TI3800 Bachelorproject

Android Tor Tribler Tunneling

Final Report

Authors:
Rolf Jagerman
Laurens Versluis
Martijn de Vos

Supervisor:
Dr. Ir. Johan Pouwelse
Project coach:
Ir. Egbert Bouman

June 23, 2014

Abstract

Tribler is a decentralized peer-to-peer file sharing system. Recently the Tribler development team has introduced anonymous internet communication using a Tor-like protocol in their trial version. The goal of our bachelor project is to port this technology to Android devices. This is a challenging task because cross-compiling the necessary libraries to the ARM CPU architecture is uncharted territory. We have successfully ported all dependencies of Tribler to Android. An application called Android Tor Tribler Tunneling (AT3) has been developed that tests whether these libraries work. This application downloads a test torrent and measures information such as CPU usage and download speed. Based on this information we have concluded that it is currently not viable to run the anonymous tunnels on an Android smartphone. Creating circuits with several hops that use encryption is very computationally expensive and modern smartphones can hardly keep up. By using optimized cryptographic libraries such as gmp or with the recently announced ARMv8 architecture which supports hardware-accelerated AES encryption, creating such circuits might become possible.

Preface

This document describes the bachelor project we performed at the TU Delft. Without the help of certain people at the TU Delft (and outside), this project would not be possible. In particular, we would like to thank the following people:

Johan Pouwelse, for his excellent guidance, deep insights and feedback.

The Tribler team, for always being able to help us with problems and questions.

Jaap van Touw, for taking the time to review our final report.

Steeve Morin, who visited Delft to help us with libtorrent.

Arvid Norberg, who helped us fix the libtorrent segmentation fault.

Contents

Chapter 1

Introduction

In recent years, censorship has become more and more apparent. In countries like China and North Korea, freedom of speech is at issue and the Internet is censored. Other countries such as Syria or Ukraine, which are currently in a state of turmoil, have their internet communications strictly monitored by both the military and the government. Freedom of speech is practically non-existent in those countries. To make sure people are still able to freely communicate, anonymity is of paramount importance. A popular protocol for anonymous internet communication is Tor[1]. However, Tor has disadvantages which limits its possibilities. More about this, can be read in the paper of Dingledine et al. [14]

Tribler is a fully decentralized peer-to-peer file sharing system developed by the Parallel and Distributed Systems group[2] at Delft University of Technology[3]. The Tribler development team[4] has recently introduced the anontunnels[5], a Tor replacement in a single Dispersy[20] community. The decentralized nature of the anontunnels makes it an interesting alternative to Tor for anonymous file sharing.

Mobile phones are increasingly being used for browsing the Internet and streaming video. Smartphones have surpassed 1 billion active users and are expected to double that amount by 2015 [19]. Popular mobile applications like WhatsApp have billions of users [8, 9]. Smartphones serve as an excellent platform for video recording and sharing due to their mobility and connectivity. Tribler's usage and popularity can be increased by supporting the smartphone operating system Android[6].

In this project we work towards achieving anonymous video streaming on Android smartphones. Our goals are to get Tribler and the anontunnels working on Android devices. Since Tribler is written in the Python programming language, it is very important to get Python working on Android. Additionally, we want to get all Tribler dependencies such as libtorrent and Dispersy – which are the key components for downloading – working on Android.

This thesis describes the prototype we have built and explains the steps we have taken to analyze our prototype. In Chapter 2, a problem definition is provided and we motivate the research question. In Chapter 3, we explain the prior work that has been done, regarding our project. In Chapter 4, we elaborate our software architecture. Chapters 5, 6 and 7 show our scrum iteration reports and explain in each scrum sprint what we have achieved and what not. Chapter 8 presents the experiments we have conducted with our prototype. In Chapter 9 we sum up our contributions to the open source community and explain what other people can use from our work. Finally, we conclude this thesis in Chapter 10. We recommend the reader to first read our plan of action which can be found in Appendix A.

[1] www.torproject.org/
[2] www.pds.ewi.tudelft.nl/
[3] www.tudelft.nl/en/
[4] www.tribler.org/trac
[5] www.github.com/Tribler/tribler/wiki/Anonymous-Downloading-and-Streaming-specifications
[6] www.android.com/about/

Chapter 2

Problem Definition

In this bachelor project we want to answer the following question:

Can we anonymously download and stream videos on Android smartphones?

Our focus within this question is on anonymous communication. We want to get Tribler, the anontunnels and all necessary dependencies working on Android smartphones. For the project to succeed, the following criteria must be met:

1. We must be able to run Python code on Android because Tribler is written in the Python programming language.
2. We must compile the following necessary dependencies to run on Android devices using the ARM[1] architecture:
   - libtorrent[2]
   - OpenSSL[3]
   - M2Crypto[4]
   - PyCrypto[5]
   - APSW[6]
   - netifaces[7]
3. We must be able to successfully download a file over the anonymous tunnels using the ported code.

2.1 Motivation

The reason why we are choosing Android, is because the platform is open source. Android is less restricted than closed source systems such as iOS[8]. Additionally, Android had a market share of 81% in the third quarter of 2013[2]. As stated in the introduction, there are more than one billion active mobile users. This means Tribler can reach a huge audience by going Android. Moreover, Tribler can scale because it uses Dispersy, where Tor is restricted to 1.2 million nodes [18]. If mobile devices would start to serve as relay nodes, Tribler is ready for it.

[1] www.arm.com
[2] www.rasterbar.com/products/libtorrent/
[3] www.openssl.org
[4] pypi.python.org/pypi/M2Crypto
[5] www.dlitz.net/software/pycrypto/
[6] www.github.com/rogerbinns/apsw
[7] pypi.python.org/pypi/netifaces
[8] www.apple.com/ios/

2.2 Tribler Play

Another bachelor project group is working on an application that is closely related to our application. This application is called Tribler Play. It can search for torrent files using the decentralized Dispersy network. Media files can be streamed using the torrent protocol and the built-in VLC for Android player[9]. More information about the project can be found on their GitHub repository[10].

To create the ultimate anonymous experience, our two applications will be merged at the end of the project. This final application could make anonymous streaming of video on the Android smartphone possible.

2.3 Android

In our preceding research question, we specifically want to research the possibility of an anonymous video streaming application on Android. Android is one of the most popular mobile operating systems available.

Writing applications for Android is traditionally done in Java. Google provides the Android SDK[11] and Android NDK[12] which contain the necessary tools to compile and build Android application packages (APKs). Several popular Integrated Development Environments (IDEs) are available such as Eclipse and Android Studio which should make the development process easier. An IDE allows developers to have a clear overview of the code and dependencies.

[9] www.videolan.org/vlc/download-android.html
[10] www.github.com/wtud/tsap
[11] developer.android.com/sdk/index.html
[12] developer.android.com/tools/sdk/ndk/index.html

Chapter 3

Prior Work

In this chapter, we will describe existing frameworks and software we can use. The largest anonymous network available now is Tor. We look at the advantages and disadvantages of Tor. Another way to anonymously share files, can be found in the Tribler software.

Since Tribler is written in Python, we need a framework that executes Python code on an Android device. Python for Android provides these tools and we will explain why and how we are using it in Section 3.1. In Section 3.2 we will review Tor, the current standard to communicate anonymously. Next, in Section 3.3 we will talk about Tribler. Tribler has implemented its own Tor-like protocol and aims for complete decentralization. Finally, The Global Square will be discussed in Section 3.4. The Global Square contributed to both Python for Android and the Tribler project.

3.1 Python for Android

The capability of running Python applications on Android devices is of paramount importance to the success of this project. Tribler is written in the Python programming language and has dependencies on many Python libraries. Fortunately a tool called Python for Android allows us to build and bundle Python code and its dependencies into standalone Android APK packages.

Python for Android uses Scripting Layer 4 Android[1] (SL4A) to run Python in an Android environment. For performance reasons this layer uses the CPython[2] binary compiled for the ARM architecture. This means that running python scripts will not work on other mobile architectures such as the Intel Atom x86[3] architecture. This restriction however allows us to focus the development solely on the ARM architecture. This mitigates some of the difficulty of getting Tribler dependencies such as libtorrent to work on different CPU architectures.

3.2 Tor

Tor, the second generation onion router, is a privacy-enhancing overlay network. Onion routing was first described by Chaum in his paper "Untraceable electronic mail, return addresses, and digital pseudonyms" [13]. Tor is the most widely used and secure implementation of onion routing. It was first implemented in 1996 by the U.S. Navy Research Laboratory as a means to protect government and military communications from digital and physical attacks [15].

Tor uses the principle of onion routing to ensure secure communication between parties. Network traffic is encrypted and forwarded over a circuit of nodes, see Figure 3.1. Each node in this circuit only knows the previous and next node. The communicating parties stay hidden when this circuit consists of at least three independent nodes [17].

[1] www.github.com/damonkohler/sl4a
[2] www.python.org
[3] www.intel.com/content/www/us/en/processors/atom/atom-processor.html

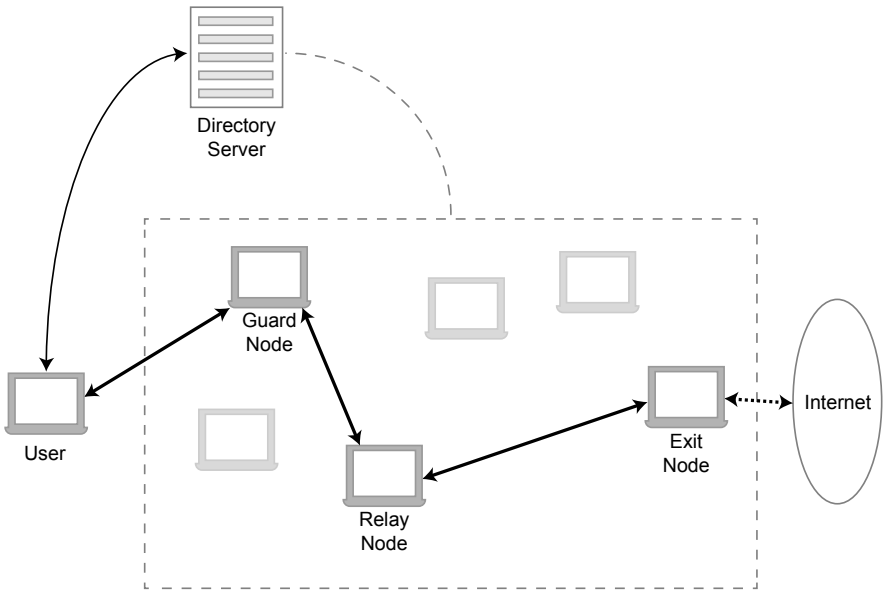

Figure 3.1: The components of the Tor network. After downloading the node list from the directory server, the user creates a circuit through a guard node, a relay node and an exit node. This circuit is used to communicate (anonymously) with the Internet.

There are several drawbacks to the current implementation of Tor. Centralized components, such as the directory server, act as a bottleneck and limit the number of possible users [17]. Since anonymity in a network such as Tor is directly linked to the number of active users this is an alarming situation.

3.3 Tribler

In this section an overview of Tribler and its components is presented. We describe the new trial version that uses anonymous tunnels in depth. Furthermore, we look at what specific dependencies Tribler relies on.

Tribler is a fully decentralized peer-to-peer file sharing system developed by the Parallel and Distributed Systems group at Delft University of Technology. It has been in development for over nine years and has a mature and well-established code base. It allows users to search for and share files in a fully decentralized way. The decentralized nature of Tribler has several advantages over existing file sharing systems. The lack of a centralized component makes it scalable and practically impossible to bring down.

An experimental version of Tribler is currently available that includes a Python implementation of a Tor-like protocol. This enables users to share files anonymously. By encrypting and routing traffic over a circuit of nodes, it ensures the communicating parties are oblivious of each other's virtual and physical location.

