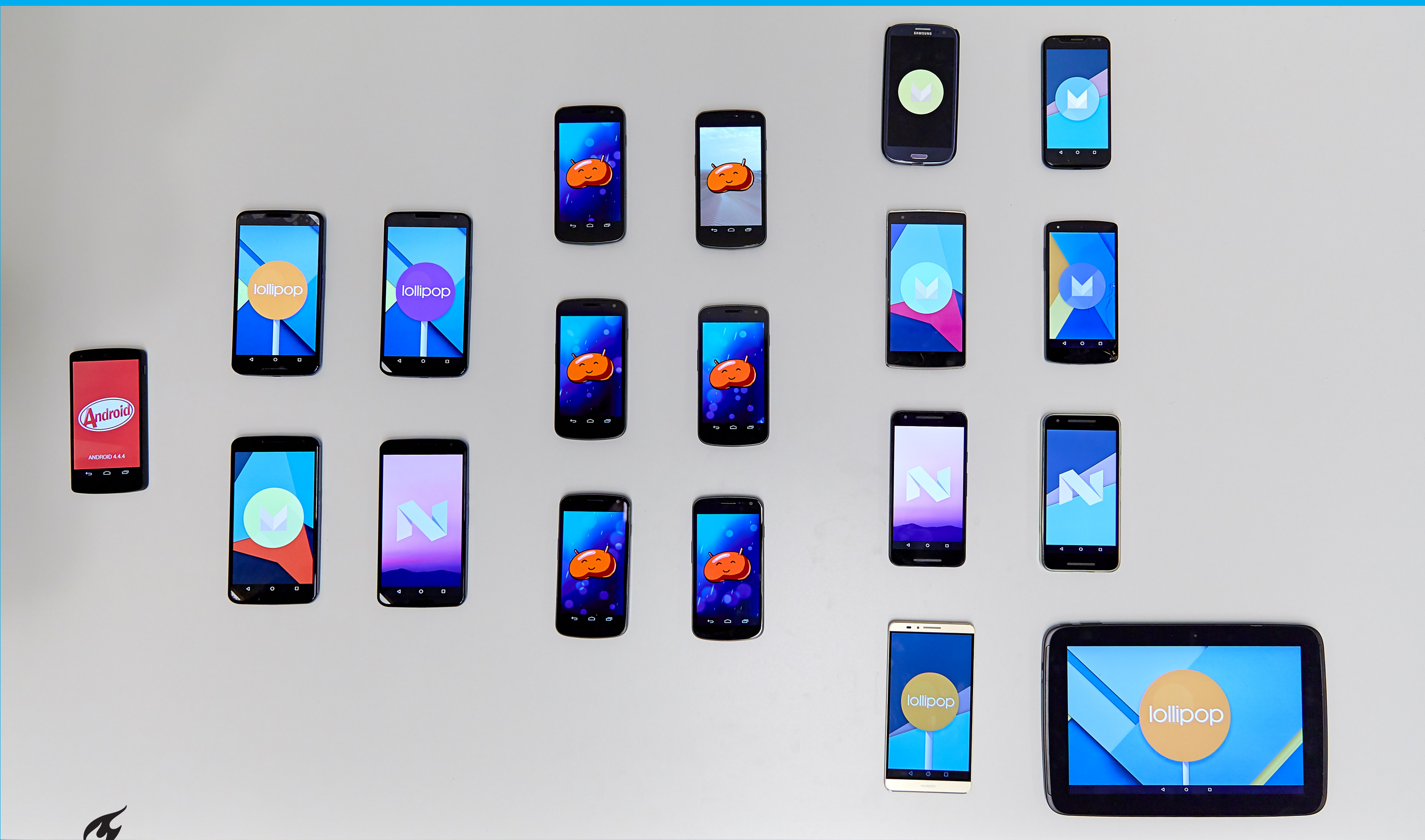
Attack-resilient media using phone-to-phone networking

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 Tuesday December 13, 2016 at 9:00 AM.

Student number:	1308025	
Project duration:	December 1, 2015 – December 13, 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/`.

Contents

1

Introduction

Modern media and news consumption is shifting from traditional outlets to social media and mobile devices. People receive local and global news perceived relevant to their group on their social media feed. The ease of reaching a global audience by an individual with a smartphone diminishes the role of an expert curator handling incoming information. As such, news is not bound to expert opinion or an editorial news desk anymore. According to Reuters Institute's Digital News Report 2016, young people in particular are shifting to social media as their number one source [24], shown in the age distribution graph, Figure 1.1.

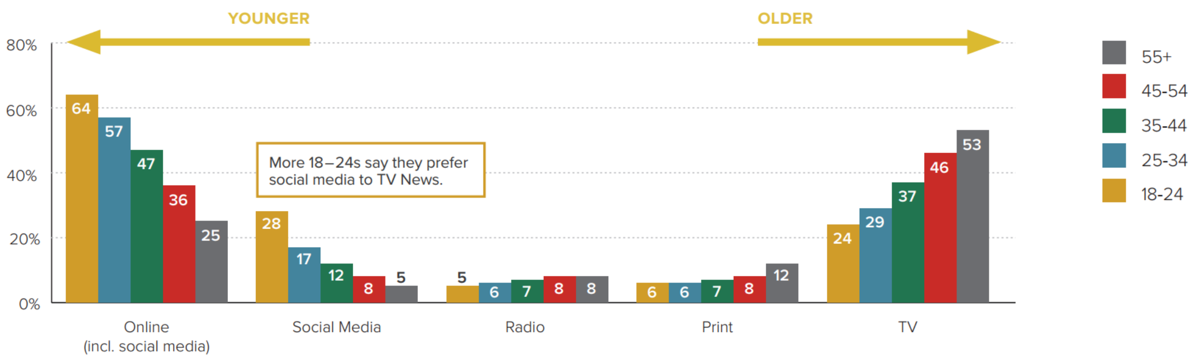


Figure 1.1: Main news sources split by age [24]

The mobile devices on which news and modern media are consumed are also capable of news production. The capabilities and versatility of smartphones in particular enable them to be used for both production and consumption. Most smartphones have one or more cameras to record multi-media content that can be shared immediately from the device. A smartphone also has the unique property of being a ubiquitous device that is highly mobile and extremely connectable. Figure 1.2 shows that, world-wide, 1.4 billion smartphones were sold to end-users last year, and shows a considerable growth in the past five years.

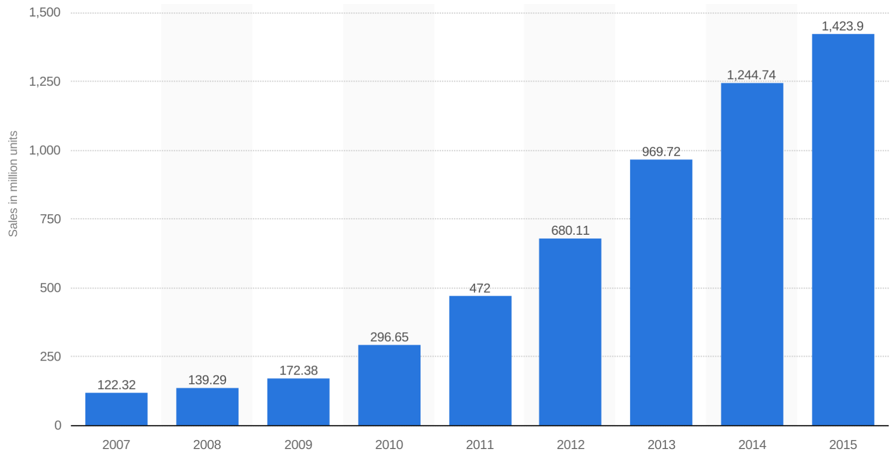


Figure 1.2: Number of smartphones sold to end users worldwide from 2007 to 2015 (in million units) [43]

Eye-witnesses often have smartphones 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. The users themselves are turning from con-sumers into pro-sumers [42].

Unfortunately, the Internet is censored in some parts of the world, limiting access to news and global opinion online. Invasion of privacy by large scale monitoring is grave cause for concern [17]. However, censorship and large scale monitoring on the Internet are problems that can be battled with decentralized solutions [32].

In crisis situations, like natural disaster or unrest, people need to communicate and coordinate their efforts to restore safety. In this context the smartphone becomes particularly important because it is often carried on person and provides connectability. In the wake of recent calamities, people could mark themselves as safe on social media [13], 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 [9]. Therefore we require a distributed solution using smartphones, that does not require infrastructure.

Tribler is a fully decentralized video-on-demand system. [31, 41, 47] It is autonomous, attack-resilient and self-organizing. [1, 34] 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.[11, 12, 39, 50]

The autonomous, attack-resilient and self-organizing properties of Tribler make it an interesting candidate for handling use scenarios of rapid media dissemination without dependency on centralized infrastructure, and with preservation of anonymity. However, so far Tribler only supports desktop and server versions of Linux, Mac and Windows. In order to realize the mobile media dissemination use cases as mentioned above, it will be necessary to enable Tribler on mobile devices, which however may be resource limited.

In this thesis, the first prototype is presented that has all Tribler functionality fully enabled on mobile devices. The two following research questions are answered in this work:

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

2. Given the constraints and unique abilities of mobile devices, what functionality of Tribler can be added or enhanced?

A prototype is built for Android OS, since Android currently dominates the smartphone market. Its modular architecture is specifically designed for portability and maintainability.

The remainder of this thesis is organized as follows. The problem of media dissemination under adversary conditions is described by Chapter 2. Tribler's functionality is described in Chapter 3, as well as a short discussion of the specific opportunities and challenges that mobile devices bring with them. These lead to the requirements, which are listed in Chapter 4, followed by a fitting system architecture design. An implementation for Android, that satisfies the design requirements, is presented in Chapter 5. This implementation is then used to analyze the performance of Tribler on smartphones and tablets, and resulting performance measurements are presented and discussed in Chapter 6. Finally, the research questions are answered based on the results of the experiments in Chapter 7, and suggestions are made for future research, based on this work.

2
Problem description

While mobile devices have become ubiquitous and powerful for spreading digital information quickly, present-day commercial services and centralized infrastructure pose risks with regard to freedom and privacy. In this chapter, these risks are discussed in more detail, we explain how fully decentralized distributed solutions can help to increase resilience, and we present the contributions of this thesis.

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 [17]. 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. Even more risk for violation of privacy, comes from integrating social media into regular websites, to de-anonymize and track the whereabouts of users even outside of the realm of social media. Whenever users lose control over their privacy it becomes a serious problem.

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).

Cuba offline Internet [52]. For example: Arab Spring, kill switches are real [36], within half an hour all networks were unreachable. Syrian offline [37]. China: Apple's news app deactivated [45], last year. Censorship in China [51]. BBC: China and Iran [23]. Censorship in Iran [21]. Turkey social media block [46].

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.

2.2. 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 [33] 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 smartphones 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 smartphones. The adversary cannot compromise smartphones or other participating devices.

