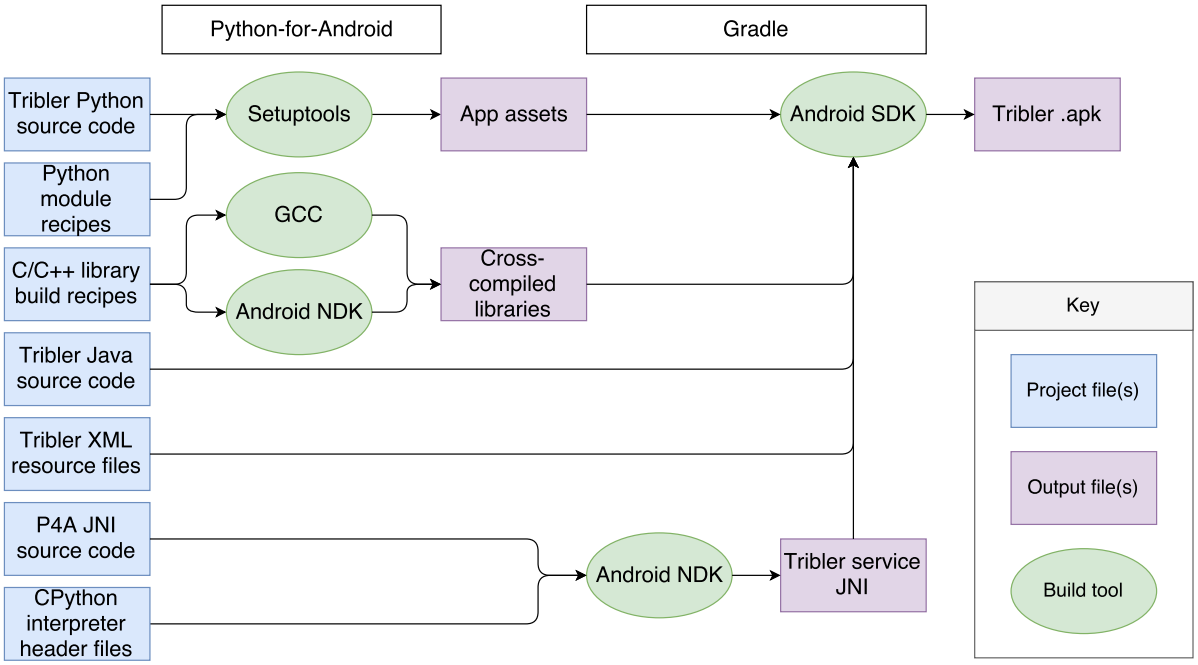


Figure 5.2: High level overview of the build tool-chain

To run in the background Tribler uses an Android service and all communication is performed asynchronously. The reactive programming paradigm is a perfect fit for asynchronous tasks. Thanks to RxJava and RxAndroid asychronous multi-threaded coding is made very enjoyable: As shown in the code example performing IO tasks on the dedicated Android thread and making UI changes on the main thread becomes trivial. The HTTP client that talks in JSON with the REST API is build on the popular library OkHttp and fits perfectly to RxJava with a library called Retrofit. The Retrofit library enables a very declarative API client.

Because of the nature of Android to destroy interface elements if a configuration chance occurs, the asynchronous tasks running in the background must be registered and unregistered properly to avoid memory leaks. To detect notorious memory leaks on Android we use another library made to do exactly that: LeakCanary. Android provides a way to deal with continuity of activities with fragments that can remain in memory which were used in conjunction with composite subscriptions of RxJava.

### 5.4.1. Videoplayer

The video player VLC can be integrated as hard to maintain library and a custom GUI. This was done first, but afterwards we decided to improve maintainability by packaging the entire Android app installation package (.apk). Which was also much easier to accomplish in hindsight.

The API combined with Java is better for parallelization than the coarse grained locking by the CPython interpreter. Shared memory threading in Python code is restricted to single-thread performance because of the global interpreter lock (GIL).

The implementation consists of: 12.806 + X lines of code in the Tribler repository in 35 pull requests and Y lines of code in the Python-for-Android (P4A) repository in 46 pull requests and 2 lines of code in the M2Crypto repository in 1 pull request. This excludes the work of reviving the old P4A toolchain.

Of the X lines (x %) consists of unit testing code. Y lines (y %) is made up of Android specific libraries, and the remainder is.. Table 5.1X shows the 25 largest source file contributions for this thesis work.

Code coverage and testing details are further explained in chapter 6.