3.3.1 Downloading files

Tribler uses the torrent protocol[4] for downloading files. Torrent files contain metadata about the files that will be downloaded. The torrent protocol is peer-to-peer, which means that users download from each other. Users that provide files for their peers are called seeders. Tribler is

[4] www.bittorrent.org/beps/bep_0003.html

using the libtorrent library as an implementation of the torrent protocol. This library is licensed as open source software and we are allowed to use or modify it.

Torrent files can be found using the graphical user interface (GUI) of Tribler. Users enter their search query in a search bar and Tribler will return the results matching the search criteria. A family filter has been added to Tribler to filter out adult content. The underlying search is done using Dispersy which will be described in Subsection 3.3.4. In Figure 3.2 a general overview of the download process is given. The individual components like Dispersy and the anontunnels are described in more detail in the following sections.

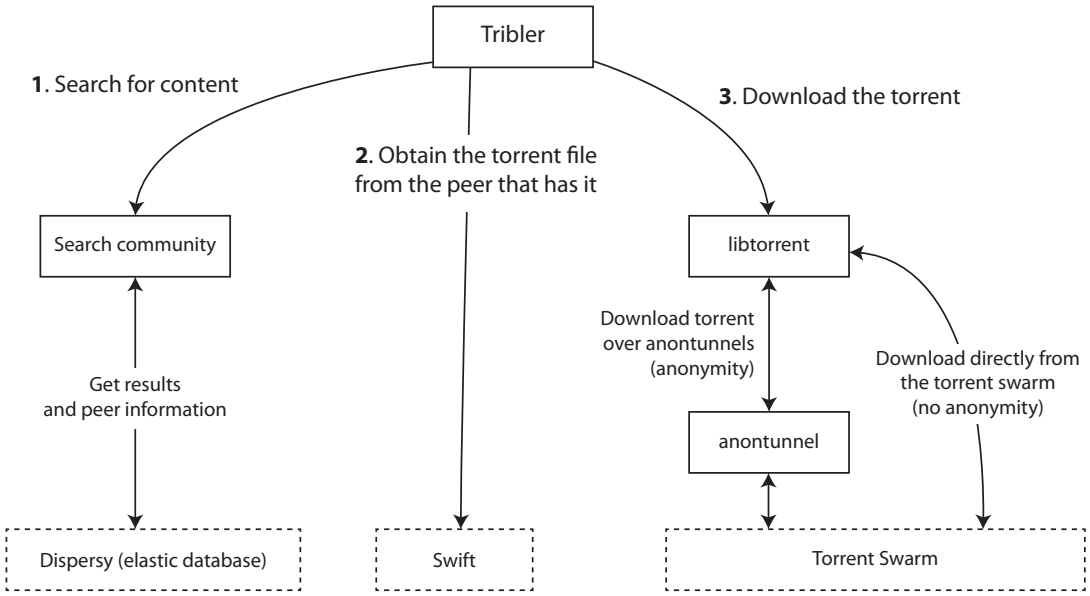

Figure 3.2: An overview of downloading a file using Tribler.

3.3.2 Anonymous tunnels

Recently, the research team of Tribler started to work on the implementation of anonymous downloads. Pull request[5] 525 on the Tribler GitHub page [6] is an experimental build of Tribler with the implementation of anonymous communications. The anonymous communication is achieved by using a Tor-like protocol. This protocol uses a three-hop circuit for anonymous communication. A circuit is a route from source to destination running over relay nodes (also called hops). Three-hop means that data travels over three nodes before it reaches the destination, to add anonymity. Note that Tribler does not use the Tor network, only a Tor-like protocol with UDP connections[6].

The Python module `Tribler.community.anontunnel`[7] contains the implementation of the anonymous tunnels.. Taking the code as reference, we now describe various details of the anonymous tunnels.

- The circuit setup is using the Diffie-Hellman[8] key exchange protocol to establish a secure connection. The M2Crypto library which is explained in Subsection 3.3.3, performs the Diffie-Hellman key exchange.
- The experimental code is using the Socks5[9] protocol for communication (see Figure 3.3).

[5] oss-watch.ac.uk/resources/pullrequest
[6] www.erg.abdn.ac.uk/ gorry/eg3561/inet-pages/udp.html
[7] www.github.com/Tribler/tribler/tree/devel/Tribler/community/anontunnel
[8] www.ietf.org/rfc/rfc2631.txt
[9] www.ietf.org/rfc/rfc1928.txt

- Dispersy is used as a data synchronization system.

In order to test the anonymous connections and the download speed, a special anonymous tab has been built into Tribler. Clicking on this tab brings up a graph of the current anonymous network as a graph (see Figure 3.4). It also logs the circuit events such as when extending or creating a circuit. When enough nodes are online, a 50 MB test download file starts to download.

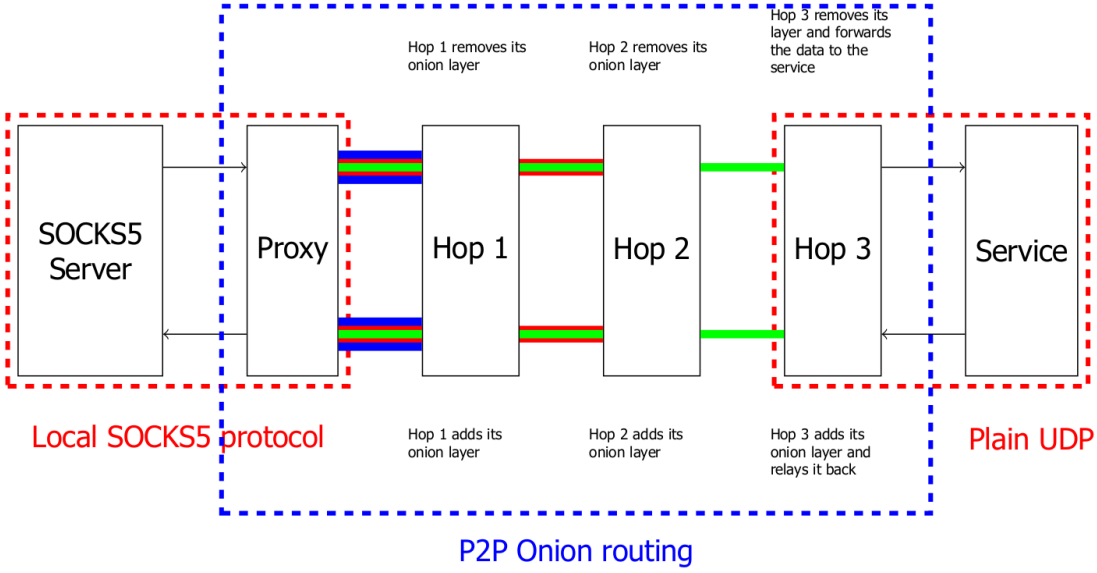

Figure 3.3: The onion encryption and decryption used by the anontunnels. Taken from www.github.com/Tribler/tribler/wiki/Anonymous-Downloading-and-Streaming-specifications

3.3.3 Security

Tribler makes use of cryptographic functions to encrypt data and make secure communication possible. Security is a big issue in the world of peer-to-peer networks. Not only do we want anonymous downloads, we also want confidentiality and integrity of our data. Confidentiality means that unauthorized parties cannot see the content of the information. This could be achieved by encrypting the data. Integrity of the data means that the data is protected from being modified by other parties or the network. Integrity is important when verification of the data is required.

Some well-known open source frameworks exist for these cryptographic tasks. An example is OpenSSL [10]. OpenSSL implements the popular SSL and TLS protocols[10]. These are cryptographic protocols that provide security when communicating over the internet. Besides that, OpenSSL provides libraries for various encryption and decryption protocols such as 3DES[11], RSA[12] and RC4[13]. OpenSSL also supports key exchange protocols such as Diffie-Hellman[14].

OpenSSL is written in the C programming language. To use the OpenSSL libraries in Python, Tribler uses M2Crypto [4]. The M2Crypto library, which acts a wrapper for OpenSSL, provides many implementations of popular cryptographic protocols that are used for secure communication.

[10] tools.ietf.org/html/rfc5246
[11] tools.ietf.org/html/rfc2420
[12] www.ietf.org/rfc/rfc2437.txt
[13] tools.ietf.org/html/rfc4757
[14] www.ietf.org/rfc/rfc2631.txt

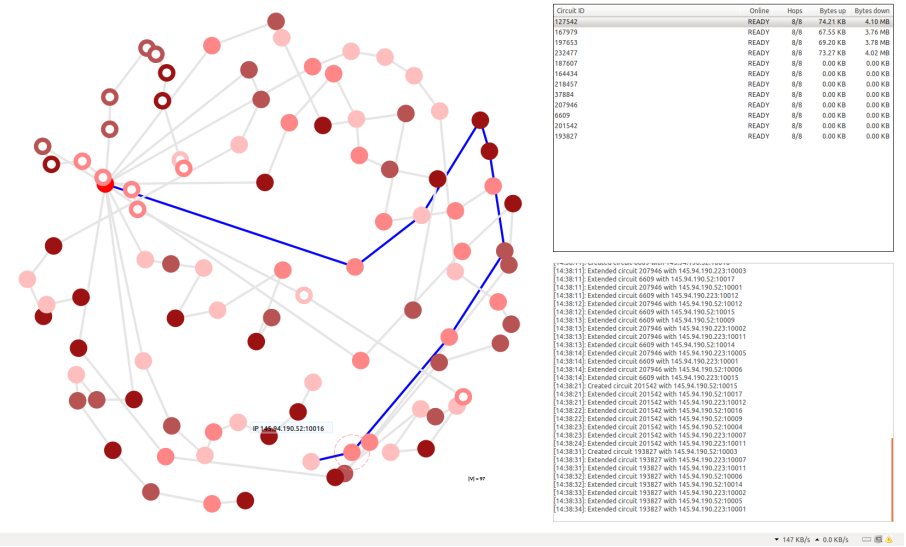

Figure 3.4: The graphical user interface in Tribler that shows the anonymous network.

3.3.4 Dispersy

Tribler makes use of Dispersy. As elaborated by Zeilemaker et al; Dispersy [20] is a fully decentralized system for data bundle synchronization. This means that data is exchanged between peers to make sure they are up-to-date with the same information. The system is designed in such a way that it is capable of running in a challenging network environment. Such an environment is often characterized by:

- Nodes randomly joining and leaving.
- Delays in the network.
- Nodes having different networking speeds (Edge, 3G, WiFi).
- Nodes often being behind routers that use Network Address Translating (NAT) and firewalls.

All communication done by Dispersy uses UDP. Because up to 64% of the Internet is behind a NAT, Dispersy has to use UDP NAT-firewall puncturing mechanisms[20].

In Dispersy, each node has a candidate list. A candidate list is a list of active connections within the node's overlay. Using this candidate list, a node can exchange data with its peers.

3.4 The Global Square

One of Tribler's contributors is The Global Square[15]. This organization has made contributions to Dispersy and libswift[16]. They also worked on the Python for Android library and documented the necessary steps to compile the libraries used by Tribler in Python for Android. Currently, TGS has become inactive and are no longer contributing.

Roarmag, an online journal writing about the global struggle for democracy, published a proposal in 2011 describing The Global Square: an online platform for our movement [11]. Their proposal is to make an online platform where people of all nations can come together as equals. This platform could be used to participate in the coordination of collective actions. It could provide the following tools:

[15] www.global-square.net
[16] www.libswift.org

- An interactive map that lists all ongoing assemblies around the world.
- A search option to find squares and events.
- Individual pages for each local square/assembly where they can organize events and share information.
- A public and private messaging system so individual users and groups can communicate with each other.

Traditional social media such as Facebook and Twitter, only allow to share and promote content, while TGS encourages the active participation of citizens and the consolidation of working groups in a local and global context between individuals and assemblies.

TGS has their own GitHub repository where they host contributions to various projects [5]. Notable are the contributions to the libswift and Dispersy projects. The last commits to these projects are over a year ago.

Another important contribution has been made to the Python for Android project. TGS has added the libswift, netifaces and M2Crypto libraries to the Python for Android project, making it possible to create an Android application bundle with these libraries included. This is an important contribution because it can be used as a first step to implement Tribler or anonymous tunnels on an Android device.