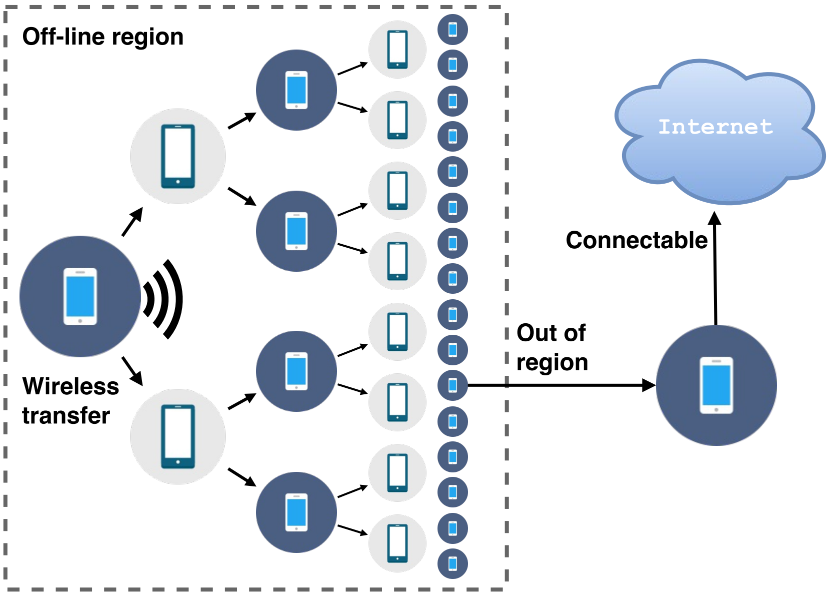


Figure 2.1: Viral spreading from one device to another within an off-line region

2.3. Distributed solutions

With distributed solutions, a scenario like in Figure 2.1 would become possible. Mobile devices can freely move around off-grid, and eventually one or more devices will move out of the off-line region, also known as "the freedom border".

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. Finally, all users must be able to share, order and appreciate information of other users, 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. Fully distributed systems capture the characteristics just mentioned. 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 smartphones 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.

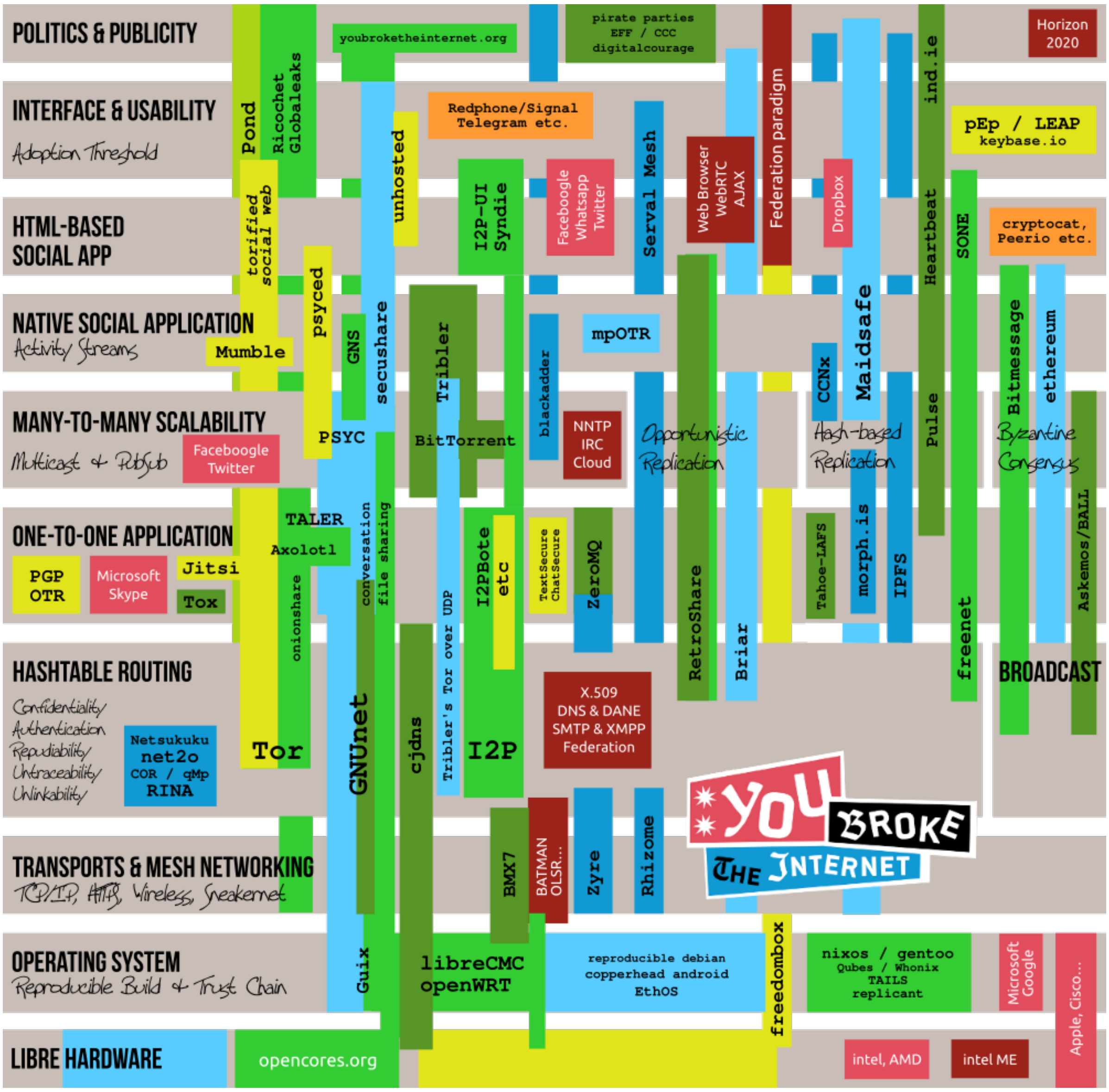


Figure 2.2: Various partial solutions to mend the broken Internet according to www.youbroketheinternet.org

The following network properties are defined [22] to provide a technical solution that enables free expression in the face of an adversary as defined in Section 2.2:

- *Resilient against communications blackouts.* Should be challenging for any entity to disable.
- *Resistant to monitoring and tracking of users.* Both who is using the network and any sensitive messages they send should be secret.
- *Able to be built from innocuous components.* Should only require readily available hardware, and the possession and use of required hardware should not be illegal or suspicious.
- *Able to run at meaningful scales.* Should be more effective at disseminating information than people with megaphones; more broadly, given a level of service, should be able to run at non-trivial scales.

Mobile devices are essential to take the next step in the face of censorship. If we can use smartphones to create a fully decentralized social network, like in Figure 2.1, we can go further and say: Even if you can take down the Internet, the only way to take Tribler down is to take everybody's smartphones 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.

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. Smartphones 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 [32] Decentralized solutions can work in these situations. Censorship and large scale monitoring is difficult in decentralized networks.

Various initiatives have been started to deal with one or both of these problems [35].

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. [4] 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.

2.4. Contributions

The main contribution of this thesis is making Tribler available on mobile devices and the utilization of their unique properties in the context of censorship and communication during crises. This work also enables a new direction for future research with Tribler: mobile devices. The resilience of Tribler against attacks on the Internet is greatly increased. The goal of Tribler is to become an information sharing platform that protects the privacy of its users and is resilient to attacks while not relying on existing infrastructure.

Previous attempts did not to deliver all functionality [41, 47, 50]. Maintainability issues with earlier designs, and large amounts of technical debt [10], were the major causes. We changed the architecture of Tribler for our approach.

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

3 Tribler functionality

Tribler is a fully decentralized video-on-demand system. [31, 41, 47] It is autonomous, attack-resilient and self-organizing. [1, 34] 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 [11, 12, 39, 50].

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 [15]. Video-on-demand means that users can search for desired videos and simply click to play videos in a streaming fashion, so without waiting for the entire video to be present on the device. Users can search from within the application [54] and browse for videos in channels rated automatically by popularity. Recommendations are given based on the user's preferences and those of users with a large shared interest [7, 53]. 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. Tribler can use the distributed hash table (DHT) of a BitTorrent swarm for a specific torrent to discover peers and gather meta-information, as well as enhanced peer exchange protocols [38, 49]. 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 connectable 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. Trust

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. 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 blockchain to foil attempts of cheating the system. This is a significant improvement over previous attempts to prevent free-riders [25–27, 29].

Multichain is explicitly designed to not use a global state. Syncing a global state is less scalable in terms of storage and network bandwidth. The Bitcoin blockchain, for example, does have a global state and partly because of that the average transaction confirmation time is in the tens of minutes [40].

3.6. Anonymity

Tribler can protect the privacy of users by hiding their identity [30, 44, 47]. To connect to others on the network, anonymous connections are created on behalf of downloading peers (leechers) and uploading peers (seeders). 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 hidden seeding protocol, modeled after the hidden services of TOR, allows for anonymous content sharing via said TOR-like onion routing [39]. 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 blockchain technology with the purpose of building trust on-line and creating the Internet of Money.

3.7. Towards Tribler on mobile devices

In the previous sections we discussed the features and applications of Tribler. 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. All positive and negative properties that come from these features are transferred to mobile devices, which will add their own distinct properties to the mix. Bringing Tribler to mobile devices will give potentially millions of users access to these features on the move. Expanding the Tribler network with mobile devices could also benefit the research that can be performed on the live network.

3.7.1. Opportunities

What does mobile allow in addition to desktop? Mobile devices are inherently easy to move around and very portable. Since there are so many smartphones with builtin cameras around that may be brought in almost everywhere, eyewitnesses have the opportunity to record videos, as the story unfolds [42], and share those news stories instantly with the world, since the smartphone is very connectable. In case of a breakdown in communication infrastructure, 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. Tribler can exchange channel ID's to subscribe to another Tribler channel peer-to-peer also via NFC. Thanks to the innocuous nature of a smartphone and its connectability, it can be used as a component to build an alternative network out of components that are not illegal in themselves [22].

3.7.2. Challenges

What does working on mobile phone mean? The portability of mobile devices requires any network interface and power supply to be wireless. Mobile devices are typically equipped with batteries to operate without a power cord. Considering the size and weight, capacity is limited. Smartphone batteries usually barely hold a charge that can sustain a day of heavy use. Tribler could potentially drain the battery faster. Heavy encrypted network traffic does 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 cryptographic work. Because Multichain punishes cheating with a permanent ban, it must never lose 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.

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 system architecture and the reusable components of the current Tribler application. Some functionality of Tribler has been shown to work on Android before [41, 47, 50]. Our design will make Tribler fully work on mobile.

4.1. Functional requirements

Following from the problem description and Tribler functionality described in Chapter 3, we define the following functional requirements:

A1. The implementation must be capable of effortless peer-to-peer transfer of itself.

A2. The implementation must be capable of publishing videos to other devices without the need for an Internet connection.

A3. Any video available on the device must be directly publishable.

A4. The implementation must enable a user to record videos.

A5. The implementation must enable a user to create a channel.

A6. The implementation must enable the owner of an existing channel to edit it.

A7. The implementation must enable creating a torrent file from a file available on the device.

A8. The implementation must support streaming video playback.

A9. The implementation must support all other Tribler functionality not mentioned above.

4.2. Non-functional requirements

Following from the challenges and opportunities, as described in Section 3.7, we define the following non-functional requirements:

B1. The mobile device must be capable of running Tribler independently.

B2. The implementation must incorporate the existing Tribler Python core and the required C/C++ libraries.

B3. The implementation must consider the restricted resources of a mobile platform in terms of RAM and processing power.