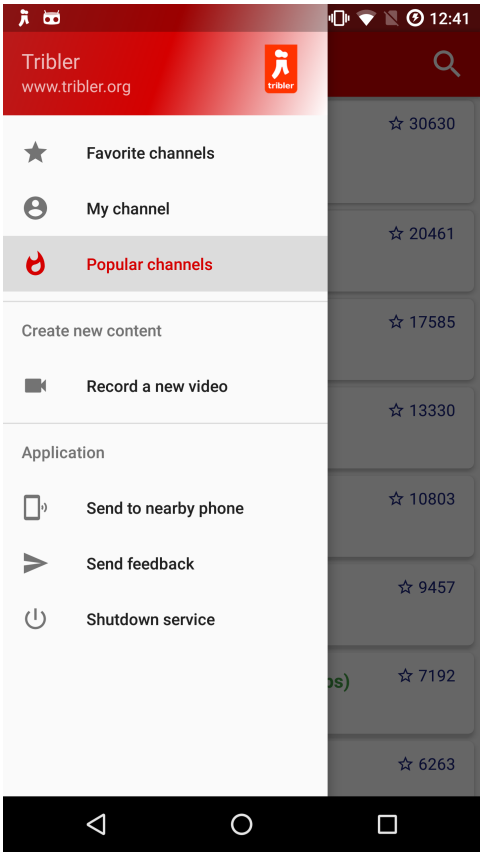


Figure 5.3: Navigation menu of the Tribler app

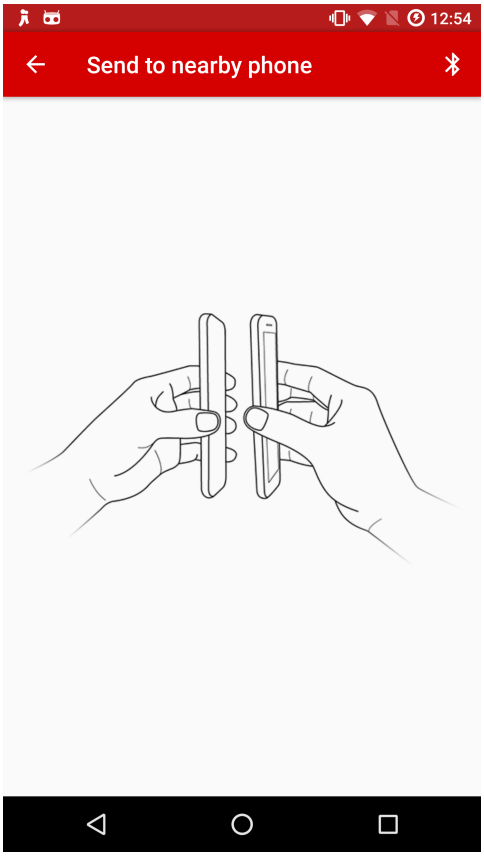


Figure 5.4: NFC+Bluetooth transfer of app or channel

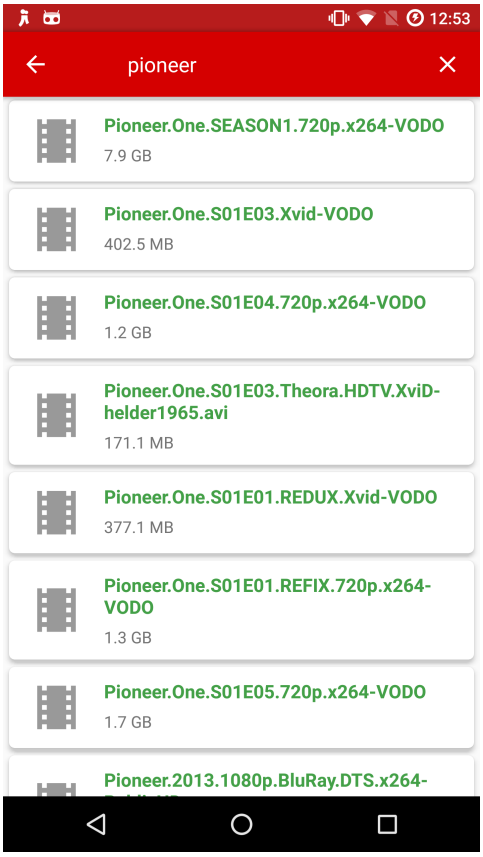


Figure 5.5: Search results showing video content

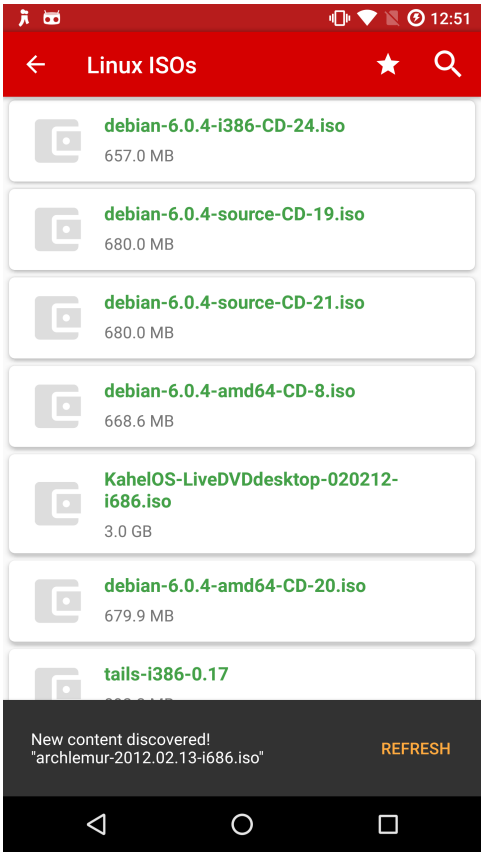


Figure 5.6: Channel with newly discovered content

| LOC | File | Path |
|---|---|---|
| 718 | MainActivity.java | .../org/tribler/android/MainActivity.java |
| 666 | MyChannelFragment.java | .../org/tribler/android/MyChannelFragment.java |
| 482 | MyUtils.java | .../org/tribler/android/MyUtils.java |
| 403 | DefaultInteractionListFragment.java | .../org/tribler/android/DefaultInteractionListFragment.java |
| 318 | start.c | android/TriblerApp/app/src/main/jni/src/start.c |
| 314 | TriblerViewAdapter.java | .../org/tribler/android/TriblerViewAdapter.java |
| 281 | build.gradle | android/TriblerApp/app/build.gradle |
| 256 | IRestApi.java | .../org/tribler/android/restapi/IRestApi.java |
| 234 | ChannelFragment.java | .../org/tribler/android/ChannelFragment.java |
| 186 | AssetExtract.java | .../org/kivy/android/AssetExtract.java |
| 182 | BeamActivity.java | .../org/tribler/android/BeamActivity.java |
| 175 | ListFragment.java | .../org/tribler/android/ListFragment.java |
| 172 | CopyFilesActivity.java | .../org/tribler/android/CopyFilesActivity.java |
| 166 | AndroidManifest.xml | android/TriblerApp/app/src/main/AndroidManifest.xml |
| 166 | EventStreamCallback.java | .../org/tribler/android/restapi/EventStreamCallback.java |
| 155 | ChannelActivity.java | .../org/tribler/android/ChannelActivity.java |
| 150 | ViewFragment.java | .../org/tribler/android/ViewFragment.java |
| 149 | PythonService.java | .../org/kivy/android/PythonService.java |
| 149 | SearchActivity.java | .../org/tribler/android/SearchActivity.java |
| 141 | EditChannelActivity.java | .../org/tribler/android/EditChannelActivity.java |
| 131 | FilterableRecyclerViewAdapter.java | .../org/tribler/android/FilterableRecyclerViewAdapter.java |
| 130 | BaseActivity.java | .../org/tribler/android/BaseActivity.java |
| 114 | SearchFragment.java | .../org/tribler/android/SearchFragment.java |
| 112 | TriblerDownload.java | .../org/tribler/android/restapi/json/TriblerDownload.java |

Table 5.1: Top 25 of largest source file contributions.

# 6

# Verification and validation

To verify if running all Tribler functionality on mobile devices is feasible we take a look at relevant performance characteristics and take several measurements. For scale a laptop and PC are included in some measurements. We focus on the design aspects and evaluate the key performance indicators:

- *usability* in terms of latency and startup time,
- *resource usage* in terms of CPU utilization,
- *performance* in terms of frequency and duration of function calls,
- *scalability* in terms of discovery time and bandwidth accounting.

## 6.1. Content discovery

New content is generated on the device, like for example a video that has been recorded. Before anyone can view that content it has to be added to a channel and discovered by other devices. We measure the amount of time it takes for other devices, which are subscribed to the channel, to discover new content. We also measure the amount of time it takes for content to be added to a channel in the form of a torrent. The amount of time it takes for a torrent to be created is relative to the size of the content. Therefore these measurements are normalized to a file of 1 MB. Depending on the random walk in the channel's community content can be discovered either very quickly or after a while due to property of eventual consistency. On one Nexus 6 device a channel is created to which 13 other devices subscribe via NFC. Then repeatedly a new video is recorded and added to that channel. A content discovered event is registered on each device individually and all devices are synced with NTP. Each device is connected to the same WLAN and within 1 to 2 meters distance from the access point. Figure 6.3 shows the amount of time it takes a number of devices to discover new content on a subscribed channel. The results show that within 4 seconds 9 devices have discovered the new content. From the 10th device and beyond an increase in discovery time is noticeable. This could be explained by the fact that only 10 peers are connected at the a time. Much more peers are needed to give an accurate representation in terms of scalability. What can be concluded from this experiment is that on the same local network the first device discovers the new content in less than 2 seconds. From the 10th device onwards the dissemination slows down.