Chapter 4

Software architecture

Much of our work is focused on porting software libraries such as the Tribler core, the anontunnels and libtorrent to Android. We are not developing a typical software product. Instead, this bachelor project is more research-oriented. That means that we try to research our problem definition described in Chapter 2, with the help of a prototype application. We have developed an Android application called Android Tor Tribler Tunneling (AT3), a research prototype that we use to test Tribler and especially the anontunnels on Android (see Figure 4.1). Additionally, this software makes it possible to perform experiments such as bandwidth or CPU measurements, see Chapter 8.

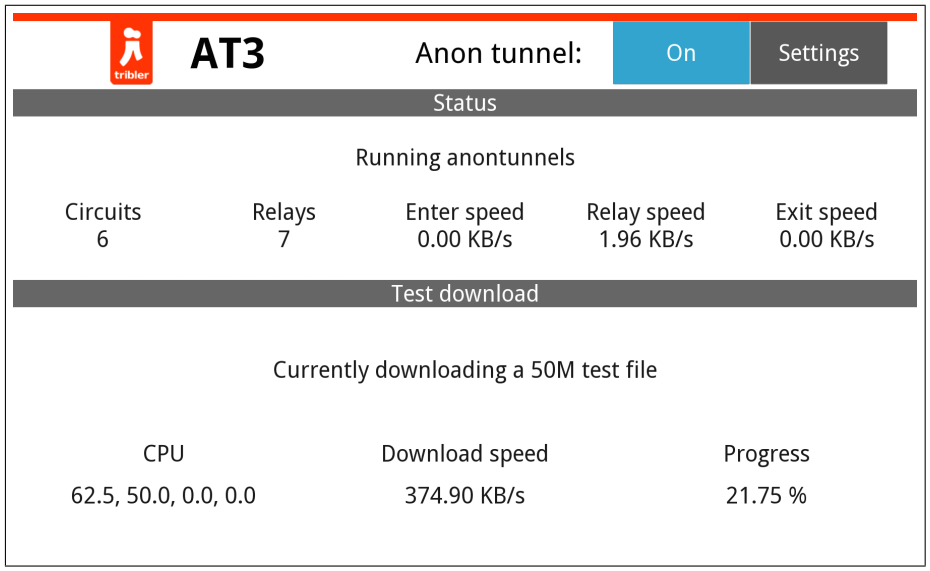

Figure 4.1: The AT3 application.

The general overview of the application is given in Figure 4.2. The AT3 application is a simple application used to test Tribler and the anontunnels on Android. It uses an Android service to run the anontunnels in the background. The service is a separate process and continues running even when the application is closed.

The main functionality is contained within the Tribler package, where we have created an additional `androidinterface.py` script. This script serves as an interface between the AT3 application and Tribler. It is used by the application's Android service to start the anontunnels, download torrents using Tribler and obtain status information about the torrent files that are being downloaded.

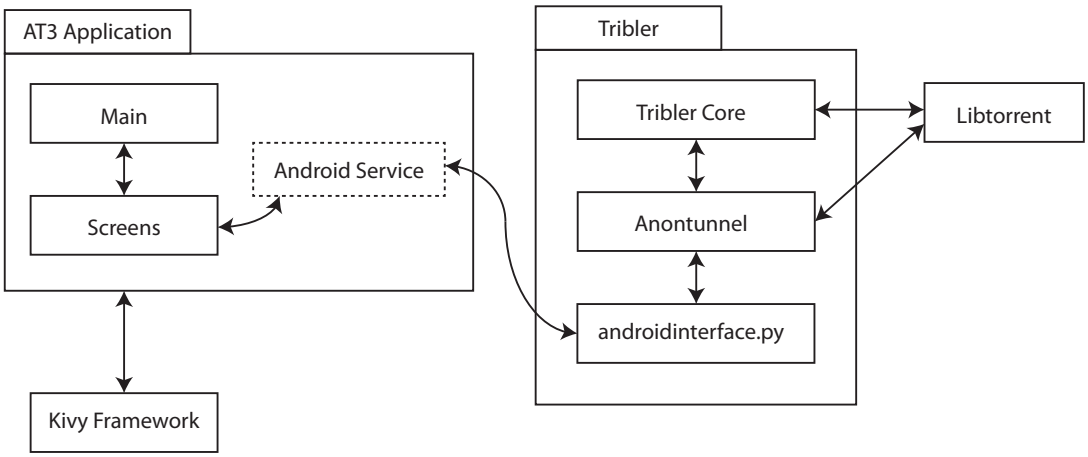

Figure 4.2: A high level overview of the application

The AT3 application uses Kivy. Kivy[1] is a cross-platform application development framework written in Python. Kivy includes Python for Android, a build tool that can generate standalone Android applications that run Python code. We chose to use Kivy for the development of the application because all existing Tribler and anontunnel code is written in Python. Therefore, this framework offers the most seamless integration with existing code. Kivy handles everything related to the user interface. It hides all the complexity of handling user input and displaying output.

Tribler handles everything related to managing torrent downloads. The anontunnels, which are a part of Tribler, handle the anonymous tunnels over which we will download a test file. These two components form the core part of our application. Tribler is dependent on several Python modules. A package dependency graph for Tribler can be found in Appendix B. One of the most important dependencies here is libtorrent, which is the library that handles torrent downloads.

The main functionality of the AT3 application is running the anontunnels and starting a test download. To do this, the application contains a start button which starts the anonymous tunnels and triggers a test download of 50 Megabyte. The sequence of this process is displayed in Figure 4.3.

Periodically, we want to obtain information about the status of the test download. Using a timer, the application will request this information every second from the Android interface. Because the AT3 application and the Android service run as different processes, the communication between them is facilitated by the Kivy OSC library[2]. The sequence diagram of this process is displayed in Figure 4.4.

[1] www.kivy.org
[2] www.github.com/kivy/kivy/blob/master/kivy/lib/osc/oscAPI.py

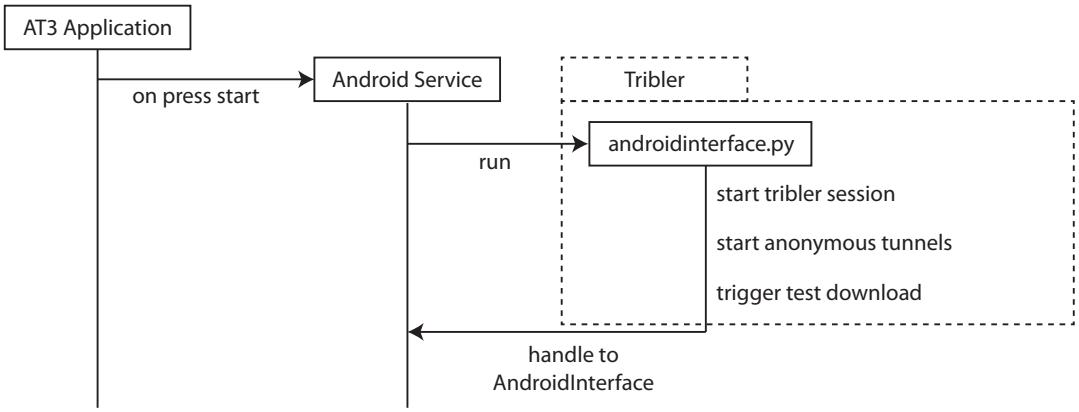

Figure 4.3: The sequence diagram of starting the anontunnels and triggering a 50 Megabyte torrent download

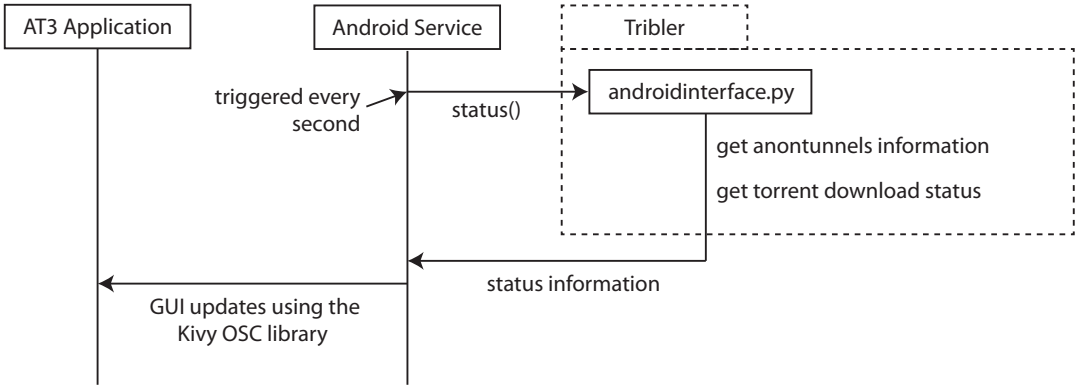

Figure 4.4: The sequence diagram of obtaining status information from the anontunnels and the test download

Chapter 5

Scrum iteration 1: creating a basic application

In this chapter we describe our first sprint that lasted two weeks.

5.1 Goals

During this sprint, we had the following goals:

1. Get Python code working on Android. We must evaluate and set up Python for Android (must have feature because this is the foundation of our whole project).
2. Get all packages that are needed to run the anonymous tunnels working on Android (should have, we should have most packages working but if there are some packages that do not work yet, it is manageable).
3. Implement a basic GUI to test with. This GUI should be created with Kivy (could have, we can use logcat – the Android logging tool – if we do not have a GUI available).

5.2 Python for Android

The Python for Android framework allows developers to add existing Python packages by creating recipes (for more information about recipes, see Section 9.1). As most of the packages were not available, we had to build a lot of these recipes ourselves.

Python for Android builds these packages for the ARM architecture. By using the `push arm` and `pop arm` commands in the recipe files, compilation for the ARM architecture can be triggered.

Below are the packages described we use in our application and in most cases had to create a recipe for. We describe the functionality of each package in our application and which dependencies it has.

- Kivy
  We use the Kivy framework for creating the Graphical User Interface (GUI). Kivy is an open source software library for creating GUI applications. It is easy to use and cross-platform, allowing users to create a GUI on their PC and then integrate it in their products (we integrate it in our Android application). As Kivy uses Python for Android, it is a natural choice to use it. Kivy is dependent on Python, as it is a Python package.
- OpenSSL
  OpenSSL is the world's most well-known open source cryptography toolkit, also available for Python. As our application makes use of PyCrypto and M2Crypto which are both dependent

on openSSL, we have to include it in our app. More information about OpenSSL can be found in Subsection 3.3.3.

- M2Crypto
  Dispersy and Tribler are dependent on M2Crypto as it has some security features which M2Crypto implements such as elliptic curves cryptography[1]. As the anonymous tunnels, our core function of the application, are dependent on both Dispersy and Tribler we also need M2Crypto. M2Crypto itself is dependent on Python and OpenSSL as it is a Python package using the OpenSSL implementation.

- PyCrypto
  PyCrypto is a Python library which implements certain cryptography functions used by the Tribler package. PyCrypto itself depends on functionality of the OpenSSL package.

- Boost[2]
  The libtorrent library is used to download torrent files with the Tor protocol. To compile libtorrent, the Boost package is required. Boost is a C++[3] library which provides code for multithreading, regex, math and asynchronous operations. We have to compile the Boost library with Python bindings enabled so we can invoke the library from Python code.

- netifaces
  The netifaces package provides methods for resolving addresses in a network. It is dependent on Python and used by the Dispersy package.

- Zope[4]
  Zope is an open source web framework for object-oriented web application servers. The Twisted package makes use of this framework for asynchronous networking tasks.

- Twisted[5]
  Twisted is an extensible framework for asynchronous networking written in Python. The framework has special focus on event-based network programming and multi-protocol integration. It has dependencies on the Zope framework. In the Tribler software, Twisted is used for callbacks of network events.

- anontunnels
  This package is the core of our application and contains the code needed for the anonymous tunnels. This package actually contains another package: the support for the Socks5 proxies. The files for this package come from pull request 525 on the Tribler GitHub page.