B4. The implementation must support WiFi, Bluetooth and NFC peer-to-peer features.

B5. The implementation must utilize the built-in camera for recording videos.

B6. The mobile device must be connectable via WiFi or mobile data connection

B7. The implementation must be distributed as a single installable container.

B8. The implementation must be able to keep running in the background even if the user is not actively using it.

B9. All processing tasks must be performed asynchronously.

B10. The user must be able to interact with the implementation via a graphical user interface (GUI).

B11. All ongoing tasks must be indicated as such in the GUI.

B12. The GUI must stay responsive to the users' input while performing a background task.

B13. The GUI must stay responsive to the users' input while presenting large amounts of data on screen.

B14. If invalid input is provided by the user through the GUI the user must be asked to correct the input.

B15. If a recoverable error occurs, the implementation must automatically retry.

B16. If an exception occurs, the user must be able to restart Tribler.

B17. Upon restarting, the implementation must return to a working state.

B18. The entire build tool-chain must be integrated on the build server of Tribler.

B19. As much code as possible must be covered by tests.

B20. The implementation must be agnostic to version differences of supported platforms and operating systems.

B21. The user interface design must follow established best practices.

B22. The implementation must be attack-resilient.

4.3. System architecture

The requirements dictate how the system architecture of Tribler on mobile devices will take shape. The proposed architecture as shown in Figure 4.1 is specifically designed for portability and maintainability, and clearly separates components with a distinct responsibility.

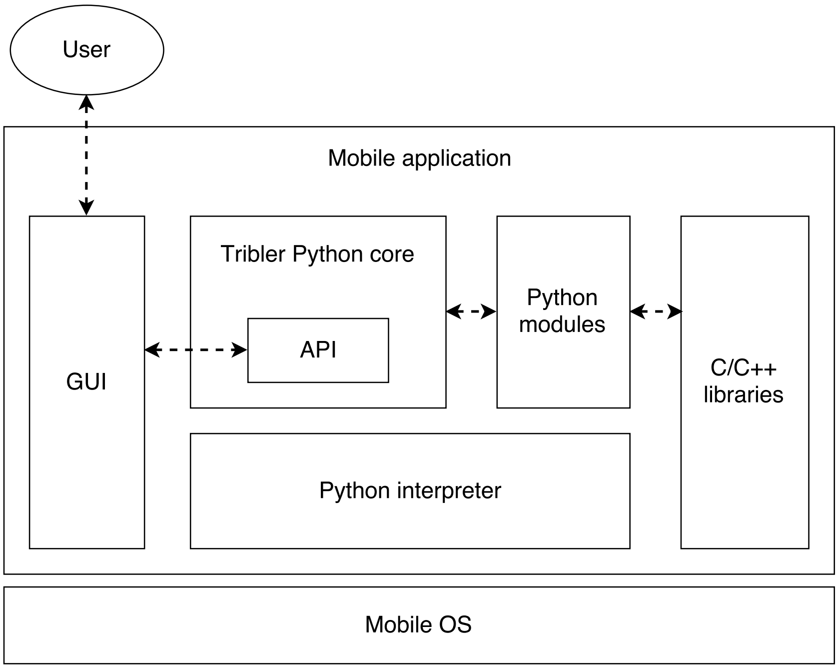


Figure 4.1: System architecture design

The requirement (B2) of re-using the Tribler Python core and its C/C++ dependencies requires the architecture to incorporate these as is. Mobile platforms are not required to support Python code natively and the implementation must be distributed as a single installable package and run independently (req. B1). As a consequence, a Python interpreter must be incorporated into the implementation. The user must interact with the implementation via a GUI, as stated by requirement B10. As such a GUI is not part of the Tribler core, it has to be included separately in the design. A separate GUI can be made and optimized for any specific platform and target device. For instance, a design can be made for large surface displays and another one for small touch-screens, like a smartphone. This leads to the design choice of creating an API, that allows for a common interface across all platforms, to let the GUI communicate with the existing Tribler core. The API will yield a more maintainable solution than previous attempts to bring Tribler to mobile devices [41, 47, 50] that did not include such an API.

5 Implementation

In Chapter 4 we proposed a generic design to enable Tribler on mobile devices. Many types of connectible mobile devices exist. In the context of the problem description from Chapter 2, devices used for human communication, such as smartphones, will be our prime focus. The largest potential user base in the smartphone market can be reached by targeting Android OS [16]. Therefore, our implementation will target Android OS.

Following the choice for Android, the generic design in Figure 4.1 can be further specified as presented in Figure 5.1. Two main additions are a separate video player and a Java native interface (JNI) component. In the following sections, each component from Figure 5.1 will be described in more detail. The complete build toolchain is presented in Section 5.5, which combines everything to build the final Android application package (APK). Finally, the statistics of the implementation are presented in Section 5.6.

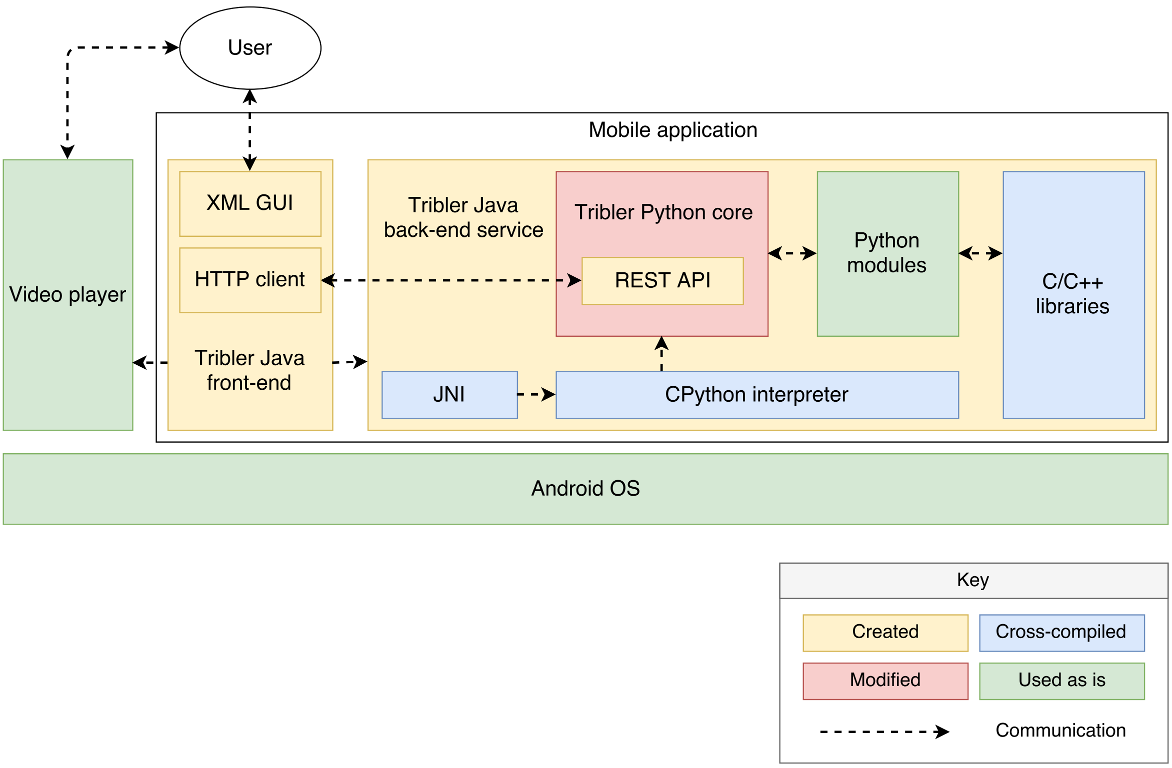


Figure 5.1: Implemented system architecture

5.1. Android OS

Android is an operating system, based on the Linux kernel, that runs on smartphones, tables, wearables and smart-TVs. It provides a Java Virtual Machine (VM) and a Java application framework API.

Since API version 14, near field communication (NFC) and Wi-Fi peer-to-peer (P2P) connections between compatible devices is supported. And since API version 16, a NFC push message can be used to start large file transfers over Bluetooth (req. A1). This can be used in the context of the problem description, for Tribler needs to be able to spread wireless from phone to phone. Using this NFC push message functionality, the transfer of the APK file is fully automated (req. B4). Figure 5.3b shows the instructions for the user of this effortless transfer: just hold the phones back to back. Bluetooth, rather than Wi-Fi, is the technology of choice here, because the former has a standard file transfer protocol, built into Android OS, and the latter does not. This means there are no prerequisites on the receiving device for receiving and installing Tribler from a nearby phone via NFC+Bluetooth. If NFC is not available, all other options to transfer the APK are presented to the user to choose from instead. If NFC is available, but not enabled on the device, the implementation will prompt the user to do so. In that case, the other options are accessible via a button with the Bluetooth icon on the action bar. NFC is used to enable easy sharing of channels as well, for example your own channel or your favorites. After receiving the NFC push message, Tribler is automatically started and asks the user if the received channel must be added to their favorites.

Our implementation supports API version 18 and higher, because of reasons explained in Section 5.2.5. 85.6% of Android devices run API version 18 or higher [19]. Android support libraries are used to abstract from differences between API versions (req. B20).

Inter-application communication, via Android intents, must be secured (req. B22). An Intent is a messaging object to request an action from another app component [20]. Therefore, all Android intents are explicit for internal actions and the action of all received intents is checked, especially broadcast intents [6]. Also, only the activities and services that should be publicly accessible are exported in the application manifest.

5.2. Tribler Java back-end service

Our implementation uses Java to build upon the Android Java API. To run an application in the background, even when the screen is turned off, it must be run as a service. Therefore, every component, except the GUI, is part of the back-end service (req. B8). The service is started as a separate process by the Java front-end (req. B12). This way the user can be presented with the GUI, that indicates the service is loading in the background (req. B10, B11).

5.2.1. Tribler Python core

Tribler is written entirely in Python, but Android does not support this natively. Python is an interpreted language. Because of our design requirements (B1, B2), we incorporate a Python interpreter into our design. On top of the entire core of Tribler can run, containing a REST HTTP API module also written in Python. All communication with the front-end is done asynchronously via a REST API (req. B9). Finally, the REST API communicates with an HTTP Client on the user interface side via JSON.

5.2.2. JNI

Using Java requires the use of the Java native interface (JNI) to communicate with the C/C++ components. JNI enables functions that are written in C to be callable from Java and vice-versa. The Python interpreter is started by the Java service using JNI, after loading the necessary C/C++ libraries.

5.2.3. CPython interpreter

Tribler uses C++ implementation of torrent protocol, called libtorrent. To use this C/C++ library from within Python a certain capable interpreter is required. P4A offers such an interpreter: CPython.

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).

5.2.4. Python modules

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 5.2 shows the dependency tree.