## 6.2. Startup time

Key to user retention is a fast startup time. Therefore the service starts up in the background, separately from the GUI. This way at the GUI is responsive to user input while the service may still be loading. One of the design objectives for separating the front-end from the back-end was responsiveness. However before any task can be executed by the service it needs to be actually started. To measure the startup time we register the time of launching the app and the moment the Tribler-started-event occurs. This event is sent over the API event-stream and signifies that Tribler is fully started and ready to accept all incoming requests. We expect consistent loading times because right after starting we shut Tribler down again. To get a good idea of how the user experience may differ we measure the startup time 10 times on 5 different devices. The app is launched


Figure 6.1: All devices used in the experiment showing the about Android screen.

| Device | N | Avg. (s) | Min. (s) | Max. (s) | s (s) |
|---|---|---|---|---|---|
| Nexus 10 | 10 | 3.781 | 3.416 | 4.085 | 0.211 |
| Nexus 6 | 10 | 4.319 | 4.124 | 4.670 | 0.179 |
| Nexus 5 | 10 | 3.353 | 3.273 | 3.459 | 0.081 |
| Galaxy Nexus | 10 | 7.086 | 6.161 | 7.772 | 0.454 |
| S3 | 10 | 31.935 | 30.616 | 33.940 | 1.116 |

Table 6.1: These numbers are based on elapsed time, not latency, from the entire benchmark.

with Android Debug Bridge (adb) from a laptop and the Tribler-started-event is read from logcat over adb and timed on the same laptop so they use the same clock. Table 6.1 shows the statistics per device. The results show a very small sample standard deviation and a very low startup time. The S3 is performing worse than may be expected judging from the results of the other devices. The reason for that may be that this phone was not wiped and given a fresh install of Android. That could mean that other applications installed on a device could significantly impact the performance of Tribler. The sample standard deviation is relatively small though, which contradicts this hypothesis. This should be investigated further, including if anything can be done on the part of Tribler.

## 6.3. API latency

The user expects operations to take a consistent and reasonable amount of time. By design all functionality is going through the API. We use Apache JMeter to verify if the API is responding consistently within a reasonable amount of time by measuring the latency. JMeter measures the latency from just before sending the request to just after the first response has been received. [**?** ] Thus the measurement includes the time needed to assemble the request as well as processing it and returning a response with latency on the way back. It excludes the transfer time of the complete response and subsequent processing and rendering time. Any client can process and render a response differently, possibly in a streaming fashion. By leaving out that element this metric is measuring the operation time via the API. We want to see that the latency is bounded, consistent and generally low. Previous research has shown the following results on a computer: We did the same benchmark with a slightly different setup. A Nexus 6 smart-phone with Android 6.0.1 Cyanogen mod was connected via USB to a laptop running JMeter. The API tcp-port was forwarded with ADB. With JMeter

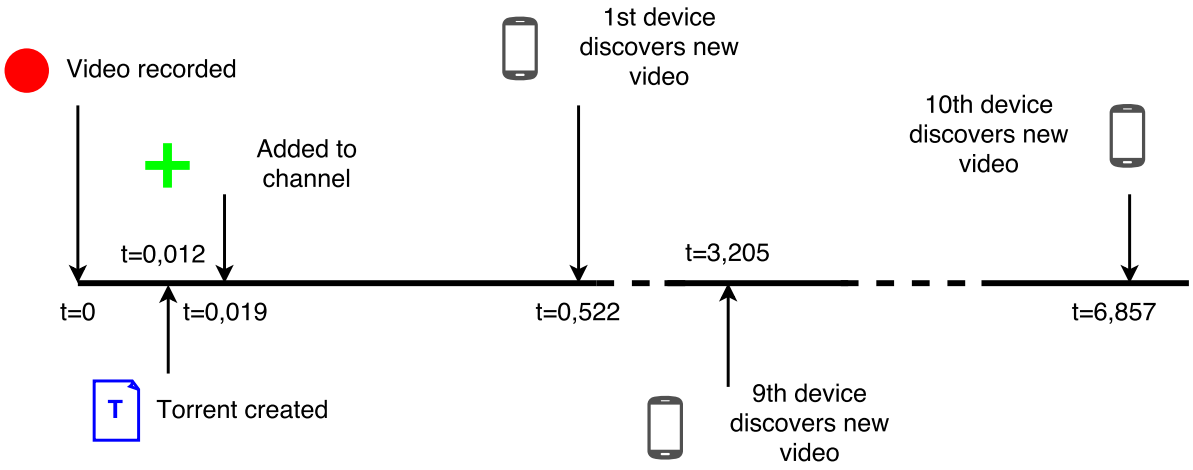


Figure 6.2: Sequence of events with time values from median of results.

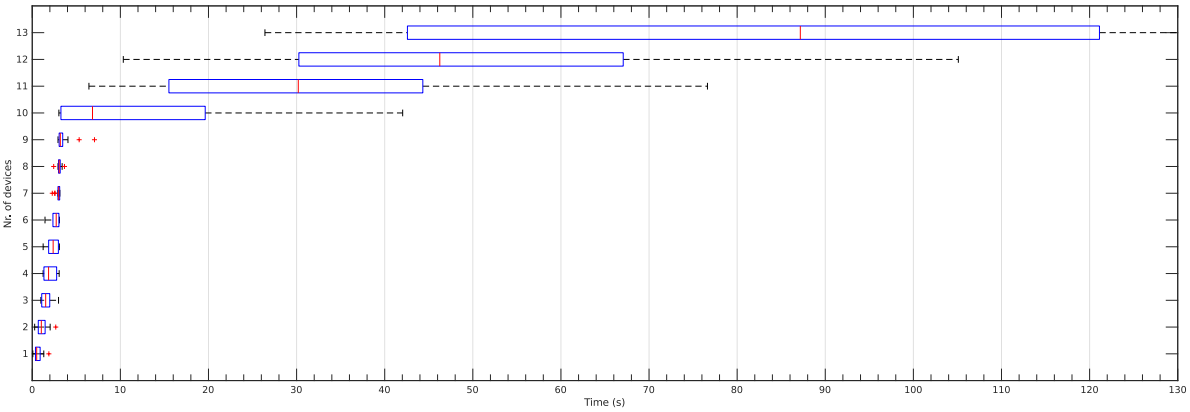


Figure 6.3: Elapsed time from adding new content to a channel and it being discovered by subscribed peers.