  We made some minor changes to this code: we have changed the master key of the anontunnel community to create our own testing environment. Several paths have been changed to ensure the anontunnels run on Android.

- Tribler
  This package has been obtained from the Tribler devel branch[6]. Changes to the source code were necessary to run this package on Android. In later versions these changes were reverted and the only modifications were to the Tribler path variables.

- Dispersy
  Since the code of the anonymous tunnels is using Dispersy for node discovery and data synchronization, we have created a Python package with all the code that is needed for Dispersy. This package can be run standalone.

[1] www.stanford.edu/class/cs259c/syllabus.html
[2] www.boost.org
[3] www.isocpp.org
[4] www.zope.org
[5] www.twistedmatrix.com/trac/
[6] www.github.com/Tribler/tribler

The files we have bundled are from pull request 525 on the Tribler GitHub. We did not use the files from the official Dispersy GitHub because this build was missing some classes we needed (for example, the `decorator.py`). Besides that, some changes in the pull request have been made to the Dispersy core to add support for the anonymous tunnels.

5.3 Porting the anonymous tunnels to Android

One of our first challenges was to port the code that sets up the anonymous communication to the Android device, using Python for Android. We started by inspecting the current code from pull request 525 and dived into the dependencies this code has with the Tribler core and Dispersy. We decided to create three packages: one package with the Dispersy code, one package with the files we needed from the Tribler core and one package containing the anonymous tunnels code.

After we created these packages, we had to find out which other packages we needed to run everything. To do this, we imported the dependencies in our application and ran it on the device. Each test run, we examined the import error and added the missing dependency. We used various packages that The Global Square has ported such as M2Crypto and netifaces.

Import errors were not the only issue we ran into: we had some problems with the netifaces package. Since this package is copied into the final APK file as a Python egg[7], it will be extracted on the device. This failed because the application did not have writing permissions. To solve this, we specified the egg extraction path and made sure the application has writing permissions to that path.

We also found out that some files were missing and not copied into the APK. The `curves.ec` file was missing and is needed by the cryptography classes found in the Tribler core. We also needed the configuration file of the logger, `logger.conf`. To make sure these files are part of the application, we copied them into the final application in our build script.

5.4 Attempt to compile libtorrent for Android

To get libtorrent working on Android there are several big obstacles:

- Compiling Boost for the ARM architecture.
- Compiling libtorrent for the ARM architecture.
- Compiling libtorrent Python bindings for the ARM architecture.

The official documentation of libtorrent states multiple ways to build libtorrent. The first way is using Boost's build system *Jam*. The second way is using *automake*[8]. However, before we can compile the libtorrent source code with either of these methods, several modifications had to be applied.

5.4.1 Source code modifications

We found several modifications to the source code to be necessary to compile libtorrent. These modifications are described below.

- `INT64_MAX` is not defined for Android, so we have to specifically define it.
- Multiple environments are defined in `include/libtorrent/config.hpp`. We add an environment for `ANDROID`, which sets the following options:
  - `FALLOCATE` is disabled
  - `ICONV` is disabled

[7] www.mrtopf.de/blog/en/a-small-introduction-to-python-eggs
[8] www.gnu.org/software/automake/

  - `IFADDRS` is disabled
  - `MEMALIGN` is enabled

- Instead of `<sys/statvfs.h>` we include `<sys/vfs.h>`, and redefine `statvfs` and `fstatvfs`. This is necessary because the Android libraries only have `sys/vfs.h` and not `sys/statvfs`.
- Finally, we add an include for `<sys/syscall.h>` and redefine `lseek` to `lseek64`.

5.4.2 Boost Jam

Boost Jam[9] is a build environment created specifically for Boost. However, it can be used to build other software. It is often included as a build option for software that is dependent on Boost, such as libtorrent.

Running Boost Jam is straight forward. After executing `bootstrap.sh` we can run `b2` or `bjam` to compile Boost itself. With this command we specify the architecture which is described in more detail in the `user-config.jam` configuration file. The resulting compiled library files are compatible with the ARM architecture.

Compiling libtorrent using this method is more advanced. We again specify a `user-config.jam` with appropriate settings. However, the build process fails during compilation. The source for libtorrent will have to be modified in several places, because the ARM compiler and libraries differ from the normal GNU[10] compiler. After modifying the source code, the build still fails. Linker errors occur when we try to compile the Python bindings. The standard Unix[11] libraries `pthread` and `util` do not have to be linked on Android, yet the Boost Jam environment forces these options for the Python bindings compilation. Due to the complexity of Boost Jam build environment, we decided to try using automake.

5.4.3 Automake

Automake is a standard set of tools for Unix-based systems that makes it more convenient to configure and compile software on a wide variety of systems. The tools are designed in such a way that it is possible to configure the build process using a simple script.

Following libtorrent's official documentation[12] we first run `bootstrap.sh`. Now, in order to configure and compile we will have to set up an environment in which it will use the Android GNU ARM compiler. To do this, we set the following environment variables:

```
export SYSROOT=$ANDROIDNDK/platforms/android-14/arch-arm
export PATH=/usr/local/gcc-4.8.0-arm-linux-androideabi/bin:$PATH
export CC=arm-linux-androideabi-gcc
export CXX=arm-linux-androideabi-g++
export CROSSHOST=arm-linux-androideabi
export CROSSHOME=/usr/local/gcc-4.8.0-arm-linux-androideabi
```

Note that we have set up a custom NDK toolchain. More information about setting up a custom toolchain can be found in Appendix D.

Compiling libtorrent with this set-up works, but the Python bindings still gives linker errors. These are the same errors as Boost Jam is showing. The linker tries to link `pthread` and `util`, which are not required on Android.

We will move two items involving libtorrent to the next scrum iterations:

- Compiling Python bindings for libtorrent
- Creating a proof-of-concept application to test if libtorrent works natively and with Python bindings.

[9] www.boost.org/boost-build2/doc/html/bbv2/jam.html
[10] gcc.gnu.org
[11] www.unix.org
[12] www.libtorrent.org/manual.html

5.5 Creating a Graphical User Interface with Kivy

The first version of AT3 used the standard output for printing status information. This required the phone to be connected to a computer so we can examine the log with the ADB[13] logcat tool. That is why we decided to create a Graphical User Interface (GUI) for our application. The purpose of this application is to provide a button to start the tunneling and a log to display the status of the application.

Creating a GUI was a small step for us: we already included the Kivy package in our Python for Android distribution. Creating interfaces in Kivy is similar to creating user interfaces for Android with Java: the layout is specified in Kivy files which have the *.kv* extension. In Python this interface file is loaded.

5.6 Sprint evaluation

During this sprint, we did not manage to complete all the goals set for this sprint. We did not succeed in compiling the libtorrent library. However, because we worked in parallel, we did start with some tasks we had set for the next sprint.

We have talked with Jaap van Touw, a member of the Tribler team. He told us that in his 20 weeks of work, he never managed to get the latest libtorrent to compile for Android. Currently he runs an old and modified libtorrent version. We managed to get in contact with Steeve Morin. He got the latest version of libtorrent working with Go bindings on Android. He gave us advice on compiling libtorrent on Android, possibly with Python bindings. We have set the libtorrent package as a separate goal for the next sprint.

[13] developer.android.com/tools/help/adb.html

Chapter 6

Scrum iteration 2: unit testing, relaying and libtorrent

In this chapter we describe our second sprint that lasted two weeks.

6.1 Goals

During this sprint, we had the following goals:

1. Get libtorrent from the first sprint working (must have, libtorrent is a high priority and we need libtorrent for our further work).
2. Set up a testing environment on Jenkins (should have, applications must be tested thoroughly to ensure everything is in order).
3. Get the anontunnels running, relaying and downloading (should have, we should investigate how these tunnels are working and how we can trigger a download, even if libtorrent is not working yet).
4. Merge our work with the other group to create a first prototype of the complete application (would have, this is very dependent on the progress of the other group).

6.2 Jenkins

In this sprint we decided to make use of the continuous integration system called Jenkins[1], which is already in use by the Tribler development team. Jenkins automatically runs specified tests when a build has been changed or a pull request comes in / has changed. This is a good addition to improve and maintain the (code) quality of our application.

Jenkins provides an environment to run tests in, which is perfect for our set-up. Currently Jenkins executes the following steps:

1. Jenkins cleans the environment when a build starts, so previous test runs do not influence the outcome of the test.
2. It then clones our repository and the Python for Android framework from GitHub.
3. If one of the tests fail, Jenkins will mark the build as failed. Otherwise the build succeeds.

[1] www.jenkins-ci.org

4. Finally, once the tests are done, the Tribler IRC[2] bot (an automated program that can insert message into an IRC chat group) reports the results of the test in our IRC chat group.

Because this is done for every change and pull request, we can closely monitor if changes have unexpected side effects. If they do, we can address them immediately to prevent the problem from spreading or growing more complex if the number of dependencies increase.

6.3 Libtorrent

We have continued working on libtorrent in this sprint and have made major progress in getting it to work on Android devices. Using Steeve's help from last sprint we were able to get a basic version of libtorrent to compile and link for Android. In this sprint we wanted to test the compiled version and build Python bindings.

6.3.1 Testing libtorrent with a simple application

We have built a simple test application with JNI[3] C++ bindings in order to test whether libtorrent actually works. This application does the minimal work required for downloading a torrent:

- It sets up a session.
- It starts listening.
- It opens a torrent file.
- It keeps looping while requesting status updates to keep track of the progress.

This very simple torrent client turned out to work well on a Galaxy S2 device. We were able to download the official Ubuntu[4] 14.04 distribution without problems. By manually checking the MD5[5] checksum of the file we were able to verify that it was downloaded successfully. We have chosen to download this particular torrent because it has many seeders and is big enough for a good test run.

6.3.2 Compiling Python bindings

To compile Python bindings, one has to add the `--enable-python-binding` to the `configure` call. However, doing that in our case causes the configure process to fail. The gcc-arm toolchain can not link with `-lpthread` and `-lutil`.

After more investigation we have found out what causes this. Essentially the configure process calls a bunch of *.m4*[6] files which try to find out the Python compilation settings automatically. This is done by running a Python interpreter and printing specific values obtained from distutils[7]. However, it turns out that the python process that gets run is the system Python installation (from Ubuntu). This does not match the same settings of Python for Androids its interpreter. To solve this we had to set some environment variables to point to the Python for Android interpreter:

- `PYTHON = /path/to/python-for-android/build/python/Python2.7.2/hostpython`
- `PYTHON_CPP_FLAGS="-I/path/to/python-for-android/python/Python2.7.2/Include"`

After setting this, the configure and compilation process runs fine and is able to create a `libtorrent.so` file with Python bindings. The compiled library works on a Galaxy S2 device. We created a simple Python application that downloads the Ubuntu distribution. This application runs without problems on a Samsung Galaxy S2.

[2] tools.ietf.org/html/rfc1459.html
[3] docs.oracle.com/javase/7/docs/technotes/guides/jni/
[4] www.ubuntu.com
[5] www.ietf.org/rfc/rfc1321.txt
[6] www.gnu.org/software/m4/m4.html
[7] docs.python.org/2/library/distutils.html

6.3.3 Segmentation faults on other devices

Running libtorrent, either the native JNI/C++ or the Python bindings version results in segmentation faults on some devices. For example, the Sony Xperia Z throws segmentation faults frequently, whereas this never occurs on the Samsung Galaxy S2. After a lot of debugging it is still not exactly clear what triggers these segmentation faults.

6.4 Downloading over the anonymous tunnels

One of the goals of this sprint was to find out how we can trigger a download of a torrent. We started to do this shortly after the libtorrent library worked on Android.

We found that modifications to our existing code were necessary because we discovered that a Tribler session is required to start the download of torrent files. The download itself is managed in the LibtorrentMgr class which is part of the Tribler package. The Tribler session is initialized when starting up Tribler with the GUI. This session also initializes Dispersy and handles the loading of the proxy community (among other communities). Once everything is running we download a file and verify its contents.