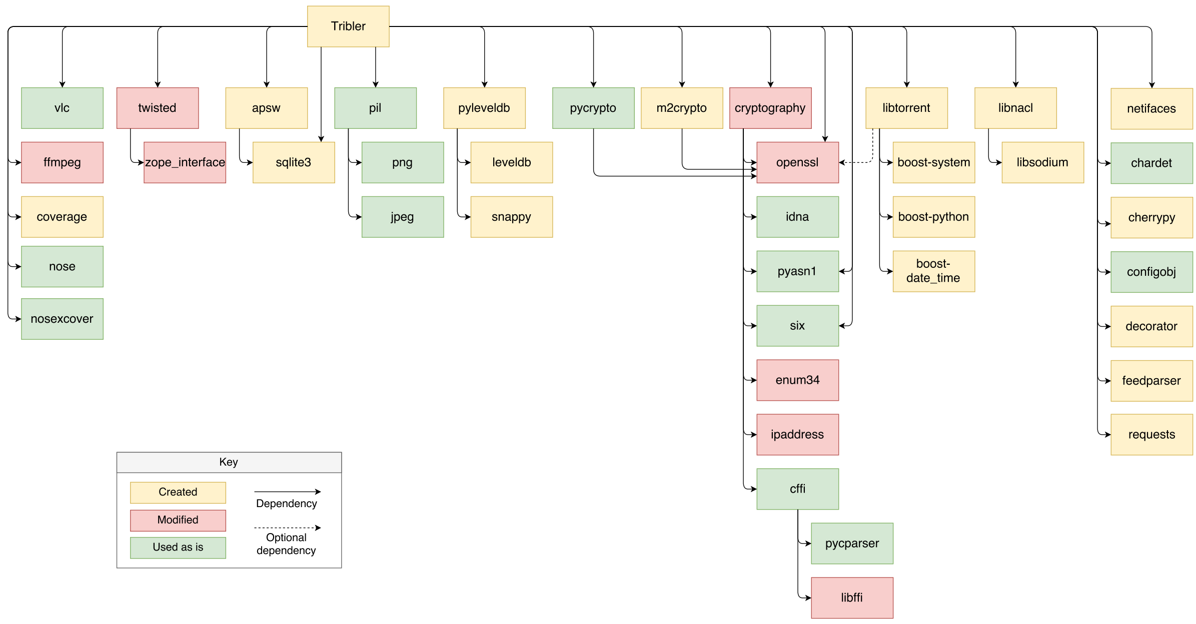


Figure 5.2: Tribler dependencies in terms of Python-for-Android recipes

5.2.5. C/C++ libraries

The standard C library of Android differs from the GNU C Library (glibc), which makes it not trivial to port Linux libraries to Android. All C/C++ dependencies of Tribler were therefore linked against glibc and glibc is included as a shared library (req. B1, B2). Static linking could result in unexpected behavior if more than one library is linked [18], and Tribler uses many, as shown in Figure 5.2. Since Android 4.3 (API version 18) shared libraries do not have to be loaded in order manually anymore. Therefore,

The Python code loads other Python and native modules directly.

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.

5.3. Tribler Java front-end

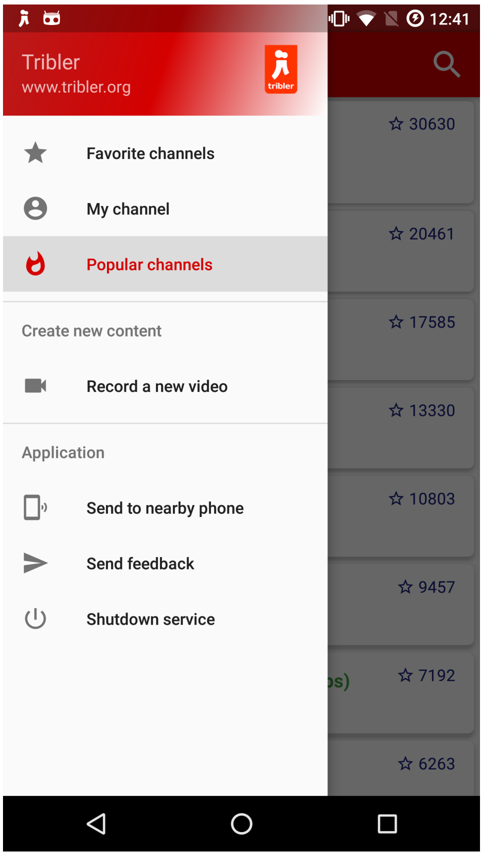
The requirements on asynchronous communication (B9) and responsiveness (B12) require the decoupling of the GUI from the back-end.

The GUI is created by a native Android Java application, which talks to the REST API module. 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).

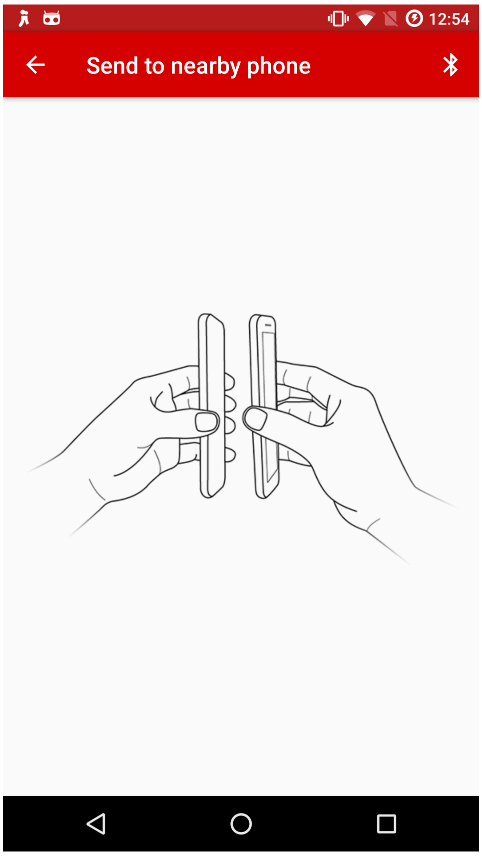
Having two separate Python interpreters in distinct processes talking to each other, because that means using a very resource heavy Python GUI instead of the regular and lightweight native Android Java XML GUI. The latter has tools available for automated UI testing.

Figure 5.3a shows the main menu of the Tribler app. The first three items presented to the user are the main views: the user's favorite channels, their own channel and discovering popular content. The next four items are actions related to creating new content or related to the application. It is organized this way to benefit from the simplicity of a flat menu while the actions are grouped by context with a header for clarity.

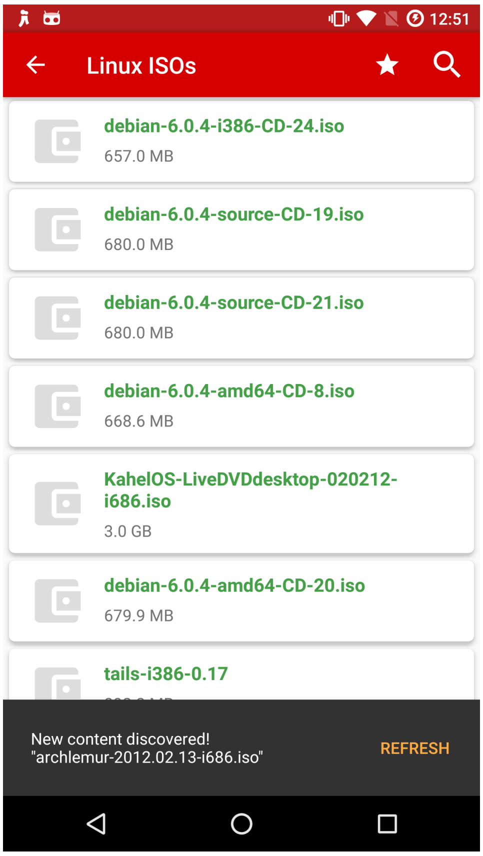
Users can browse through a list of popular channels or their own favorites. Each channel has an indicator



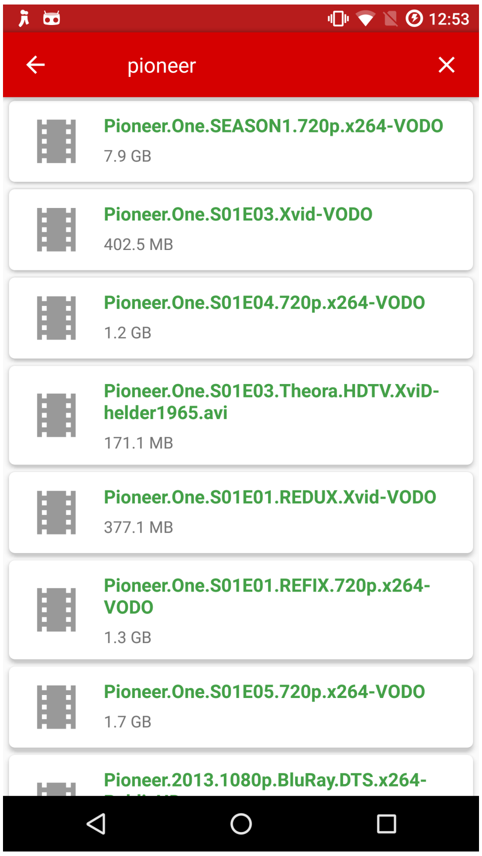
(a) Navigation menu of the Tribler app



(b) NFC+Bluetooth transfer of app or channel



(c) Channel with newly discovered content



(d) Search results showing video content

Figure 5.3: Screenshots of the Java front-end

of the amount of users that have added that channel to their list of favorites, as can be seen behind the menu in Figure 5.3a. Channels contain multi-media content added by their respective channel owner or everyone, depending on the security policy of that channel. Tribler allows three settings: open, semi-open and closed.

The favorite channels and popular channels views both show a list of channels contain content

Search is accessible from any channel view. Upon typing keywords into the search bar debounce

5.3.1. HTTP Client

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.

5.3.2. XML GUI

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.

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.

5.4. Video player

To support streaming playback of videos (req. A8) a capable video player is required. The video player VLC is integrated in the desktop version of Tribler. However, such a library, and integration of a custom GUI, is hard to maintain. VLC is offered as a standalone Android application package (APK) from their website [48]. Therefore, rather than implementing our own GUI, this APK is embedded as a whole inside Tribler's APK as an asset (req. B7). If VLC is not yet installed on the device, the user is prompted to do so and offered the version from inside Tribler. This allows the user to install VLC without further requirements.

5.5. 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.