we request the discovered channels from the API a 1000 times and measure the latency. JMeter is also capable of setting a desired number of requests per minute, provided the device can handle it. We set this parameter to 300 requests per minute, equal to the other benchmark. Our database to be queried is not equal to one used in figure 6.6. The following figure shows the latency for every request. The plot shows large spikes and seemingly a curious and unexpected pattern. Table 6.2 shows the large difference in statistics. PC is added for reference. The set rate of 300 requests per minute is clearly not reached by the smart-phone in our benchmark by a factor of 9. The difference in amount of response data turned out to be much bigger than expected with a factor of 35. IO bound + crappy Python code WHY? Considering the difference in amount of response data and the bandwidth of USB versus internal memory it is an unfair comparison. However, based on our result we may conclude that the API does not respond consistently and appear to behave differently than the other benchmark. The device not being able to process 5 requests per second possibly reveals information about the connection bandwidth and the multi-threaded processing capability. We assume that the 480MB/s theoretical bandwidth of USB2.0 is not a bottleneck considering the 2,5 MB/s result of the PC. Being a mobile device also other aspects my be at play here, like CPU frequency scaling. However this was turned off by acquiring a wake lock from the Android OS. It appears then that our design approach still suffers from performance issues due to imperfect multi-threaded performance. Because if the CPU is simply not powerful

| | N | Avg. (ms) | Min. (ms) | Max. (ms) | $\sigma$ (ms) | Throughput | KB/second | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|
| Nexus 6 | 1000 | 1803 | 1292 | 3292 | 301.25 | 33.3 req/minute | 347.44 | 641709.0 |
| PC | 1000 | 5 | 3 | 107 | 4.42 | 140.8 req/second | 2548.00 | 18525.0 |

Table 6.2: These numbers are based on elapsed time, not latency, from the entire benchmark

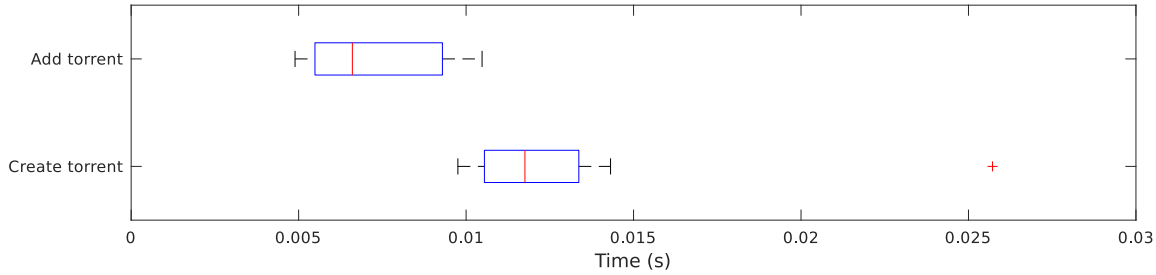


Figure 6.4: Time required to create a torrent and add it to a channel, normalized for 1 MB of content.

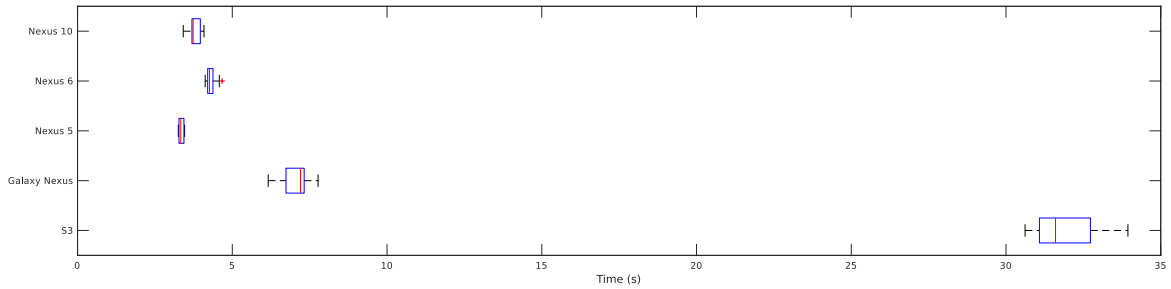


Figure 6.5: Startup times per device

enough we expect to see a linear pattern instead of what we actually see in 6.7. This phenomenon is not of great importance though, because users are not expected to fire hundreds of requests per minute to the API. Further investigation should figure out if this phenomenon is also observed with lower amounts of requests per minute.

## 6.4. Profiling

Because of the challenges put forward in chapter 3.6 we investigate if time is spent disproportionately on some function. We expect that the limited resources of a mobile device may impact particular features more than others. If hardware acceleration is not present the less powerful CPU may struggle with encryption tasks. Instead of CPU time, time actually spent processing by the CPU, we measure wall-clock time. This way we measure the amount of time a user would have to wait for a certain function to be executed. We focus on wall clock time instead of CPU time because...

With the cProfile Python module and the visualisation tool SnakeViz we can see if any function takes a disproportionate amount of time. A Nexus 6 smart-phone with Android 6.0.1 Cyanogen mod was used for profiling Tribler. The profiler was running for 10 minutes with Tribler during normal operation and without any user input. Figure 6.8 shows that 27% of the time is spent on verifying cryptographic signatures. The bright pink represents the update function, which signifies various business logic upon receiving a torrent. Also notable is the same stack of functions within and outside of a community on top of the wrapper. This can be explained by the fact that torrents can be discovered outside of a community too. execute of apsw.Cursor execute of sqlite3.Cursor select.epoll [**?** ] The time per call is most important for optimization because...

The significant chunk of time that the crypto takes is as expected. Since this task is actually delegated to the C library M2Crypto it should be possible to release the GIL of the Python interpreter so other Python code that does not depend on it can be executed. The main alternative provided in the standard library for CPU bound applications is the multiprocessing module, which works well for workloads that consist of relatively small numbers of long running computational tasks, but results in excessive message passing overhead if the duration of individual operations is short [**?** ]. As seen from the time per call for __m2crypto.ecdsa_verify in table 6.3 the multiprocessing module would likely cause too much overhead.