Our next step was to port this code to Android. After several import errors (for example, we had to remove the ncurses import and use apsw for the database transactions), they started to run. We first tried to download a torrent file with the computer as proxy. In some cases, this download triggers a segmentation fault which means that something goes wrong with regards to the libtorrent library (or the Python bindings). The error does not always show up: most of the time it occurred during the download. The application crashes and the download is stopped.

This is a serious issue we should further look into. Since the occurrence of the segmentation fault is random, it is hard to debug. We had several attempts to trace down the error, by disabling the anonymous download and using the Tribler session with minimal settings but our application still crashes during the download process.

6.5 Shell script unit tests

As we make use of shell scripts to set up variables, run checks and build the application with, we created unit tests.

The first thing we had to do is look for a shell script test framework. As no official test framework is available, we had to search for one that suits our needs. After comparing some frameworks, we decided to go with the shUnit2[8] test framework. This framework is lightweight and has some of the standard testing functions such as `AssertEqual`, `AssertTrue` and `AssertFalse` which are all we need.

In total we have created fourteen tests that check the following points:

- The required export variables.
- Whether certain necessary files exist.
- Build the application and test if everything runs correctly.

These tests cover all functions present in our build script and all possible subroutines. All of these tests are integrated in Jenkins as described in Section 6.2. This means whenever a pull request comes in that modifies code or packages that the application is dependent on, it will run the test to ensure the application still can be built and does not throw errors while building.

[8] shunit.sourceforge.net

6.6 Sprint evaluation

This sprint was a setback for us. Even though libtorrent throws a segmentation fault from time to time, we still managed to complete some download runs. Now that we have Jenkins up and running, we can focus on writing more automatic tests to measure changes and maintain a working prototype. The shell script tests we wrote during this sprint should provide a good way to ensure that our builds are correctly configured.

For the next sprint, we will look into the issue of the segmentation faults. As for Jenkins, we will write unit tests to test our written Python code and we will use the Kivy recorder to apply application tests on our application. These tests will also be sent to the Software Improvement Group (SIG) for evaluation.

Chapter 7

Scrum iteration 3: stabilizing libtorrent and experiments

In this chapter we describe our third sprint that lasted two weeks.

7.1 Goals

During this sprint, we had the following goals:

1. Write unit tests for our Python code and application tests for our application (must have: applications must be tested thoroughly to ensure all possible settings are working correctly).
2. Send our tests to SIG (must have: this is required for the bachelor project).
3. Investigate the segmentation fault and check for alternative solutions (should have: libtorrent is close to being stable, but due to time constraints we give priority to writing tests for SIG).
4. Update our application to use the new Tribler code (should have: while it is not a critical feature, it is good practice to keep the code up to date).
5. Measure CPU usage and download rates of our application (should have: while it is not needed in order to run the application, it is good to have some performance measurements).

7.2 Application tests

In order to verify that the application is working correctly we have our tests running on Jenkins. Using the default unit test framework provided by Python and the Kivy recorder we were able to set up basic user interface tests. These tests run the application, perform a series of user actions such as clicking buttons, and finally assert the state of the application. The tests currently run successfully on Jenkins, which means that future additions to the code will automatically be tested.

7.3 RUTracker libtorrent

Since we would like to have a stable libtorrent library that downloads a torrent without segmentation faults, we decided to try out the libtorrent that Jaap van Touw used in his bachelor project. This is an older version of libtorrent but has proven to be successful in his project. Jaap van Touw provided us with a link to a GitHub page that contains instructions on how to build libtorrent from

source[1]. This libtorrent version is used in an Android application called RUTracker[2], an Android application that allows users to download torrent files in the background. We are confident that this version of libtorrent is stable.

Instead of using a custom toolchain like we did to compile libtorrent-rasterbar, we used the toolchain that ships with the NDK. The first step was to build some Boost libraries libtorrent depends on (`Boost.filesystem`, `Boost.system` and `Boost.thread`). We used the Boost for Android project[3] to build Boost 1.49. After that, we compiled libtorrent according to the instructions that Jaap van Touw provided for us. We linked against the Boost libraries we just compiled and we got a library file that we can use in our Android application.

7.3.1 Python bindings

After writing a small example to test the stability of libtorrent, we came to the conclusion that it does not crash. Since this libtorrent version looked promising, we delved into the Python bindings that we need to communicate between Python and a native C library. Importing this library without Python bindings, results in an error that an initializer function could not be found.

For the Python bindings, it is convenient to use the `Boost.python` library. This library contains several macros and methods to easily define Python calls. The macro `BOOST_PYTHON_MODULE`, declared in `boost/python.hpp`, initializes our Python library and makes it ready for an import in Python. However, when running a minimal Python script that only imports libtorrent, a segmentation fault is thrown. This means that we are unable to use this version of libtorrent in Python. We are not sure what the cause of this error is. If the initialization and import of the library would work correctly, we could write our own bindings for the libtorrent functions and methods that we need in Tribler.

7.4 Libtorrent RC2 progress

While we were working on the Russian libtorrent version, we also kept working on the second Release Candidate (RC2[4]) of libtorrent we originally tried. The first step we took, was to compile libtorrent with asserts on. We did this to gain a better understanding in why the segmentation fault occurred. An assert had indeed triggered: in the destructor of the Torrent class, the m_abort variable should be true but it was false. This could mean that the Torrent object is released too soon.

We decided to post the issue to the official libtorrent bug tracker[5] where we got in contact with a libtorrent developer named Arvid Norberg. He helped us fix the error and gave us the advice to compile with the `BOOST_SP_USE_PTHREADS` flag. When we tried to compile with this flag we still got the segmentation fault. We took a closer look at what this flag exactly does. It turns out that this flag is responsible for the shared pointers: according to Arvid Norberg, with this define, the shared pointers are using mutex operations instead of atomic operations. He had some bad experience with atomic operations on embedded devices. Using this flag we were able to compile a stable version of libtorrent.

7.5 Updating the Tribler package

One of the goals of this sprint was to update the Tribler package we are using. The Tribler repository was updated. Twisted was updated to version 14.0.0, certain callbacks were removed and a Twisted Reactor[6] was introduced. When we began updating, we discovered that the default

[1] www.github.com/javto/tribler-streaming
[2] softwarrior.googlecode.com/svn/tags/RutrackerDownloader/2.6.5.5/
[3] www.github.com/MysticTreeGames/Boost-for-Android
[4] www.sourceforge.net/projects/libtorrent/files/libtorrent/
[5] code.google.com/p/libtorrent/issues/detail?id=627
[6] twistedmatrix.com/documents/12.0.0/core/howto/reactor-basics.html

recipe in the Python for Android framework required adaptation, upgrading the version number did not work. After inspecting the recipe we asked for advice on the Kivy repository, and they provided us with an updated recipe for Twisted 13.1.0. The updated recipe was also compatible with the latest Twisted release.

The next step was updating the Tribler package. Previously, we downloaded the package and adapted it. Because it was not forked on GitHub we could not automatically update it. We decided to fix this issue immediately and thus created a fork of the main Tribler branch. From this fork we modified the Tribler path variables to make the new Tribler code compatible with our application. This is necessary because several files that get opened by Tribler assume the working directory is the Tribler root. This does not work when executing on more exotic environments such as Android, where the working directory might be different.

Since our package is now a fork, future updates are more easy to merge into our package using the GitHub merge functionality.

7.6 Relaying and downloading over multiple hops/multiple circuits

In our last sprint, we had some issues with downloading and relaying over multiple hops / multiple circuits. Downloading the 50 MB test file was working correctly over one circuit with one hop, however, adjusting the values of the minimum amount of circuits required for the downloading and the length of the hop would not work.

To debug this issue, we got in contact with the original authors of the anontunnels code. They suggested to place loggers when receiving messages from other nodes and look closely to the incoming messages while disabling the encryption. We logged the messages but we could not find anything that could cause the application to reject circuits. The same code, when executed on a laptop, can successfully download over multiple circuits with multiple hops.

At this time, we were quite some commits behind the devel branch of Tribler. After updating and merging our package with the newest code, we tried again and the downloads were starting. We immediately tried to download the anonymous test file over four circuits and three hops and it worked fine. Also other lengths and other amounts of circuits were working correctly. During the tests (see Chapter 8) we came to the conclusion that three hop encryption is too heavy for a smartphone.

In order to verify the stability of relaying and downloading, even with encryption disabled, we decided to write a test application that can keep running overnight. We can start some anontunnel instances on our own computers and let the test application download the torrent. After the download is finished, the application restarts and the process starts over. The output is redirected to a log file so we can see issues or problems that occurred during the download.

7.7 Sprint evaluation

We managed to stabilize libtorrent, updated our Tribler package and made preparations to apply future updates more easy. Furthermore we managed to identify the problem with our download and relaying issue with multiple hops and multiple circuits. The stable environment and the option to download via multiple hops and circuits allowed us to gather data and generate graphs.

The additional unit tests contributed to the stability and robustness of our application. These new tests along with the previous ones from iteration 2 were sent to SIG for evaluation.

Looking back we can conclude that this sprint was successful. The most difficult challenges have been overcome.

Chapter 8

Experiments

In this chapter, we will discuss several experiments we have conducted. Since our application will be used for anonymous downloading and streaming of videos, we are interested in how our final application will perform and whether it is feasible to stream and watch videos on an Android smartphone. Our measurements can be divided in two categories: an analysis of the CPU usage and an analysis of the download speed.

First, a theoretical analysis is given in Section 8.1, which states the minimum performance required for video streaming. Next, we discuss how we have set up the experiments in Section 8.2. The actual measurements are provided in Section 8.3. Finally, we discuss our findings in Section 8.4.

8.1 Theoretical analysis

To make sure a video can be downloaded in a reasonable amount of time or streamed instantly, the application must achieve certain minimum bitrates. In the following Subsections, we will discuss the bitrate we need to stream a movie.

8.1.1 Required bitrate

Most smartphone screens do not have a high resolution, therefore we do not need to be able to stream very high resolution videos. The additional quality is negligible on such a small screen and therefore a waste of resources. For a smartphone, a resolution of 480p (854x480 pixels) or 720p (1280x720 pixels) is appropriate.

YouTube[1], a major video streaming platform, has published recommendations on the various bitrates for different resolutions[7]. The recommended bitrate for 480p is 1000 Kb/s and for 720p it is 2500 Kb/s. These values serve as an indication for the bitrates we will need to achieve.

8.1.2 Factors that impact the download speed

There are some factors that cause the download speed to increase or decrease. When downloading a torrent, one of the most important factors that influences the download speed is the amount of seeders. When we do not have enough seeders, we generally can not expect high download speeds. Another factor is the strength of each seeder: it is possible that some seeders have their upload rate limited or have a bad connection.

When downloading over anontunnels, an important factor is the amount of circuits. When we download over more circuits, the download should be faster. In general, the length of the circuits negatively impacts the download speed. When circuits are longer, there is a higher chance that

[1] www.youtube.com

there is a bottleneck in the circuit. The maximum speed of the download over one circuit is as fast as the slowest hop.

Another important factor we should take into account, is the possibility of random disconnects of peers during the download. When a seeding peer disconnects, the download speed of the torrent will decrease. When a circuit drops, our application will try to set up a new circuit or connect to an existing one. During this period, a decrease in download speed will occur.

Since there are many variables that influences the download speed, we cannot expect the speed rate to be stable. When streaming movies, these changing download rates could be problematic. A video should buffer to take these variable download speeds into consideration.

8.2 Set-up

To measure the performance of the anontunnels running on an Android device, we decided to make use of the Tribler test torrent. This test downloads a 50 MB test file on a Sony Xperia Z (C6603) connected to the eduroam Wi-Fi network. Running multiple times, each with a different setting of variables such as amount of hops or circuits, we can measure how these variables impact the performance of the CPU usage and download rates.

We are interested in the impact of the amount of hops and circuits on the performance of the anontunnels. The following four configurations were chosen to measure this:

1. Download the test file with 1 hop and 1 circuit.
2. Download the test file with 1 hop and 3 circuits.
3. Download the test file with 3 hops and 1 circuit.
4. Download the test file with 3 hops and 3 circuits.