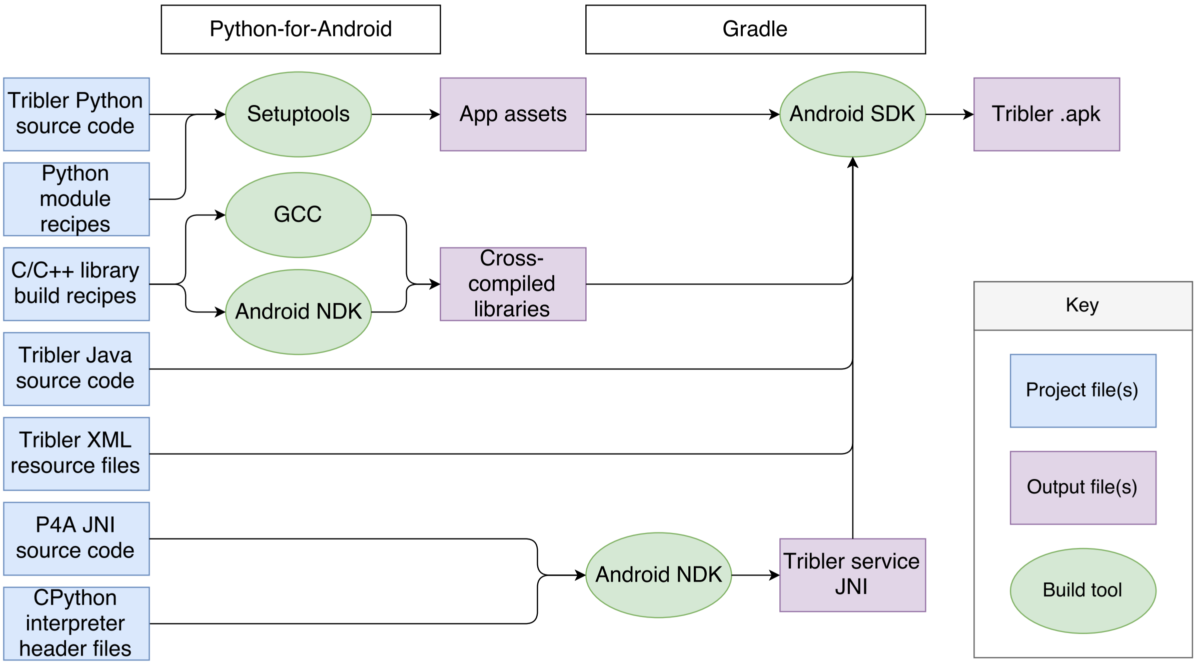


Figure 5.4: High level overview of the build tool-chain

The libraries are compiled for any ARMv7 compatible platform, and little effort is required to replace Android with another OS in this respect.

5.6. Implementation statistics

All requirements have been met. 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.

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

Performance analysis

To analyze how feasible is it to run all Tribler functionality on mobile devices, we measure several performance characteristics relevant to the functional and non-functional requirements in Chapter 4. We take several measurements on different devices to quantify the performance and resource usage in the context of the scenario in the problem description in Chapter 2. The results will indicate the state of the art, before any optimization, in functionality of Tribler on mobile devices as described in Chapter 3. From the results, possible angles for optimization will emerge and further described in Chapter 7.

6.1. Content discovery

Before anyone can view new content that has been added to a channel, it needs to be discovered by other devices. We measure the amount of time it takes for other devices to discover new content in a channel they are subscribed to, starting from the moment it is added to that channel. Depending on the random walk in the channel's community content can be discovered either very quickly or after a while due to eventual consistency.

Figure 6.1: Experimental setup with various smartphones and one tablet, all showing the About Android screen

Device	Nexus 5	Nexus 6	Galaxy Nexus	Galaxy S3	OnePlus One	Nexus 10	Total
Amount	1	4	6	1	1	1	14

Table 6.1: Devices used in the content discovery experiment

Figure 6.1 shows the experimental setup with various smartphones and one tablet. Each device is connected to the same wireless network and within 1 to 2 meters distance from the same access point. Different versions of Android OS are installed, ranging from 4.3 to 7.1, and some run a CyanogenMod. Each device is installed with the same version of Tribler and the same database, containing up to date information about existing channels and their content. This database was gathered in the days before this experiment, serving as a hot cache. On one device, a Nexus 6, from now on referred to as the source, a new channel is created to which the other 13 devices subscribe via NFC. Then, repeatedly a new video is recorded and added to that channel by the source. The new videos are discovered, and the event logged, by each device individually.

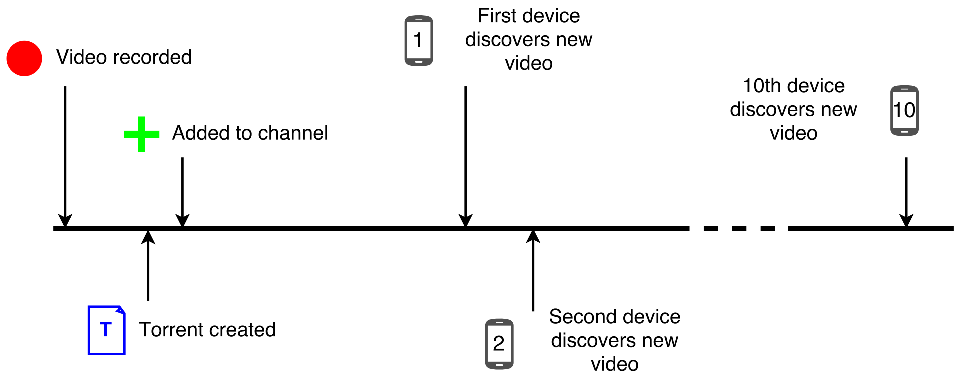


Figure 6.2: Sequence of events when a video is recorded and distributed

The sequence, as shown in Figure 6.2, of recording a new video, adding it the the same channel, and letting the subscribers discover it, is repeated 15 times for accuracy. All devices are synced with NTP to be able to have a common timeline for this experiment.

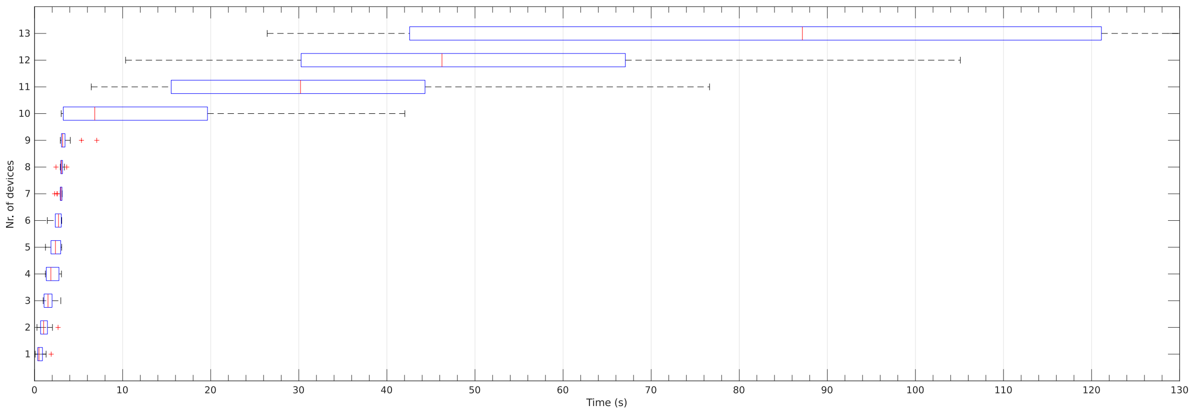


Figure 6.3: Elapsed time after adding new content to a channel and it being discovered by subscribed peers

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 the first device discovers new content in less than 2 seconds, while 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 a time. Many 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 onward the dissemination slows down.

6.2. Multichain performance

If mobile devices are to become full-fledged nodes on the network they must support the Multichain feature. The creation and signing of these blocks is measured to determine whether it scales on mobile devices. 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. A laptop is included to give some more perspective. Its specifications are listed in Table 6.2. Figure 6.4 shows the performance graphs of every measurement.

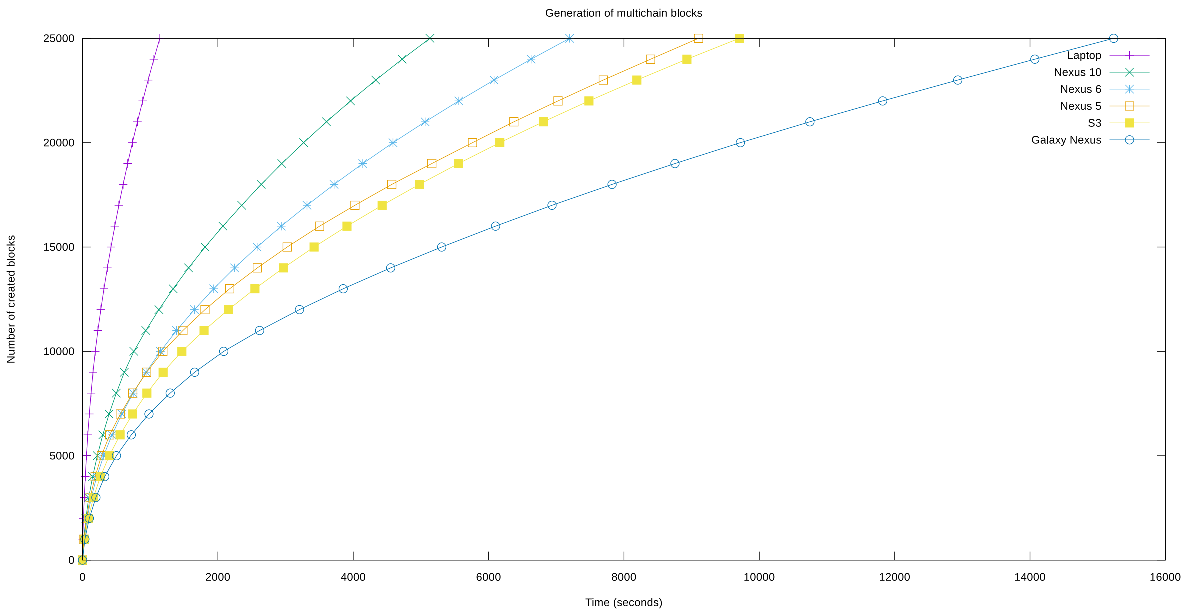


Figure 6.4: Creating and signing of 25,000 blocks between two peers

Laptop	Dell Latitude E6520
CPU	i7-2760QM @ 2.4GHz
RAM	8 GB @ 1333 MHz
SSD	80GB X25-M

Table 6.2: Hardware specifications of the laptop used in the Multichain and API measurements

Clearly visible from the graphs is that Multichain does not scale linearly on any device. Mobile devices are at least a factor of two slower than an ordinary laptop and scale worse. Due to the nature of blockchain, every new block needs to contain the hash value of the previous block. If a 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 optimized 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-fledged 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, the gap may become 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.

6.3. Startup time

Key to user retention is a fast startup time. The app starts the GUI first, followed by the service. Loading of the GUI is fast, because it is in fact a thin shell, thanks to the separation of the back-end and the front-end, and the asynchronous implementation. This enables the GUI to be visible and responsive to user input, as shown in Figure 5.3a, while the service may continue loading in the background. However, before any task can be executed by the service, it needs to be fully started. To measure the total startup time, we register the time of launching the app and the moment the Tribler-started-event is registered by the GUI. This event is sent over the API event-stream and indicates that the service is fully started and ready to accept all incoming requests. We expect consistent loading times on each device, and potentially significant different loading times between devices, because of differences in hardware. Therefore, we measure the startup time 10 times on 5 different devices. The app is launched with Android Debug Bridge (ADB) from a laptop and the Tribler-started-event is read directly from the device's log over ADB and timed on the same laptop, so they use the same clock.