| # Calls | Total time (s) | Time per call (s) | Function |
|---|---|---|---|
| 15 | 0.4867 | 0.03245 | method 'commit' of 'sqlite3.Connection' objects |
| 1 | 0.01692 | 0.01692 | method 'executescript' of 'sqlite3.Cursor' objects |
| 3820 | 64.28 | 0.01683 | method 'poll' of 'select.epoll' objects |
| 2 | 0.01048 | 0.005241 | __m2crypto.ec_key_gen_key |
| 31075 | 162 | 0.005212 | __m2crypto.ecdsa_verify |
| 1650 | 7.133 | 0.004323 | __m2crypto.ecdsa_sign |
| 1 | 0.001708 | 0.001708 | _socket.gethostbyaddr |
| 1 | 0.001284 | 0.001284 | built-in method SSL_library_init |
| 567 | 0.565 | 0.0009965 | method 'executemany' of 'sqlite3.Cursor' objects |
| 8 | 0.005731 | 0.0009552 | __import__ |
| 12 | 0.01083 | 0.0009029 | method 'connect_ex' of '_socket.socket' objects |
| 1 | 0.00055 | 0.00055 | built-in method SSL_load_error_strings |
| 5 | 0.002515 | 0.000503 | method 'recv' of '_socket.socket' objects |
| 6989 | 3.436 | 0.0004917 | method 'executemany' of 'apsw.Cursor' objects |
| 1 | 0.000485 | 0.000485 | dir |
| 63677 | 20 | 0.000314 | method 'execute' of 'apsw.Cursor' objects |
| 5546 | 1.38 | 0.0002488 | __m2crypto.ec_key_read_pubkey |
| 15943 | 3.414 | 0.0002141 | method 'sendto' of '_socket.socket' objects |
| 44 | 0.009405 | 0.0002137 | open |
| 1 | 0.000212 | 0.000212 | built-in method OpenSSL_add_all_algorithms |
| 2 | 0.000405 | 0.0002025 | __m2crypto.ec_key_new_by_curve_name |
| 2 | 0.000394 | 0.000197 | netifaces.interfaces |
| 296 | 0.05179 | 0.000175 | method 'sort' of 'list' objects |
| 5 | 0.000826 | 0.0001652 | __m2crypto.ec_key_read_bio |
| 1 | 0.000147 | 0.000147 | __m2crypto.rand_seed |
| 4 | 0.00058 | 0.000145 | netifaces.ifaddresses |
| 47 | 0.005664 | 0.0001205 | androidembed.log |
| 12 | 0.001445 | 0.0001204 | thread.start_new_thread |
| 8 | 0.000936 | 0.000117 | posix.mkdir |
| 2240 | 0.2615 | 0.0001167 | posix.open |
| 17 | 0.001964 | 0.0001155 | compile |
| 3 | 0.00034 | 0.0001133 | __m2crypto.ec_key_write_bio_no_cipher |
| 5553 | 0.6196 | 0.0001116 | __m2crypto.ec_key_write_pubkey |
| 7 | 0.000777 | 0.000111 | method 'send' of '_socket.socket' objects |
| 140069 | 15.4 | 0.00011 | method 'execute' of 'sqlite3.Cursor' objects |
| 16 | 0.001759 | 0.0001099 | method 'shutdown' of '_socket.socket' objects |

Table 6.3: Native function calls wall clock time broken down per call during the 10 minute profiling (600 seconds total time)

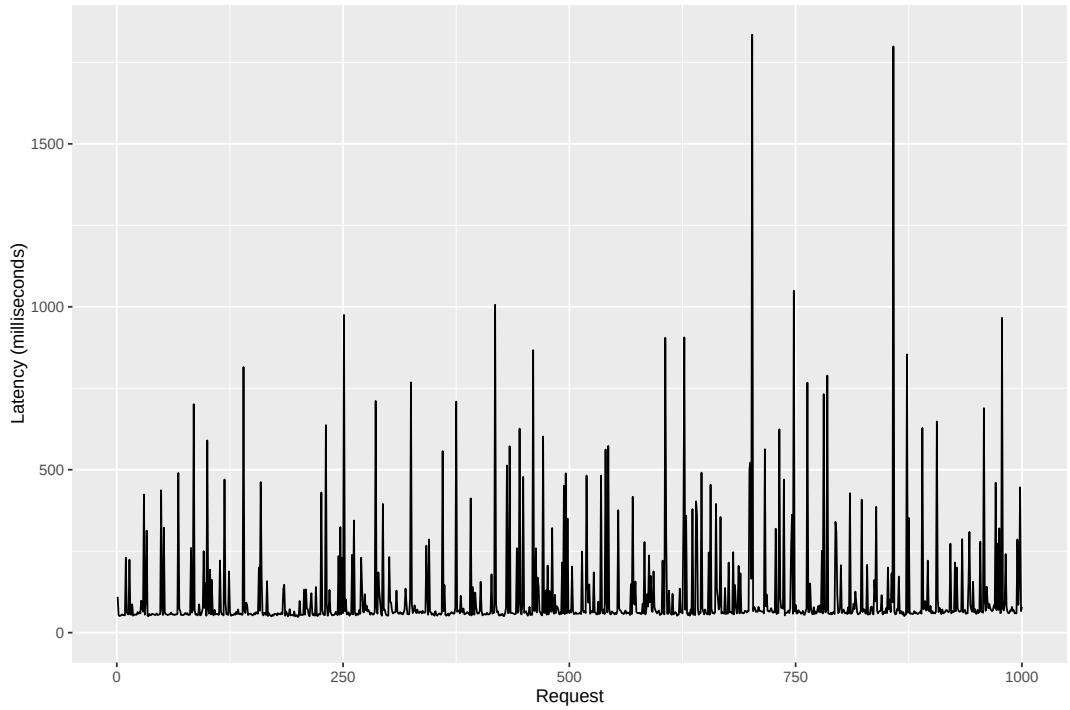


Figure 6.6: API latency on PC

## 6.5. CPU utilization

Python-for-Android supplies a CPython interpreter out of the box. CPython is optimized for single thread performance and compatibility with C extension modules. It is limited by a global interpreter lock (GIL) in multi-threaded use cases with shared memory. Tribler uses C extension modules for crypto tasks, which are CPU intensive. Tribler also uses the event-driven networking engine Twisted, which is written in Python. The core of the event loop within Twisted is the reactor, which runs on a single thread. The reactor provides a threading interface to offload long running tasks, such as IO or CPU intensive tasks to a thread pool. The GIL prohibits more than one thread to execute Python bytecode at a time. This negates all performance gains in terms of parallelism afforded by multi-core CPUs, making Python threads unusable for delegating CPU bound tasks to multiple cores. As shown in the precious section the crypto function took a considerable amount of time to compute. To see if the releasing the GIL as put forward as a solution is feasible we measure if the CPU has more capacity than is being utilized right now. During normal operation equivalent to the profiler measurement, we take a snapshot of the CPU utilization. If there is any performance to gain by releasing the GIL the CPU must be under-utilized right now. A Galaxy S3 smart-phone with Android 6.0.1 Cyanogen mod was used for this measurement. The results show that indeed not al 4 cores of the CPU are utilized by a large margin. Even when performing the intensive crypto work of the Multichain experiment 2 CPU cores appear to be idling. This suggests that releasing the GIL during heavy crypto work could result in a significant performance gain. However our results are inconclusive and warrant further research.