CPU usage is measured with the *psutil* Python module[2]. During the tests we shut down all other applications running on the smartphone. This minimizes the impact other applications have on the CPU measurements.

Bandwidth is measured using the download status obtained from Tribler. This provides us with information about the test file we are downloading. This includes current speed (in KB/s) and the current progress (in percentage) of the download.

We run 10 stand-alone anontunnels on a MacBook Pro 7.1 running Ubuntu 14.04. This computer is connected to the Internet over an ethernet cable with a speed of 100 Mb/s.

During our experiments we found that the download would not start when enabling cryptography. CPU usage during these attempts would stay at 100%. This means that either the implementation of the cryptography is bad or the device's hardware is not powerful enough. For this reason we were forced to run the experiments with cryptography disabled.

8.3 Measurements

Using the set-up described in section 8.2, we have performed several experiments.

By looking at the CPU usage we can determine whether running the anontunnels is too computationally expensive for the application of video streaming. Even without cryptography, we are seeing a CPU utilization of about 75% when downloading (see Figure 8.1). Due to the fact that we are not using cryptography, the amount of hops or circuits does not impact the CPU performance.

It is also interesting to look at the download speed of the application. This indicates whether we can accomplish the necessary bitrate to stream video, which is about 1000 Kb/s (see Subsection 8.1.1). We managed to achieve a download speed on our test set-up of around 500 - 600 KB/s, which corresponds to 4000 - 4800 Kb/s (see Figure 8.2). This is sufficient for the application of

[2] pypi.python.org/pypi/psutil

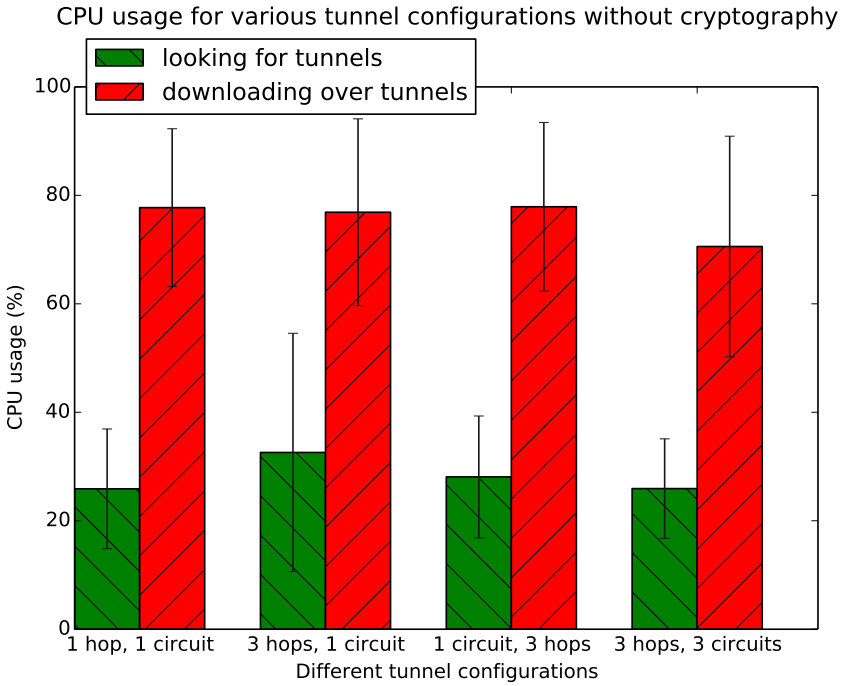

Figure 8.1: CPU measurements for various tunnel configurations without cryptography.

video streaming. Due to the fact that all stand-alone anontunnels are running on the same computer, they will all perform equally well. Not a single tunnel will act as a bottleneck. Additionally, the computer running the anontunnels is connected to the internet with a high speed connection of 100 Mb/s. Because of these factors, the impact of the amount of circuits or hops is negligible. The computer is capable of relaying data fast enough to not impact the download speed of the Android device.

The time between when the download is triggered and when it actually receives its first byte is about 140 - 150 seconds when using one circuit. With three circuits, this takes about 180 - 190 seconds. This difference can be explained by the fact that more circuits take more time to set up.

8.4 Conclusion

We come to the conclusion that streaming video over the anonymous tunnels is possible, as long as cryptography is disabled. The average download speed is capable of providing the necessary bitrate for videos with a 720p resolution.

The unforeseen CPU bound that we encountered means that either the cryptographical implementation is bad or the current generation smartphones are not powerful enough. By compiling and running gmpy[3], a fast large number library for Python, we could see a large speedup during the Diffie-Hellman handshake. Due to time constraints we were unable to compile and implement this in the current version of the AT3 application.

It was announced that the new ARMv8 architecture will support AES hardware acceleration [16]. Because the anontunnels use AES encryption, this would increase performance and reduce the computational burden on the CPU.

Without a significant speedup of the cryptographic functions, our conclusion is that the anontunnels will not be able to run with three hops and encryption enabled.

[3] www.gmpy.org

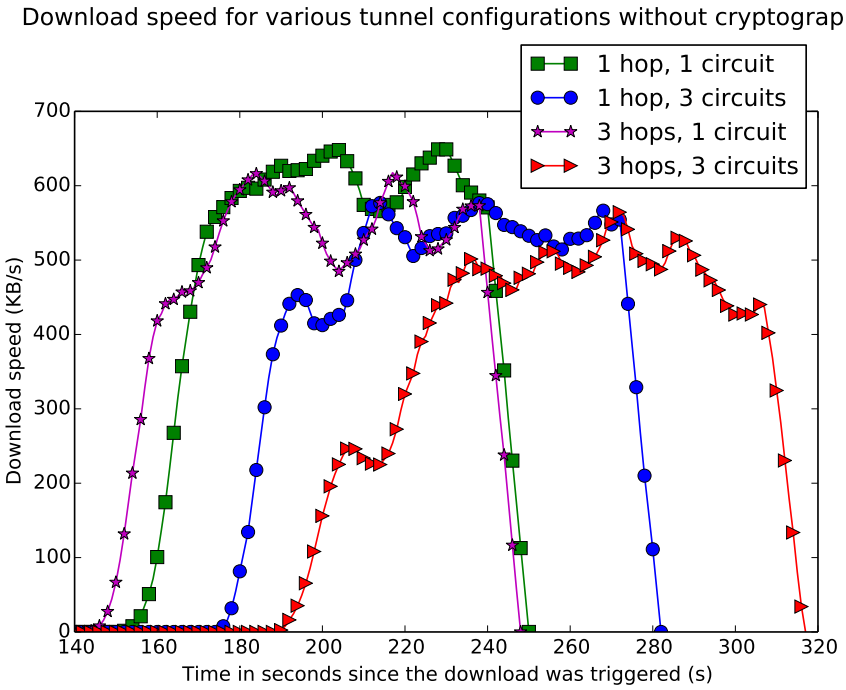

Figure 8.2: Download speed measured for various tunnel configurations without cryptography. The plot has been smoothed with a moving average of 10 seconds.

Chapter 9

Our contributions to the open source community

During our bachelor project, we used many open source libraries and frameworks. In addition, we received help from several people including Steeve Morin who personally came to Delft to help us. In this chapter, we will describe our contribution to the world of open source software, explain what we have achieved and what we have done to make software people can use.

9.1 Python for Android

Python for Android enabled us to run Python code on Android. We made use of many Python for Android recipes. A recipe downloads a package, extracts the contents and applies operations on them when required. In addition, we updated some recipes and created our own. For example, when Tribler started using the newest version of Twisted, we updated the Twisted recipe to version 14.0.0.

Examples of recipes we created are libtorrent and Tribler. The libtorrent recipe contains the shared object file that is being copied to the final Python distribution on the Android device. The Tribler package contains all Tribler code with some minor modifications to make it work on Android. These two packages and our updated packages will be submitted to the official Python for Android repository via a pull request because they have the potential to be used in other Python for Android projects in the future.

Because of the separated nature of the recipe system of Python for Android, it is easy for developers to add their own recipes. Several widely used packages such as gmpy, which is an optional dependency of Tribler, are not available as a recipe yet. In the future, a contribution could be to make a recipe of gmpy and submit it to the Python for Android repository via a pull request.

9.2 Libtorrent

Our work was greatly dependent on one of the world's most popular libraries: libtorrent. Since so much time and effort has gone into libtorrent we decided to create a separate tutorial about the compilation process. This guide is available in appendix D. This means that other people can extend our work and use our building process to compile libtorrent for their own purposes. In the future, a recipe that performs the compilation process of libtorrent can be made.

When compiling libtorrent, we used the Boost library that provides advanced C++[1] features such as memory management and support for multi-threading. We made use of libtorrent-rasterbar's Python bindings to invoke the libtorrent library in Python code.

[1] www.cplusplus.com

9.3 Tribler

Our created prototype provides the option to download anonymously using the experimental anontunnels code. This application also uses the Tribler core code. A future goal is that our application will be merged with the Tribler Play project. This means that we will have an application that allows searching the Dispersy communities and anonymously download / stream torrent movies. An extension of Tribler to the mobile platform could be a big step towards anonymity on the internet.

During the process, we identified several issues regarding the anontunnels and Tribler itself. Some of these issues were fixed and we contributed to the stability of Tribler. During the project, we had close contact with the Tribler team and we spoke regularly with members over mail, IRC or in person.

Chapter 10

Conclusion

We come to the conclusion that the current generation smartphones are not ready yet for anontunnels. The project itself was successful; we managed to cross-compile all required libraries. The anontunnels are running and downloading when the hop count is low. Downloading over multiple circuits is possible and relaying of data works too. By creating a pull request, we provide Python for Android with some useful recipes, contributing to the open source community. All of our work, including this thesis, is open source and can be extended. Jaap van Touw, a master student currently working for the Tribler team, will merge our project with the other bachelor project group Tribler Play, creating an application that allows anonymous and encrypted streaming of movies, with a search and playback option.

We also contributed to the open source community. In particular we contributed to the following open source projects:

- Python for Android
- Tribler
- libtorrent

Once smartphones running on ARMv8 become available or if gmpy is integrated in the project, we foresee that this application will be able to run with 3 hops and multiple circuits wile having cryptography enabled. Once that is possible, Tribler takes a censorship-free internet to a new level.

10.1 Future work

If one wishes to extend our work, we recommend to try and get gmpy to compile by creating a recipe for it. This might speed up the cryptographic part, allowing for multiple hop with encryption downloading.

We would also recommend working closely together with the Tribler team, as the application integrates the code of the Tribler main code base. The Tribler team often can help with questions or advise you on best practice regarding their code.

Finally, we recommend to keep up-to-date with dependencies whenever you can. The Tribler code base changes regularly, so it is good practice to keep up-to-date. Other libraries such as OpenSSL have had some fixes regarding the famous Heartbleed bug[1]. Applying patches like these might be critical in terms of functionality or security.

[1] www.heartbleed.com

10.2 Reflections

10.2.1 Reflection Rolf

In contrast to a traditional Bachelor project where one goes through a software development process, we have decided to take a more research oriented approach. As I aim for an academic career, this bachelor project fits my personal goals very well.

The team worked together excellently with no problems in cooperation. Despite working on separate components simultaneously, everyone was kept up to date with the latest developments. Throughout the project each member shifted focus to different parts of the problem. This allowed every one of us to grasp different aspects of the project.

Our project was about porting and running bleeding edge experimental code on exotic devices and systems. This was challenging in particular because it was uncharted territory. At the start of the project it was not sure if it would be possible to get all Tribler code and dependencies working on Android. I am glad that through hard work and effort we were able to accomplish this. Parts of our work are going to be contributed to open source software and Tribler. Contributing to an open and censorship-free Internet in this way is worth all the work we have put into this and I am happy to have been a part of it.

10.2.2 Reflection Laurens

This project was interesting and challenging on certain points. We worked well together as a team and it was very interesting to be in uncharted territory. This project was more of a research project than a software development process.