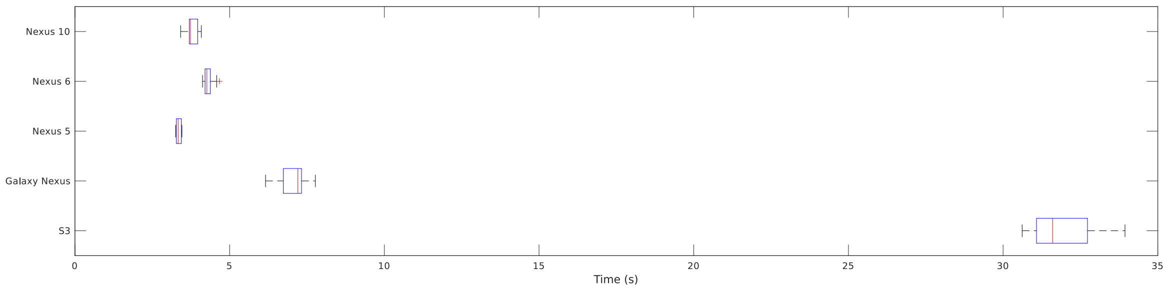


Figure 6.5: Startup time per device for 10 consecutive runs

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 6.3: Statistics of startup time per device

Table 6.3 shows the statistics per device. The results show a very small sample standard deviation and a very low startup time. The S3 is performing way worse than may be expected from comparing the results of the other devices and the Multichain experiment. The reason for that may be that this phone was not wiped and given a fresh install of Android before starting the experiment like the other devices were. Which would mean that other applications installed on a device could significantly impact the startup performance of Tribler. This should be investigated further, including if anything can be done on the part of Tribler. The sample standard deviation is relatively small for all devices, which indicates that the startup time of Tribler is consistent.

6.4. Content creation

New content can be generated with a smartphone, like for example a video that has been recorded with the built-in camera. How quick one can create content and distribute it, depends not only on the discovery time, as measured in Section 6.1, but first, and perhaps foremost, on the speed of the torrent creation process. In this experiment we measure the time required to create a torrent file for different sizes of videos. We also measure the amount of time it takes to add that torrent to a channel. In the torrent creation process, a hash is calculated for each piece of the content. Therefore, the amount of time it takes for a torrent to be created is relative to the size of the content. Because this is a CPU intensive task, we expect the time required to create a torrent file to follow the time complexity of the hash function. The setup for this measurement is exactly the same as in Section 6.1, but we only look at the source, creating and adding the torrent of the video content.

Figure 6.6 shows the relation between the size of the content and the time required to create a torrent file for it.

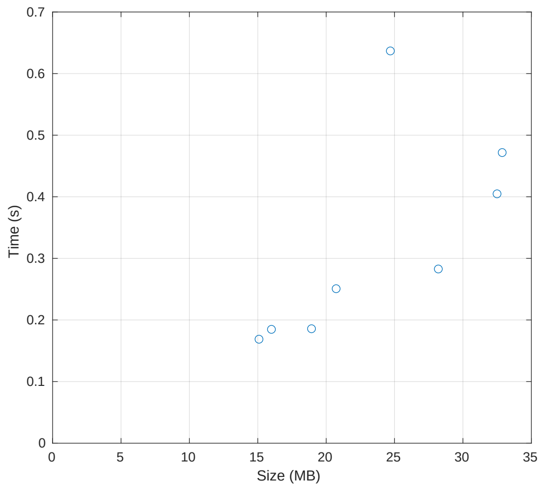


Figure 6.6: Creating a torrent for content of varying size

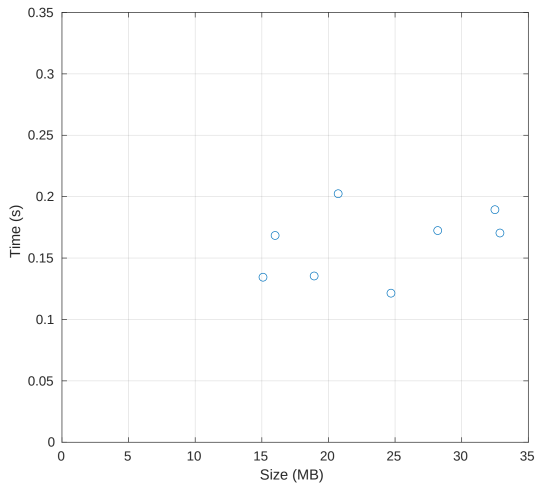


Figure 6.7: Adding a torrent to a channel for content of varying size

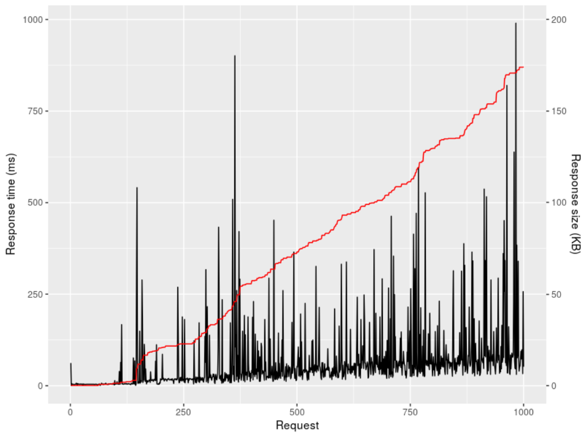
The results suggest a linear correlation between the size of a video and the torrent creation time, with one outlier in this set of eight samples. However, due to differences in hardware acceleration of the hashing algorithm implementation and small amount of data, no claims can be made in general. Finally, to add content to a channel, only metadata is required, which does not scale with content size, as can be seen in Figure 6.7. However, the margin of error is also determined by the API response time, which we will measure in the next section.

6.5. API responsiveness

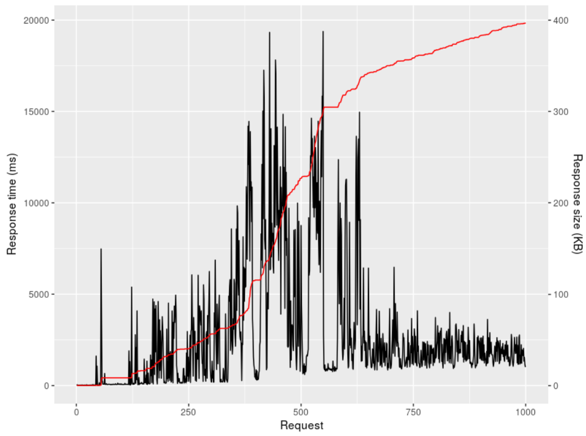
By design, most functionality is operated through the API. Therefore, measuring the responsiveness of the API will give a good indication of the responsiveness overall. We use Apache JMeter to fire requests at the API and measure the time it takes to respond to each request. JMeter's 'latency' metric measures the time from just before sending the request, to just after the first part of the response has been received [14]. Thus, this metric includes the following:

time needed to assemble the request by the client
+ *time to connect to the server*
+ *latency towards the serve*
+ *time for processing the request by the server*
+ *time to generating a response by the server*
+ *latency back from the server.*

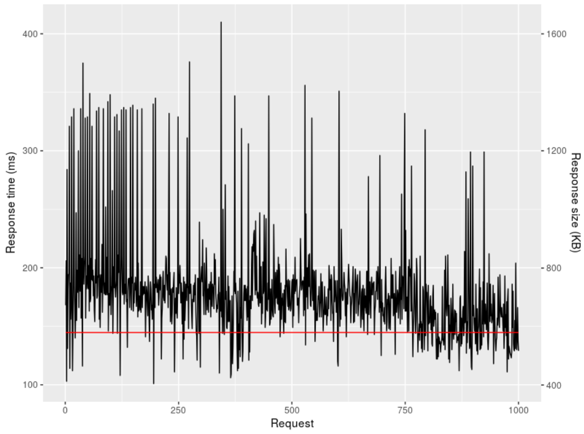
It excludes the transfer time of the complete response, and subsequent processing and rendering time by the client, because any client can do so differently, for example in a streaming fashion. We want to see that the response time is bounded, consistent and generally low. A Nexus 6 smartphone with Android 7.1 CyanogenMod is connected to a laptop running JMeter and the API port is forwarded with ADB over USB2.0. With JMeter we request the discovered channels from the API a 1,000 times at a constant rate of one request per second sequentially. A laptop is included to give some more perspective. Its specifications are listed in Table 6.2. The measurements are repeated for two scenarios: at first launch, and when almost no new channels are discovered anymore. Figure 6.8 shows the response times and sizes for every request in both scenarios for the laptop and Nexus 6 smartphone.



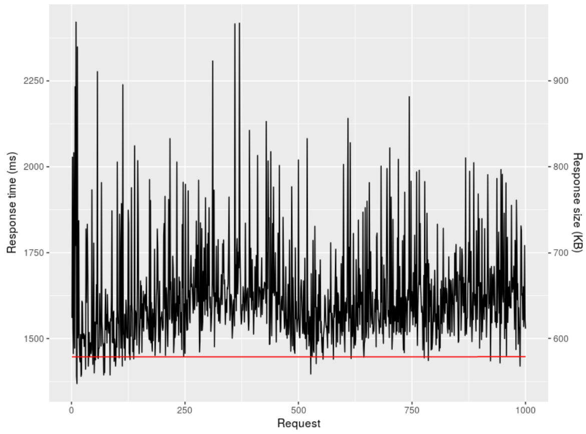
(a) Laptop, at first launch



(b) Nexus 6 smartphone, at first launch



(c) Laptop, after most channels are discovered



(d) Nexus 6 smartphone, after most channels are discovered

Figure 6.8: API response times (black) and sizes (red) for a 1,000 requests, to return all discovered channels, at first launch (top), and when almost no new channels are discovered anymore (bottom)
N.B. Response time is measured up to the moment the first part of a response has been received, therefore the transfer time of the entire response is not included

	N	Avg. (ms)	Min. (ms)	Max. (ms)	σ (ms)	Req./min.	KB/second	Avg. Bytes
Laptop	1000	65	1	1021	99.00	60.0	73.72	75410.6
Nexus 6	1000	3975	8	39477	5362.51	14.4	49.19	210506.8
Laptop	1000	181	101	416	42.72	60.0	579.49	592791.3
Nexus 6	1000	1671	1390	2925	174.14	35.9	345.94	592649.8

Table 6.4: Statistics of API response times and sizes from Figure 6.8a, 6.8b, 6.8c and 6.8d respectively

As shown in Table 6.4, the smartphone achieved a considerable lower amount of requests per minute than the constant rate of one per second in both scenarios. Therefore, because of the fixed amount of requests, proportionally more time elapsed during the measurements on the smartphone than on the laptop. This in turn, explains the significant difference in response size at the end of the measurements at first launch between the smartphone and the laptop. The throughput of the smartphone in the second scenario is higher than the throughput of the laptop in the first scenario. The 480 MB/s theoretical bandwidth of USB2.0 cannot be a bottleneck, considering the 579.49 KB/s result of the PC. From this, we can conclude that bandwidth is not a limiting factor in the first scenario for the smartphone. Although the requests are sequential, they can still influence each other, due to caching for example. Therefore, to explain the jitter, we have to look at the internals of the Tribler core, and the environment.

Tribler uses the event-driven networking engine Twisted, which is written in Python. Twisted allows you to build inter-process communication protocols, and provides the HTTP server 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 delays and potential timeouts. Also, only one thread can be executed at a time, because our implementation uses the CPython interpreter which has a global interpreter lock (GIL). CPython is optimized for single thread performance and compatibility with C extension modules, which are typically not thread-safe.