## 6.6. Testing and coverage

The design choice of reusing all Tribler core source code means we need to verify its correctness. To make sure all code on Android works the same as on other supported platforms we need to test all code. Tribler has some unit tests and integration tests that cover a large portion of the code, but not all. The ratio of tested lines of code with respect to the total number of lines of code is the coverage line-rate. We expect to see a line-rate value close to 1, but not 1 since we know the tests do not cover everything. All tests were run two times on the same device with 11 weeks of development in between. The nose module was used for running the tests together with the coverage module for gathering coverage data. The same Nexus 6 smart-phone with Android 6.0.1 Cyanogen mod was used in both runs. The following table shows the results of both executions. The number of tests has increased as well as the coverage line-rate while the number of errors and skipped

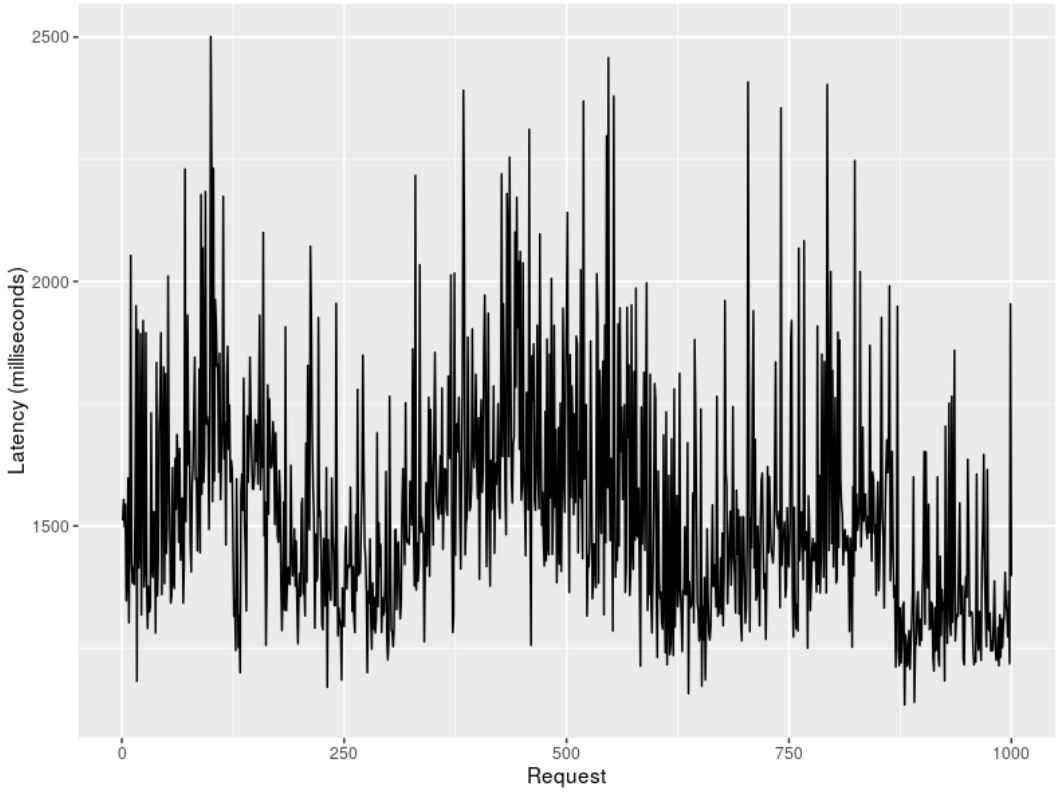


Figure 6.7: API latency on Nexus 6 smart-phone

| Run | Tests | Errors | Failures | Skipped | Line-rate |
|---|---|---|---|---|---|
| 1 | 711 | 14 | 13 | 30 | 0.7241 |
| 2 | 749 | 12 | 15 | 3 | 0.7861 |

Table 6.4: Tests results and coverage line-rate at different points in time during development

tests have decreased. A failure means an assertion was not met and an error represents an exception while running a test. Therefore seeing that the number of failures increased is not that bad since the number of errors decreased. If the line-rate is 1 you still need the the branch-rate to be 1 as well before you can be confident the code will work as expected. The branch-rate is the number of code paths tested with respect to the total number of code paths possible. Unfortunately this metric is not part of the current test plan. Nevertheless the metrics show improvement overall.

## 6.7. Multichain performance

Multichain is the new accounting system of Tribler. This feature is central to the concept of trust in the Tribler network and is very important for the future as other functionality will be built upon it. If mobile devices are to become full-fletched nodes on the network they must support this feature. With Multichain any peer registers the bandwidth it exchanges with other peers. It aggregates these exchanges in blocks and signs them like a receipt and sends that to the other party to sign as well. These blocks are linked in a block-chain to foil attempts of cheating the system. The creation and signing of these blocks is measured to determine if it scales well. Multichain signs a block every 10 minutes, meaning our experiment of generating 25.000 blocks represent about half a year (173,6 days) of continuous effort. The database containing these blocks will grow over time, but should not slow down too much because of it. Measurements were taken on six different devices on multiple moments during development. Figure 6.11 show the performance graphs of every measurement. Clearly visible from the graphs is that Multichain does not scale linearly on any device. They also show that mobile devices are at least a factor of two slower than an ordinary laptop and scale worse. Due to the nature of block-chain every new block needs to contain the hash value of the previous block. If a