I think some advice in a reflection could prove to be helpful. My first piece of advice would be to not be afraid of asking for help. Just as we did libtorrent and the anontunnel code, ask the authors or developers for help. If you are stuck on a certain area where expertise is required, you can save a lot of time if you ask the right persons. Since you show interest in their project they are often more than happy to help you. Without Arvid and Steeve, we most likely would still have an unstable libtorrent library.

We chose to have a switch from time to time in tasks, so that every member is forced to look into at least one other subject he is not currently working on. For example, I switched from shell scripting and setting up Jenkins to the anontunnel code and measuring performance. In turn, Martijn who has been working on the anontunnel code switched to investigating the Russian version of libtorrent and Rolf took over Jenkins. This way, we all learned more about the system and were forced to get an understanding of what the other was doing. If something was unclear we made sure it was immediately fixed, commentated and if needed, refactored.

One thing we probably should have done more often, is asking for feedback on written work. We had our thesis reviewed three times by our client. I believe more reviews allowed for more smaller changes rather than a large amount of work towards the end. Since the client only has a high level goal in mind, he / she often provides complementary feedback.

Concluding, I think this has been a huge step towards our client's ultimate goal. It was a real pleasure to work with Rolf and Martijn as well as the Tribler team. Since our project will be merged with the other work of the other bachelor project group soon, our effort and work will continue, which gives a content feeling. Once the encryption and decryption of data becomes viable for smartphones, Tribler will take a censorship-free internet to a new level.

10.2.3 Reflection Martijn

Working together with the Tribler team was interesting and fun. Tribler is an interesting system to work on and we had some challenging issues. The most important issue we had, was compiling libtorrent for Android. When performing the cross-compilation, I learned more about the compilation process and gained more insights in what exactly is going on when compiling. I think this is a valuable experience because this knowledge can be used in other projects as well.

The Tribler team was always willing to lend us a hand. Our client visited us regularly and introduced new people to us who might be interested in our work. During the project, I started to realize how important it is to communicate clearly with other team members and the Tribler team. In the past, several issues have emerged because there was no clear communication between members. We wanted to avoid that so we did everything to keep the Tribler team up to date with our latest progress and updates. They also got in contact with us when they had a new version of Tribler or when a bug was introduced which could affect our work.

Working with bleeding edge code was sometimes a bit frustrating because there were some bugs we had to deal with. Furthermore the code was updated quite frequently so we had to update and modify our code too to be able to run it on Android. During the process, we tried to automate these code updates so it became easier for us to keep up to date with the latest code. I recommend everyone who works with very new code, to automate the code update process: in the long run, it could save you a lot of time and effort.

Working with Python was a new experience for me: when I started with the project, I preferred languages like C and Java but during the project, I started to see the power of using Python. Changes to the code were very easy to make and could be integrated in our application quite fast. I learned a few tricks about the library and package structure Python is using and this knowledge can also be used in further projects. Working through the import errors learned us about the libraries that Python for Android already included and which not. Creating our own recipes was very fun to do and we were always glad when a new package was functioning correctly on the Android phone, especially when we had a stable version of libtorrent running.

Overall, I liked the project. We did some very interesting tasks such as the compilation of libtorrent and the code analysis of Tribler. I recommend students with a prior experience in software engineering and those who are interested in doing research in new fields to work on the team and do their thesis on the Tribler system. I hope that our application will eventually be integrated with the application of the other bachelor thesis group. After all, our project will be part of another greater system where Android users can use our applications to anonymously stream movies.

Appendix A

Plan of Action

This plan of action is part of the bachelor project on anonymous video streaming on tablets. In this project we attempt to get Tribler, a peer-to-peer file sharing application working anonymously on Android using an already implemented Tor-like protocol.

In this appendix we outline the details of the assignment. We will describe the approach we will use and how we will structure this project. Finally, we will describe how we maintain the quality of our work.

A.1 Assignment

In this section we will describe our assignment, client, contacts, the problem we will address and sketch out what product we will eventually deliver. This section will also contain some of the critical requirements that we will have to meet along with the risk involved.

A.1.1 Assignment

Our assignment is to implement a new feature of Tribler into a mobile android application. This new feature allows the creation and usage of so called anontunnels. These tunnels allow anonymous download within a peer-to-peer network between devices, in our case Android smartphones. As these tunnels run on Python code, we will have to be able to run Python on an Android smartphone, along with all the libraries it depends on.

A.1.2 The client

Our client is Dr. Ir. Johan Pouwelse, the head of the Tribler group and Assistant Professor at the Parallel and Distributed Systems Group of the Faculty of EEMCS, Delft University of Technology. Pouwelse has measured and researched peer-to-peer networks for years and has been working on Tribler for nine years.

A.1.3 Contacts

The Client:

Dr. Ir. Johan Pouwelse
J.A.Pouwelse@tudelft.nl
+31 (0)15 27 82539
Room: HB 07.290
Mekelweg 4
2628 CD Delft

TU Delft coach:

Ir. Egbert Bouman
E.Bouman@tudelft.nl
Room: HB 07.290
Mekelweg 4
2628 CD Delft

Bachelor Project Coordinator:

Dr. Martha A. Larson
M.A.Larson@tudelft.nl
+31 (0)15 27 87357
Room: HB 11.040
Mekelweg 4
2628 CD Delft

A.1.4 The final product

At the end of the bachelor project, we will deliver an Android application that allows users to find and download content anonymously. This application makes use of the code from the Tribler project, which is written in Python. Therefore, we will make sure that the Android application will be able to run Python code.
The downloads that run through the anontunnels will be anonymous, just like the current Tor protocol.

A.1.5 Requirements and risks

The final product will offer the features that are described in Subsection A.1.4 and should be considered a prototype. The prototype will offer anonymous downloading using the anontunnels mentioned earlier. The development of the prototype will be targeted to the Android platform.

The following requirements are set:

- Weekly Scrum evaluations. At the end of each Scrum iteration, we evaluate what we have done and set the target for next week. This keeps the deadlines SMART[1] and manageable.
- Weekly meeting with the client. Every two weeks we will implement a feature, but we will show and discuss our progress each week with the supervisor.
- The members of the team will complete a prototype at the end of this project and will also demonstrate this during a 30 minute presentation given in the last week of Q4.

The risks involved with this project include:

- Run Python code on an Android application. As the Tor-tunnel functionality is written in Python code, we need to be able to run Python code on an Android device as well. A library exist where you can write Python for Android, but we will still face a challenge when we will try to combine other libraries.

[1] www.techrepublic.com/article/use-smart-goals-to-launch-management-by-objectives-plan/

- As we are dependent on third party code, we might lose time to understand or read certain parts of code or documentation as well as link pieces of code that belong to different parties together.

A.2 Approach

In this section, we will describe the approach we are taking for this project. First, we will discuss the methodology we are using (Scrum). After that, we will discuss the MoSCoW technique we are using to classify requirements. Afterwards, we will give an overview of the tools we will be using during this project. Finally, we will give our planning and milestones.

A.2.1 Scrum methodology

For the project, we will be using the Scrum methodology. We have used Scrum in various projects already during our studies and it has proven to be a very efficient way of working. In this section, we will discuss how we plan to use Scrum and what our Scrum iterations will look like. First we will look at the different roles involved in the Scrum process. After that, we will describe how the Scrum process is organized.

Scrum roles

There are three primary roles involved in Scrum:

- Product owner: the product owner represents the stakeholders and is the voice of the customer. He is responsible for the success of the product. Johan Pouwelse is the product owner.
- Development team: the development team consists of Rolf, Laurens and Martijn. We are responsible for delivering the final product to the product owner.
- Scrum master: he guides the team by assuring the right choices are being made. He is responsible for arranging the meetings. Rolf is our Scrum master.

The Scrum process

There are several steps involved in the Scrum process. First, we will create a product backlog. This is a list with the functional demands the product owner has and it contains the items we still have to do. In total, we have 5 sprints. At the end of each sprint, we will deliver a part of the final product. The duration of each sprint is two weeks. At the beginning of each sprint, we will create a sprint backlog. This backlog describes the functional demands, divided in each subtask for this sprint.

Each morning, we will start the day with the daily Scrum. This is a short meeting where every member of the development team answers the following question:

- What have you done yesterday?
- What are you going to do today?
- Are there any problems you ran into?

At the beginning of each sprint, we start with a meeting. This meeting is attended by all members of the team and the supervisor. The purpose of this meeting is to evaluate the last sprint and decide on the tasks that have to be done during the next sprint.

A.2.2 MoSCoW

MoSCoW is a technique that can be used to place importance on the delivery of each requirement. During each sprint, we will classify the features in one category. The MoSCoW model has the following categories:

- Must have: the requirement must be part of the final product to be considered a success.
- Should have: the requirement has a high priority and should be in the final product.
- Could have: the requirement is desirable but not necessary.
- Would have: the requirement is not implemented in a given release but is considered as a requirement in the future.

At the start of each sprint, we evaluate the goals we want to achieve that sprint. After that, we classify each goal into one of the categories above. Since we do not have everything clear at the start of the project, it could happen that we prioritize the goals differently during each sprint.

A.2.3 Tools

For this project, we will use various tools, both hardware and software. First of all, we will be using our own laptops for the development of the software. We will develop our software on the Ubuntu platform. We are also using Android phones that we can rent from the Tribler team.

If we look at the software, we will make use of the Android SDK and NDK. The Android SDK will allow us to build .apk files. The NDK allows to implement parts of an Android application in C or C++. To be able to run Python code on an Android device, we will use the Python for Android library.

A.2.4 Planning

Our planning can be found on GitHub. As for now, we have four milestones:

- 02-05-14: the literature research should be done and a report about the read literature should have been written.
- 09-05-14: we should be able to compile the TGS for Android project and run it on an Android device.
- 30-05-14: we should be able to send a packet between two Android devices over anonymous tunnels.
- 13-06-14: a GUI for testing purposes should be designed and created.
- 27-06-14: the end presentation of our project.

A.3 Project structure

In this section the administrative aspects of the project are described.

A.3.1 Members

The members of the project are Rolf Jagerman, Laurens Versluis and Martijn de Vos. All members will work 40 hours per week to ensure the mandatory 15 EC per student are utilized. The division of labor is evenly distributed across all activities (analysis, documentation, implementation, etc.).

Contact Information
Rolf Jagerman - R.M.Jagerman@student.tudelft.nl
Laurens Versluis - L.F.D.Versluis@student.tudelft.nl
Martijn de Vos - M.A.Devos@student.tudelft.nl

A.3.2 Reporting

Weekly meetings with the client will be held in person. All documentation of the project will be written in LaTeX and will be provided as a PDF. The project material, including source code and documentation, will be available throughout the project on the AT3 GitHub repository[2].

A.4 Quality assurance

To assure a good quality of the delivered product, several agreements are made about which methods should be used. In particular we look at testing, code review and version control.

A.4.1 Testing

All written software will be tested using Python unit tests. Additionally, test code coverage is provided to ensure a majority of the code has been thoroughly tested.

A.4.2 Code review

All written code will be reviewed by at least one team member before pull requests are accepted. This will ensure the code is comprehensible, working and correct. Additionally our code will undergo a complete source review by SIG (Software Improvement Group).

A.4.3 Version control

To maintain a good overview and history of the code we write, we will use a version control system. All our code will be stored on GitHub and therefore use the Git version control system.