The fact that the smartphone in our experiment is not able to process one request per second, could indicate the multi-threaded processing capability is severely flawed due to the effects of the GIL on the Twisted reactor. Although each measurement is a snapshot, they were taken in a short time span from each other, so the environment is not expected to have a larger impact than the aforementioned effect. 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. There were no other active user-apps, but the OS itself contains system-apps that cannot be turned off. Finally, the first order derivative of the response size appears to coincide with a significant longer response time. If new content is discovered, that means other threads are active, which confirms our hypothesis that the GIL is to blame for poor multi-threaded performance. Further investigation should be conducted whether this phenomenon is also observed with lower amounts of requests per minute, in order to determine if the CPython has to be replaced by an interpreter without a GIL.

6.6. Profiling

Because of the challenges put forward in Chapter 3.7 and multi-threading issues found in Section 6.5, 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 a less powerful CPU may struggle with encryption tasks. Long running functions impact the responsiveness due to the GIL, as explained in Section 6.5. We focus on wall clock time, instead of CPU time, because this metric indicates the amount of time a user has to wait for a certain function to be executed. With the cProfile Python module we can measure wall clock time for each function call, to see if any function takes a disproportionate amount of time. A Nexus 6 smartphone with Android 6.0.1 CyanogenMod was used. The profiler was running for 10 minutes, with Tribler during normal operation and without any user input.

Table 6.5 shows that 27% of the time is spent on verifying cryptographic signatures, which is CPU bound. The database commit of SQLite3 is the slowest call, which is IO bound. Followed by the poll for new events by the Twisted reactor. The time per call, together with the number of calls, are the most important of the three measured metrics for effective optimization by parallelization. The reactor polling may be optimized by switching reactors, since there are many types with different behaviors The database commits can be optimized if less transactions are required by the protocol than are currently performed. The signature verification may be optimized by parallelization. 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 avoids the GIL completely, and 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 [8]. As seen from the time per call for __m2crypto.ecdsa_verify in Table 6.5 the multiprocessing module would likely cause too much overhead. However, another way to optimize is to use multiprocessing for separate Tribler communities, which is left for future work.

6.7. CPU utilization

In Section 6.6 we found that the cryptography tasks are taking a significant amount of time. For optimization purposes we need to confirm if this is CPU bounded.

Python-for-Android supplies a CPython interpreter out of the box. CPython is optimized for single thread performance and compatibility with C extension modules, which are typically not thread-safe. This interpreter 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, but 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 previous

# 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 6.5: Native function calls, sorted by wall clock time per call, during the 10 minute profiling (600 seconds total time)

section the crypto function took a considerable amount of time to compute. To see if the releasing the GIL is a feasible solution, we measure if the CPU has more capacity. Snapshots taken of the CPU utilization of the three separate processes involved in streaming HD video with Tribler. If there is any performance to be gained by releasing the GIL, the CPU must be significantly under-utilized in this use case, because other processes may also take up considerable CPU time. A Galaxy S3 smartphone with Android 6.0.1 CyanogenMod was used for this measurement. The HD video that was streamed has a bit rate of 4,565 kb/s.

Figure 6.9 shows that indeed not al 4 cores of the CPU are utilized by a large margin. The GUI is using the CPU barely, if at all, while VLC is playing the video at about 15%. The service's CPU utilization, streaming the video, tops out at around 25%. That leaves about 60% of the CPU to be utilized by other background apps and the OS. This suggests that releasing the GIL during heavy crypto work could result in a significant performance gain.

6.8. Software testing and code 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, since we know the tests do not cover everything. The nose module was used for running the tests together with the coverage module for gathering coverage data. The same Nexus 6 device was used in all runs, with Android 6.0.1 CyanogenMod installed for the first two runs, and Android 7.1 CyanogenMod the third and fourth run. Table 6.6 shows the results of both executions.

Run	Device	Tests	Errors	Failures	Skipped	Line-rate
Sat, 16 Jul 2016	Nexus 6	711	14	13	30	0.7241
Tue, 01 Nov 2016	Nexus 6	749	12	15	3	0.7861
Mon, 05 Dec 2016	Nexus 6	782	10	18	4	0.7871
Mon, 05 Dec 2016	PC	812	0	0	30	0.7894
Tue, 06 Dec 2016	PC	812	0	0	30	0.7901
Tue, 06 Dec 2016	Nexus 6	782	10	18	4	0.7897

Table 6.6: Tests results and coverage line-rate at different points in time during development

Failures occur if a test crashes and cannot complete. Errors occur if a test does not pass, because an assertion fails during the test. The difference in skipped tests, between the smartphone and the PC, can be explained by an import error just before the 26 desktop GUI tests are about to be skipped. If a failure like that occurs, tests in the same file are not even discovered by nose. The total number of tests has increased over time, as well as the coverage line-rate, while the number of errors have decreased. Concluding, all existing tests can be run on Android almost as successfully as on PC.

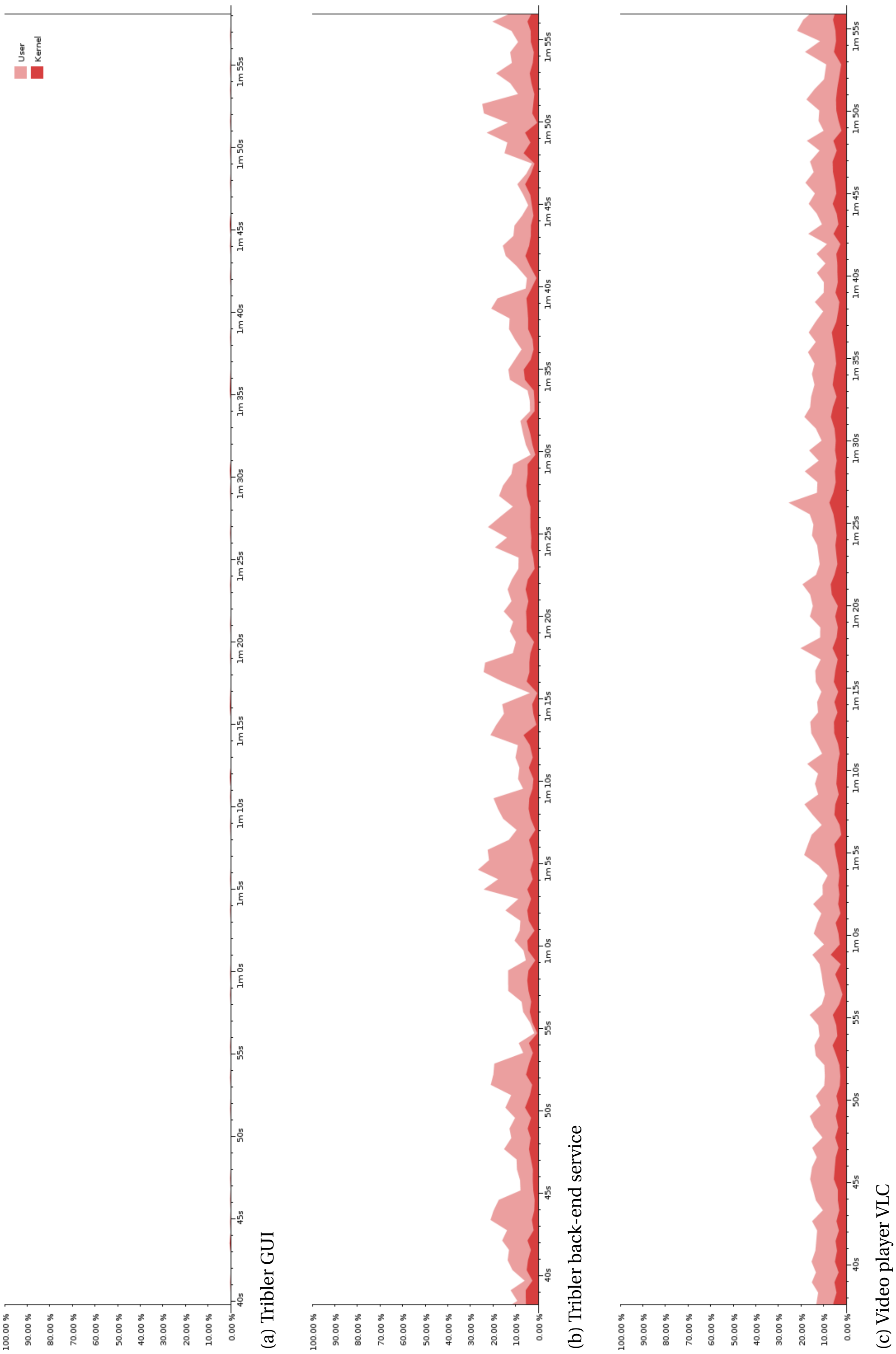


Figure 6.9: CPU utilization during HD video streaming on a Galaxy S3 smartphone

7

Conclusions and future work

All Tribler functionality runs on mobile devices with our implementation. The code is open source [2] and does not rely on any proprietary app store. This is the first successful attempt, to our knowledge, of creating a self-organizing video-on-demand platform that is attack-resilient and can operate autonomously on a mobile device. Millions of people that own a smartphone, and not a computer, can now benefit from Tribler's privacy enhancing functionality.

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

We proposed an Android implementation that fulfills all design requirements as specified in Chapter 4. It performs consistently, and faster than expected compared to a laptop, as shown in Chapter 6.

From the content discovery experiment, we learned that 9 devices discover new content within 4 seconds. Therefore, with viral spreading, it seems realistic we can reach millions within minutes. The Multichain experiment showed that creating blocks by the thousands is an IO bound process, rather than CPU bound, in the current implementation. It is also the most easy to optimize by keeping the hash of the last block for connected peers in memory. In the startup time experiment, the sample standard deviation is relatively small for all devices, which indicates that the startup time of Tribler is consistent. As expected, creating torrents appears to scale linearly with the size of the content, and adding a torrent to a channel does not scale with content size. The API performs consistently slower depending on the amount of data to be returned for a request, as expected because of JSON serialization. The API also performs consistently slower on a smartphone compared to a laptop. The first order derivative of the response size appears to coincide with a significant longer API response time. Further investigation should be conducted whether this phenomenon is also observed with lower amounts of requests per minute, in order to determine if the CPython has to be replaced by an interpreter without a GIL. Profiling revealed that cryptographic 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, because the CPU utilization of Tribler and the video player VLC combined did not reach more 40% while streaming HD video. All existing tests can be run on Android almost as successfully as on PC.

7.2. Given the constraints and unique abilities of mobile devices, what functionality of Tribler can be added or enhanced?

Mobile devices with WiFi, Bluetooth and NFC are optimally equipped for local data exchange without reliance on centralized infrastructure. These abilities have been fully employed in the current prototype, making easy to use and attack-resilient information spreading with Tribler reality.

A feature was added that transfers Tribler to a nearby device, with NFC and Bluetooth enabled, without further networking requirements, like an Internet connection or a central app store. By just holding to NFC equipped smartphones back to back, the transfer of the application started automatically. Also, users can make a channel their favorite in the same manner, from one nearby phone to another. Using their real life social network, users can build their own on-line trust network this way. And thanks to built-in capability of Android, it is possible to setup an ad hoc WiFi network to avoid any infrastructure completely. In the context of privacy and censorship, this off-grid functionality means that fast viral spreading, independent of infrastructure, is possible. Thanks to the properties of viral spreading, if content is detected by the censor, it may have already crossed the freedom border, or will still be capable of crossing it within a short time span.

7.3. Future work

Possible future work is presented with regard to the extensibility and sustainability of our current implementation in Subsection 7.3.1, and future research based on this work in Subsection 7.3.2.