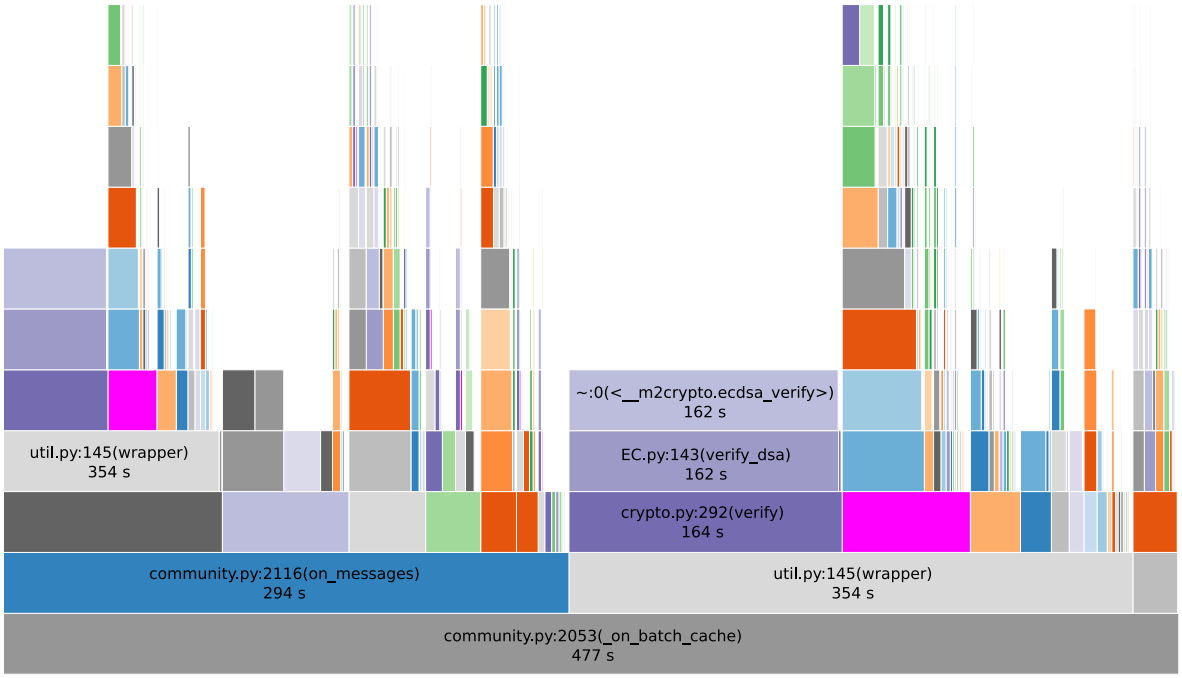


Figure 6.8: Profiler results of 10 minutes Tribler run time (bright pink represents business logic upon receiving a torrent)

database lookup is needed for this and the database is growing, that can explain the non-linear course of the graph. This can be easily resolved by keeping the last hash value for currently connected peers in memory. However this is an indication that creating blocks by the thousands is an IO bound process, rather than CPU bound. Finally, if mobile devices are to be full-fletched nodes on the Tribler network, they should not slow down significantly more than any other ordinary laptop, besides being slower in the first place. Hardware acceleration could close this gap without sacrificing battery life too much. Because mobile devices are a bit behind on the technology curve with respect to desktop computers it is probable that the gap becomes smaller over the coming years. The capacity to store enough Multichain blocks to audit past exchanges should also be on par. If not, other more powerful nodes could be queried to supply the necessary history about a peer, that requests your bandwidth, to verify if that peer is trustworthy.

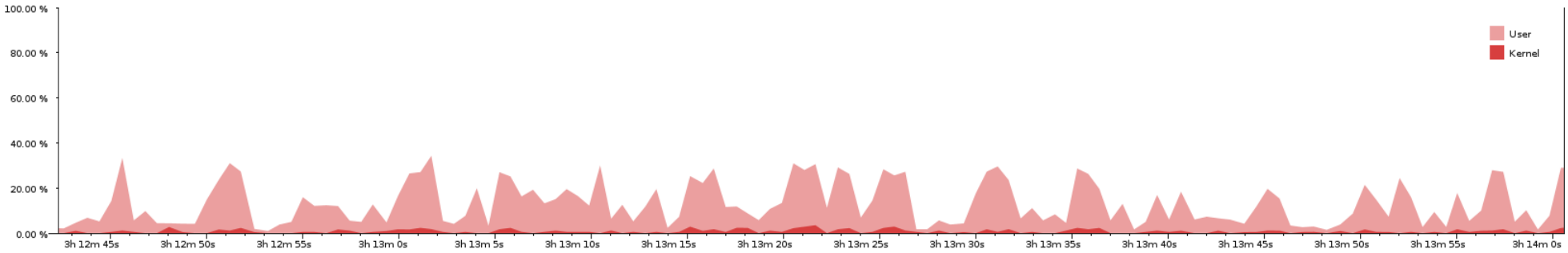


Figure 6.9: CPU usage on a Galaxy S3 smart-phone, about 3 hours into normal operation

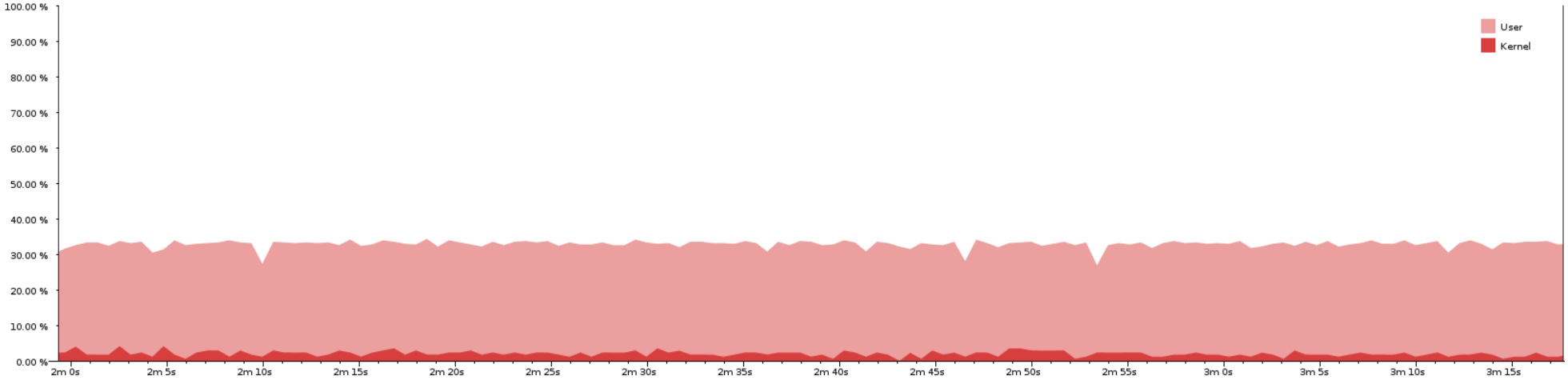


Figure 6.10: CPU usage on a Galaxy S3 smart-phone, about 2 minutes into the Multichain performance measurement