[2] www.github.com/rjagerman/AT3

Appendix B

Dependency graph of AT3

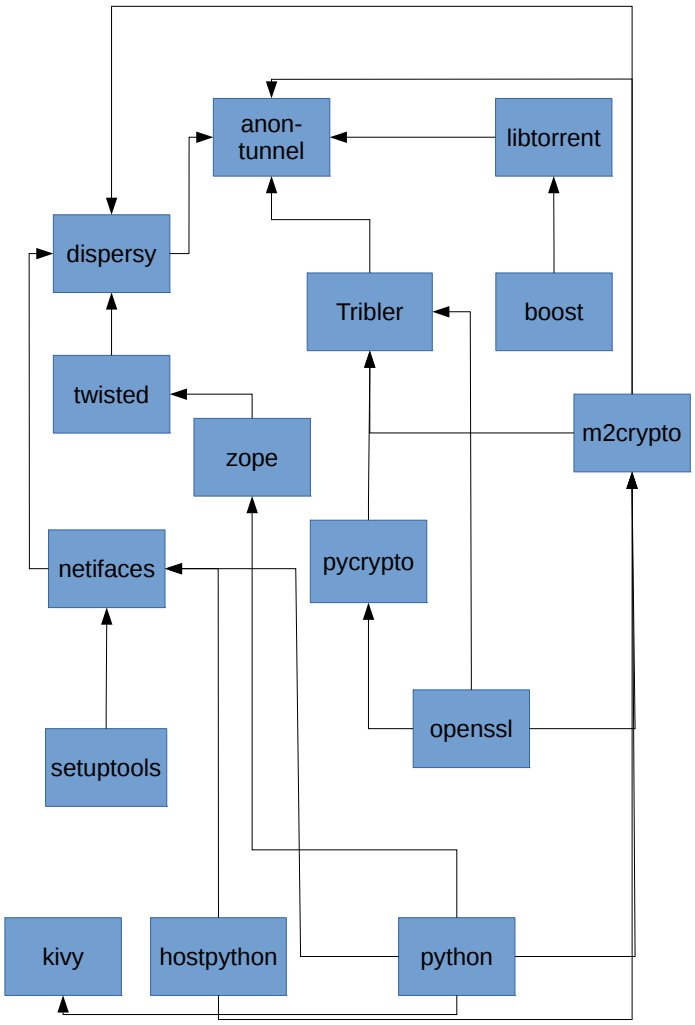

Figure B.1: The dependency graph of the AT3 application

Appendix C

Original project description

Project description[1]

In this project, a prototype of an anonymous P2P-based video-on-demand mobile application will be realized, that will allow users to search for videos on the Internet. Once the user has found a video to his or her liking, and issued a play command, the application will play the video by means of streaming it through a P2P-based network. TOR is the market leading solution for anonymous web access. The Tribler team has created a P2P implementation of the TOR prototocol and enhanced it with NAT puncturing and decentralized the directory service. Your task is to take the existing operational Python code, initial Android code and make it ready for deployment towards the Google Android Play app marketplace. Challenges are getting M2Crypto crypto libraries operational, Python wrappers, VLC playback integration, multi-device compatibility and User Interface realization.

Appendix D

Compiling libtorrent

On of the major goals of our project was to create a stable version of libtorrent capable of running on an Android device. To do this, we have to cross compile libtorrent for Android which contains an ARM chipset. To compile libtorrent, we make use of a custom toolchain we created from the tools that are bundled with the Android NDK.

D.1 Setting up the environment

To cross compile libtorrent for Android, we first need to create a custom toolchain. Before we do that, please make sure you have the Android NDK and SDK installed. We compiled with NDK r9d 32-bit and we also installed SDK version 14. We are not sure whether other versions of the NDK / SDK are working. The compilation has been executed on a 64-bit Ubuntu 14.04 machine. Please note that if you are compiling on a 64-bit machine, you need some 32-bit support packages which can be installed with the following command.

```
sudo apt-get install ia32-libs
```

This should install the required 32-bit libraries needed for the cross-compilation. The next step is to install Python for Android because we need the Python library from it. We cannot use the Python distribution the system is using because that is compiled for an incompatible architecture. The Python for Android framework comes with a distribute script that creates a distribution with the supplied libraries. Executing the following commands from the Python for Android folder should build the Python distribution that will eventually run on the device.

```
export ANDROIDSDK='<path to your SDK folder>'
export ANDROIDNDK='<path to your NDK folder>'
export ANDROIDNDKVER=r9d
export ANDROIDAPI=14
./distribute.sh -m kivy
```

An additional step is required: copy the `pyconfig.h` file to the `Include` folder in the build directory of `pythoninstall/Python2.7.2`. This file is required by the compilation process of the Python bindings.
The next step is to install the custom toolchain we create from the Android NDK. Execute the following command. Your toolchain location can be anywhere on the computer but it is recommended to install it in a location where you can access it easily.

```
<path to your NDK folder>/build/tools/make-standalone-toolchain.sh \
--platform=android-14 --install-dir=<your new toolchain location>
```

This command should generate a custom toolchain in the location specified by the install-dir argument.

Several export variables are required for compiling Boost and libtorrent, these are listed below. The `$ANDROIDNDK` variable should already be defined from the previous steps.

```
# Custom paths
export ANDROIDNDK='<path to your NDK folder>'
export SYSROOT=$ANDROIDNDK/platforms/android-14/arch-arm
export PYTHON= \
<path to your Python for Android>/build/python/Python-2.7.2/hostpython
export PYTHON_CPP_FLAGS= \
"-I<path to your Python for Android>/build/python/Python-2.7.2 \
 -I<path to your Python for Android>/build/python/Python-2.7.2/Include"

# Custom ARM toolchain
export SYSROOT=$ANDROIDNDK/platforms/android-14/arch-arm
export PATH=/usr/local/gcc-4.8.0-arm-linux-androideabi/bin:$PATH
export CC=arm-linux-androideabi-gcc
export CXX=arm-linux-androideabi-g++
export CROSSHOST=arm-linux-androideabi
export CROSSHOME=/usr/local/gcc-4.8.0-arm-linux-androideabi
```

It is recommended to create a shell script file with these exports and load it using the source command.
You should now have a custom toolchain ready to be used and the right environment variables set up.

D.2 Compiling Boost

Libtorrent is using the Boost library for threading and memory management. This means that we first need to cross compile Boost for Android. The Boost for Android project on GitHub looked promising but we decided to use our custom toolchain and manually compile Boost so we have more control over the compilation process. First, download the official Boost source code and save it to your computer (we used Boost 1.55). Navigate to the folder containing Boost and execute the following command.

```
./bootstrap.sh
```

This command will execute the bootstrap and configures the Boost building environment for the compilation. Libtorrent is using shared pointers that are using spinlocks, however, these spinlock shared pointers are not working correctly on embedded devices. To disable spinlock mechanics, some additional compilation flags are needed. Edit the `user_config.jam` file in your Boost directory, located in `build/tools/v2` to contain the following:

```
using gcc : android : arm-linux-androideabi-g++ :
  <compileflags>-DBOOST_SP_USE_PTHREADS
  <compileflags>-DBOOST_AC_USE_PTHREADS
;
```

Now we can build the required Boost libraries with `b2`.

```
./b2 toolset=gcc-android architecture=arm link=static threading=multi \
 --with-system --with-filesystem --with-python install \
 --prefix=<path to your custom toolchain>
```

The `b2` command builds and installs the system, filesystem and Python libraries in the directory specified by the prefix. We create a static library with support for multithreading. The include files and binaries are copied to your custom toolchain.
At this point, Boost should be installed in your custom toolchain. Please verify the presence of

Boost in your toolchain because these files are needed for the compilation of libtorrent in the next step.

D.3 Compiling libtorrent

With the Boost libraries compiled and in place, we are now able to compile libtorrent. First download the official libtorrent-rasterbar library from their website [12]. At time of writing, the second release (RC2) candidate of libtorrent is used. Navigate to the folder where libtorrent is located and execute the following command:

```
./configure --host=$CROSSHOST --prefix=$CROSSHOME \
--with-boost=$CROSSHOME --with-boost-libdir=$CROSSHOME/lib \
--enable-static --disable-shared --enable-debug --enable-logging \
--enable-python-binding
```

This command configures the makefiles and configuration scripts. We specify that we are using our Android host and we tell the script where Boost can be found. We also specify that we are creating a static library and not a shared library. Debug symbols and logging are turned on and we pass the enable-python-binding flag to the script so we can use the libtorrent library in Python. We also need to set compiler flags again so Boost is not using the spinlock mechanics.

```
export CFLAGS="-g -DBOOST_SP_USE_PTHREADS -DBOOST_AC_USE_PTHREADS"
export CXXFLAGS=$CLFAGS
```

Libtorrent is now ready to be compiled. Execute the following commands to start the compilation process.

```
make clean
make
make install
```

For faster execution, you can pass the `j` flag to compile on multiple cores. For instance, if your computer has four cores, you can execute `make -j 4` to speed up the compilation process.
The libtorrent shared object file is now located in your custom toolchain and is ready to be used in Python for Android or other (Android) applications. We have created our own recipe that copies the precompiled library to the site-packages directory of Python for Android. Due to limitations in time, the compilation process is not executed in our recipe script.
The final libtorrent shared object file can also be used in a native Java Native Interface (JNI) application however, to decrease the final application size, it is recommended to leave out the Python bindings. Compilation without the Python bindings can be achieved by leaving out the `enable-python-binding` flag when running the configuration script of libtorrent. When compiling Boost, the `with-python` flag is not necessary anymore.
A basic JNI application that loads and uses libtorrent can be found on our GitHub repository we created for this purpose [3].

Bibliography

[1] Bepsys anonymous hd video streaming for tablets. http://bepsys.herokuapp.com/projects/view/4. Accessed: 2014-06-18.

[2] Forbes android dominates market share, but apple makes all the money. http://www.forbes.com/sites/tonybradley/2013/11/15/android-dominates-market-share-but-apple-makes-all-the-money/. Accessed: 2014-04-29.

[3] GitHub hellolibtorrent, a sample project using libtorrent for android. https://github.com/rjagerman/HelloLibtorrent. Accessed: 2014-06-18.

[4] GitHub me too crypto. https://github.com/martinpaljak/M2Crypto. Accessed: 2014-04-29.

[5] GitHub the global square. https://github.com/GlobalSquare. Accessed: 2014-04-29.

[6] GitHub tribler pull request #525. https://github.com/Tribler/tribler/pull/525. Accessed: 2014-04-29.

[7] Google live encoder settings, bitrates and resolutions. https://support.google.com/youtube/answer/2853702?hl=en. Accessed: 2014-06-16.

[8] Google play instagram. https://play.google.com/store/apps/details?id=com.instagram.android. Accessed: 2014-05-12.

[9] Google play whatsapp messenger. https://play.google.com/store/apps/details?id=com.whatsapp. Accessed: 2014-05-12.

[10] Openssl website. http://www.openssl.org. Accessed: 2014-04-29.

[11] Roarmag the global square: an online platform for our movement. http://roarmag.org/2011/11/the-global-square-an-online-platform-for-our-movement/. Accessed: 2014-04-29.

[12] Sourceforge libtorrent rasterbar. http://sourceforge.net/projects/libtorrent. Accessed: 2014-06-18.

[13] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[14] Roger Dingledine and Steven J Murdoch. Performance improvements on tor or, why tor is slow and what we're going to do about it. *Online: http://www. torproject. org/press/presskit/2009-03-11-performance. pdf*, 2009.

[15] David M Goldschlag, Michael G Reed, and Paul F Syverson. Hiding routing information. In *Information Hiding*, pages 137–150. Springer, 1996.

[16] Richard Grisenthwaite. Armv8 technology preview. http://www.arm.com/files/downloads/ARMv8_Architecture.pdf, 2011.

[17] Rolf Jagerman, Wendo Sabée, Laurens Versluis, Martijn de Vos, and Johan Pouwelse (course supervisor). The fifteen year struggle of decentralizing privacy-enhancing technology. *CoRR*, abs/1404.4818, 2014.

[18] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable onion routing with torsk. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 590–599. ACM, 2009.

[19] Jun Yang. Smartphones in use surpass 1 billion, will double by 2015. http://www.businessweek.com/news/2012-10-17/smartphones-in-use-surpass-1-billion-will-double-by-2015, 2012. Accessed: 2014-06-17.

[20] Niels Zeilemaker, Boudewijn Schoon, Johan Pouwelse, Rahim Delaviz Aghbolagh, Niels Zeilemaker, Johan Pouwelse, Dick Epema, Niels Zeilemaker, Mihai Capota, Arno Bakker, et al. Dispersy bundle synchronization. *IFIP Networking 2013*, 4820:203–214, 2013.