7.3.1. Implementation

To safeguard the user from an even more powerful adversary as observed in China [28], more measures can be taken. Embedding and encrypting all functionality in for example a binary blob of a random game, would add another layer of security. Our SelfCompileApp [3] for example, capable of self-compilation from source, can be combined with this work to create a morphing stealth app for anonymous information sharing without the need for existing infrastructure.

From the results in Chapter 6, we found that performance can potentially be improved if the Python GIL is released during heavy cryptographic tasks by C/C++ libraries. We also suggest to cut long running methods into smaller pieces, to allow the Twisted reactor to interleave more threads, to improve responsiveness under heavy load in our multi-threaded use case. Also, a streaming API for big responses can improve the responsiveness of the API, as concluded in Section 6.5.

After our implementation was finished the standard Android integrated development environment (IDE) Android Studio started to officially support the Android NDK. Therefore, we can now move from the experimental alpha release to the stable Gradle plugin. This enables Gradle to cross-compile the C/C++ libraries instead of the build tool-chain of Python-for-Android (P4A). This in turn, makes it easier to replace P4A with for example Qt for Android, an alternative to P4A. The new desktop GUI of Tribler is built with Qt, and it would be nice to re-use code and improve maintainability. Qt for Android gained support for Android services after our implementation was already finished. This would also open the door to an iOS port, thanks to Qt for iOS. The modularity of the design and implementation enables Tribler to be easily ported to other platforms and embedded devices, like smart-TVs.

Finally, to remove the last potential hurdle for offline information exchange with Tribler between mobile devices, an updated list of bootstrap peers can be integrated into the APK, just before sending the app via Bluetooth. Or it can send via NFC, in the same way NFC is used to share channel identifiers between to NFC enabled devices. The Bluetooth transfer of the app itself can be made much faster if WiFi Direct is used instead. This can easily be done by directing the receiving device's browser to a local HTTP server on the sending device with a NFC Data Exchange Format (NDEF) URI Record.

7.3.2. Research

This work enables a new direction of research with Tribler, fully geared towards mobile devices. Future research can evaluate how well smartphones with Tribler can defeat the powerful adversary as described in Section 2.2. Several experiments can be thought of in the context of this mission.

As one example, one can setup a large-scale experiment with various degrees of powerful censors. Our implementation would enable the evaluation of Tribler in the wild, for the intended use-cases, as it can be live deployed on mobile devices in areas with restricted Internet access. Another possible research direction can consider how viral spreading of eyewitness content behaves in the real world. Furthermore, the effect on anonymity of local crowds, regarding the onion routing protocol, can also be studied.

A different direction enabled by our work, is to research the possible benefits of teaming a mobile device with a traditional desktop computer or server, with a shared key chain for a single user. The more powerful computer could be credit mining [5], while the mobile device uses the credits to download faster.

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] Paul W.G. Brussee. Tribler for android, 2016. URL https://github.com/Tribler/tribler/pull/2476.

[3] Paul W.G. Brussee and Johan A. Pouwelse. Autonomous smartphone apps: self-compilation, mutation, and viral spreading. *CoRR*, abs/1511.00444, Nov 2015. URL http://arxiv.org/abs/1511.00444.

[4] Paul W.G. Brussee and Johan A. Pouwelse. Survey of robust and resilient social media tools on android. *CoRR*, abs/1512.00071, 2015. URL http://arxiv.org/abs/1512.00071.

[5] Mihai Capotă, Johan Pouwelse, and Dick Epema. Decentralized credit mining in p2p systems. 2015.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services - MobiSys '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/1999995.2000018. URL http://dx.doi.org/10.1145/1999995.2000018.

[7] Maarten Clements, Arjen P. de Vries, Johan A. Pouwelse, Jun Wang, and Marcel J.T. Reinders. Evaluation of neighbourhood selection methods in decentralized recommendation systems. In Springer, editor, *Workshop on Large Scale Distributed Systems for Information Retrieval (LSDS-IR)*. Springer, July 2007. URL http://mmc.tudelft.nl/sites/default/files/Clements2007_LSDS.pdf.

[8] Nick Coghlan. Efficiently exploiting multiple cores with python, 2015. URL http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html.

[9] James Cowie, Alin Popescu, and Todd Underwood. Impact of hurricane katrina on internet infrastructure. Technical report, Renesys, September 2005. URL http://research.dyn.com/content/uploads/2013/05/Renesys-Katrina-Report-9sep2005.pdf.

[10] Martijn A. de Vos. Identifying and managing technical debt in complex distributed systems, 2016.

[11] 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.

[12] 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.

[13] Facebook. Introducing safety check, 2014. URL http://newsroom.fb.com/news/2014/10/introducing-safety-check/.

[14] Apache Software Foundation. *Apache JMeter Glossary*, 2016. URL https://jmeter.apache.org/usermanual/glossary.html.

[15] P. Garbacki, D.H.J. Epema, Johan A Pouwelse, and Maarten van Steen. Offloading servers with collaborative video on demand. In Adriana Iamnitchi and Stefan Saroiu, editors, *In 7th International Workshop on Peer-to-Peer Systems (IPTPS '08)*, Feb 2008. URL http://www.st.ewi.tudelft.nl/~pawel/pub/collvod.pdf.

[16] Gartner. Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2016, 2016. URL https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[17] Barton Gellman. Nsa broke privacy rules thousands of times per year, audit finds. 2015.

[18] Google. *C++ Library Support*, 2016. URL https://developer.android.com/ndk/guides/cpp-support.html#sr.

[19] Google. Platform versions, 2016. URL https://developer.android.com/about/dashboards/index.html.

[20] Google. *Intents and Intent Filters*, 2016. URL https://developer.android.com/guide/components/intents-filters.html.

[21] J. Alex Halderman. Internet censorship in iran: A first look. In *3rd USENIX Workshop on Free and Open Communications on the Internet*. Aryan Censorship Project, August 2013. URL https://jhalderm.com/pub/papers/iran-foci13.pdf.

[22] Shaddi Hasan, Yahel Ben-David, Giulia Fanti, Eric Brewer, and Scott Shenker. Building dissent networks: Towards effective countermeasures against large-scale communications blackouts. In *Proceedings of the 2nd USENIX Workshop on Free and Open Communications on the Internet, FOCI*, volume 13, 2013. URL https://www.eecs.berkeley.edu/~yahel/papers/Building_Dissent_Networks-Towards_Effective_Countermeasures_against_Large-Scale_Communications_Blackouts.FOCI2013.pdf.

[23] Karl Kathuria. Bypassing internet censorship for news broadcasters. *Usenix. org*, 2010. URL http://static.usenix.org/events/foci11/tech/slides/kathuria.pdf.

[24] 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.

[25] M. Meulpolder, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. Bartercast: A practical approach to prevent lazy freeriding in p2p networks. In State University of New York Yuanyuan Yang, editor, *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Los Alamitos, USA, May 2009. IEEE Computer Society. ISBN 978-1-4244-3750-4. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5160954&searchWithin%3Dp_Last_Names%3Ameulpolder%26matchBoolean%3Dtrue%26queryText%3D%28p_Authors%3Ameulpolder%29.

[26] J.J.D. Mol, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. Free-riding, fairness, and firewalls in p2p file-sharing. In K. Wehrle, W. Kellerer, S.K. Singhal, and R. Steinmetz, editors, *8-th IEEE International Conference on Peer-to-Peer Computing*, pages 301–310. IEEE Computer Society, sep 2008. ISBN 978-0-7695-3318-6. URL http://www.pds.ewi.tudelft.nl/pubs/papers/p2p2008.pdf.

[27] J.J.D. Mol, J.A. Pouwelse, M. Meulpolder, D.H.J. Epema, and H.J. Sips. Give-to-get: Free-riding-resilient video-on-demand in p2p systems. In *Multimedia Computing and Networking 2008*, volume 6818. SPIE Vol. 6818, January 2008. URL http://www.pds.ewi.tudelft.nl/pubs/papers/mmcn2008.pdf.

[28] Paul Mozur. China cuts mobile service of xinjiang residents evading internet filters, Nov 2015. URL http://www.nytimes.com/2015/11/24/business/international/china-cuts-mobile-service-of-xinjiang-residents-evading-internet-filters.html?_r=0.

[29] Steffan D. Norberhuis. Multichain: A cybercurrency for cooperation, 2015.

[30] Rutger S. Plak. Anonymous internet. Master's thesis, 2014.

[31] 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.

[32] Johan Pouwelse. Moving toward a censorship-free internet. *Internet Engineering Task Force*, 8(2):16–17, October 2012. URL https://www.ietfjournal.org/moving-toward-a-censorship-free-internet/.

[33] 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.

[34] 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.

[35] Redecentralize. Alternative internet, oct 2015. URL https://redecentralize.github.io/alternative-internet/.

[36] Renesys. Egypt leaves the internet, January 2011. URL http://research.dyn.com/2011/01/egypt-leaves-the-internet/.

[37] Renesys. Syrian internet is off the air, November 2012. URL http://research.dyn.com/2012/11/syria-off-the-air/.

[38] Jelle Roozenburg. Secure decentralized swarm discovery in tribler, 2006.

[39] Rob J. Ruigrok. Bittorrent file sharing using tor-like hidden services. Master's thesis, 2015.

[40] Blockchain Luxembourg S.A. Bitcoin average confirmation time, 2016. URL https://blockchain.info/charts/avg-confirmation-time.

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

[42] Jochen Spangenberg and Nicolaus Heise. News from the crowd. In *Proceedings of the 23rd International Conference on World Wide Web - WWW '14 Companion*. Association for Computing Machinery (ACM), 2014. doi: 10.1145/2567948.2579327. URL http://dx.doi.org/10.1145/2567948.2579327.

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

[44] Chris Tanaskoski. Anonymous hd video streaming. Master's thesis, 2014.

[45] New York Times, Paul Mozur, and Katie Benner. Apple is said to deactivate its news app in china, 10 2015. URL http://www.nytimes.com/2015/10/12/technology/apple-is-said-to-deactivate-its-news-app-in-china.html?partner=rss&emc=rss&_r=2.

[46] Twitter. Turkey blocks social media, 10 2015. URL https://twitter.com/policy/status/652893176676679680.

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

[48] VideoLAN. Vlc for android, 2016. URL https://www.videolan.org/vlc/download-android.html.

[49] Raynor Vliegendhart. Swarm discovery in tribler using 2-hop torrentsmell, 2010.

[50] 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/.

[51] Human Rights Watch. How censorship works in china: A brief overview, August 2006. URL https://www.hrw.org/reports/2006/china0806/3.htm.

[52] Jonathan Watts. Cuba's 'offline internet': no access, no power, no problem, December 2014. URL http://www.theguardian.com/world/2014/dec/23/cuba-offline-internet-weekly-packet-external-hard-drives.

[53] Jie Yang, Jun Wang, Maarten Clements, Johan A. Pouwelse, Arjen P. de Vries, and Marcel Reinders. An epidemic-based p2p recommender system. In *Workshop on Large Scale Distributed Systems for Information Retrieval (LSDS-IR) in SIGIR07*, 2007. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.7019&rep=rep1&type=pdf#page=16`.

[54] Niels Zeilemaker. Improving p2p keyword search by combining .torrent metadata and user preference in a semantic overlay, 2010.