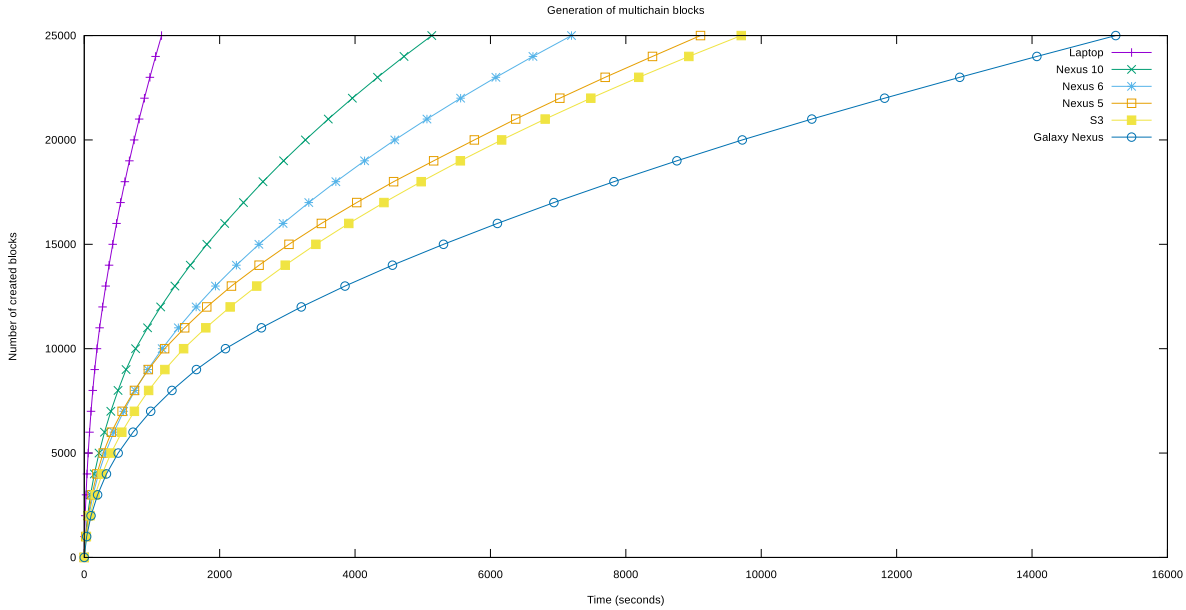


Figure 6.11: Creating and signing of 25.000 blocks between two peers

# 7

# Conclusions and future work

After validating and verifying our design in the previous chapter, we now come to the conclusions with regard to the main research question.

## 7.1. How feasible is it to run all Tribler functionality on mobile devices?

### 7.1.1. Given the constraints of mobile devices, what unique ability can be utilized to extend or enhance Tribler?

### 7.1.2. Using the unique properties of mobile devices, what features can be added or enhanced?

The feasibility in terms of functionality, performance and scalability were measured. Several of these measurements give raise to some minor considerations. The profiling revealed that crypto tasks are a significant part of processing messages. These tasks should be offloaded to a separate computational core to release the global interpreter lock. That would enable Python code to run in parallel, which in turn would improve performance and responsiveness. The CPU utilization measurement showed this approach is likely to succeed. The Multichain measurement showed that database performance in that instance was stringent in the current implementation. It is also the most easy to resolve by keeping the hash of the last block for connected peers in memory . All existing tests can be run on Android just as well as on other platforms. This means, in combination with the modularity of the new design and re-use of the Tribler core, that maintainability and testability are very fit. In terms of user experience the measurement of startup time showed to be very consistent and reasonable. The latency of the API however seemingly showed a curious pattern that cannot be explained right away. It is only known to occur if hundreds of requests per minute are fired at the API, which is unexpected behavior from the user.

Finally, to answer the secondary question: using the unique properties of mobile devices, what features can be added or enhanced? We added the capability to transfer the app to another device that doesn't have it yet via Bluetooth. If NFC is enabled on both devices the app can be transfered by just holding the devices next to each other. Also adding a channel to your favorites can be done this way. Using your real life social network you can build your on-line trusted network this way. And thanks to built-in hardware capability you can setup an ad hoc WiFi network to avoid any infrastructure completely.

What this means in the context of privacy and censorship is that if content is detected by the censor it may have already crossed or will cross the freedom border thanks to the properties of viral spreading.

## 7.2. Future work

This work enables a new direction for future research with Tribler: mobile devices. A new user group, that does not own desktop computers, gains access to Tribler's privacy enhancing functionality.

The contributions of this work enable future research into Tribler's intended use-cases fully geared towards mobile devices. Like for example a large scale experiment with various degrees of a powerful censor or a live deployment of Tribler mobile in areas with restricted Internet access to evaluate the effectiveness.

No routing, no blutooth p2p, mesh networking: that is done by Serval. Out of scope of this work. Add as enhancement later. We won't modify the networking capabilities based on the opportunities because that

would warrant a dedicated study.

shared keychain, credit minging laptop, server, mobile..

NFC is used to share a channel id and to automatically setup a Bluetooth file transfer. In the future the same method can be used to exchange bootstrap peers for the Tribler network. The peer-to-peer Bluetooth file transfer could be made much faster if the WiFi Direct would be used.

Thumbnails retrieval and distribution now that video is so important on mobile, could be added on top of groundwork by me.

credit mining on device A, usage on device B

comparative analysis

Automate ad hoc WiFi direct like nfc-bluetooth!

cutting long running methods in smaller pieces, so multi-threading is smoother (only discovered now, earlier: why is gui hanging?) release GIL in native C code, mainly for heavy crypto tasks streaming API for big responses (dumping all known channels from db)

knippen voor twisted is gewenst, nu pas zichtbaar weakness in Tribler, nuttig om te verbeteren. (conclusion)

iOS with QtPython

Software development / technical aspect only Not policy making, organisational perspective, decision making, normative, ethical, Time limit of 9 months

refereren aan eigen werk in verbergen app en mensen beschermen op bepaalde manier combine with selfcompileapp

# Bibliography


[1] S.M.A. Abbas, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. A gossip-based distributed social networking system. In *Enabling Technologies: Infrastructures for Collaborative Enterprises, 2009. WETICE '09. 18th IEEE International Workshops on*, pages 93–98, June 2009. doi: 10.1109/WETICE.2009.30. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5159221&tag=1.

[2] Roger Dingledine and Nick Mathewson. Design of a blocking-resistant anonymity system. *The Tor Project, Tech. Rep*, 1, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.7565&rep=rep1&type=pdf.

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. 2004. URL http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA465464.

[4] Dr David A. L. Levy, Nic Newman, Dr Richard Fletcher, and Dr Rasmus Kleis Nielsen. Digital news report 2016, 2016. URL http://reutersinstitute.politics.ox.ac.uk/sites/default/files/Digital-News-Report-2016.pdf.

[5] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M. van Steen, and H.J. Sips. Tribler: A social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20:127–138, February 2008. ISSN 1532-0634. URL http://www.pds.ewi.tudelft.nl/pubs/papers/cpe2007.pdf.

[6] Johan A. Pouwelse. The shadow internet: liberation from surveillance, censorship and servers. 2014. URL https://tools.ietf.org/html/draft-pouwelse-perpass-shadow-internet-00.

[7] R. Rahman, D. Hales, M. Meulpolder, V. Heinink, J. Pouwelse, and H. Sips. Robust vote sampling in a p2p media distribution system. In *Proceedings IPDPS 2009 (HotP2P 2009)*. IEEE Computer Society, May 2009. ISBN 978-1-4244-3750-4. URL http://dx.doi.org/10.1109/IPDPS.2009.5160946.

[8] Wendo Sabée, Dirk Schut, and Niels Spruit. Decentralized media streaming on android using tribler. 2014.

[9] Statista. Smartphone sales worldwide 2007-2015, 2016. URL https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[10] C. van Bruggen, N. Feddes, and M. Vermeer. Anonymous hd video streaming for android using tribler, 2015.

[11] M.A. De Vos, R.M. Jagerman, and L.F.D. Versluis. Android tor tribler tunneling (at3). 2014. URL http://repository.tudelft.nl/view/ir/uuid%3Ab258a5e8-002c-4631-9291-04be902119f6/.