UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

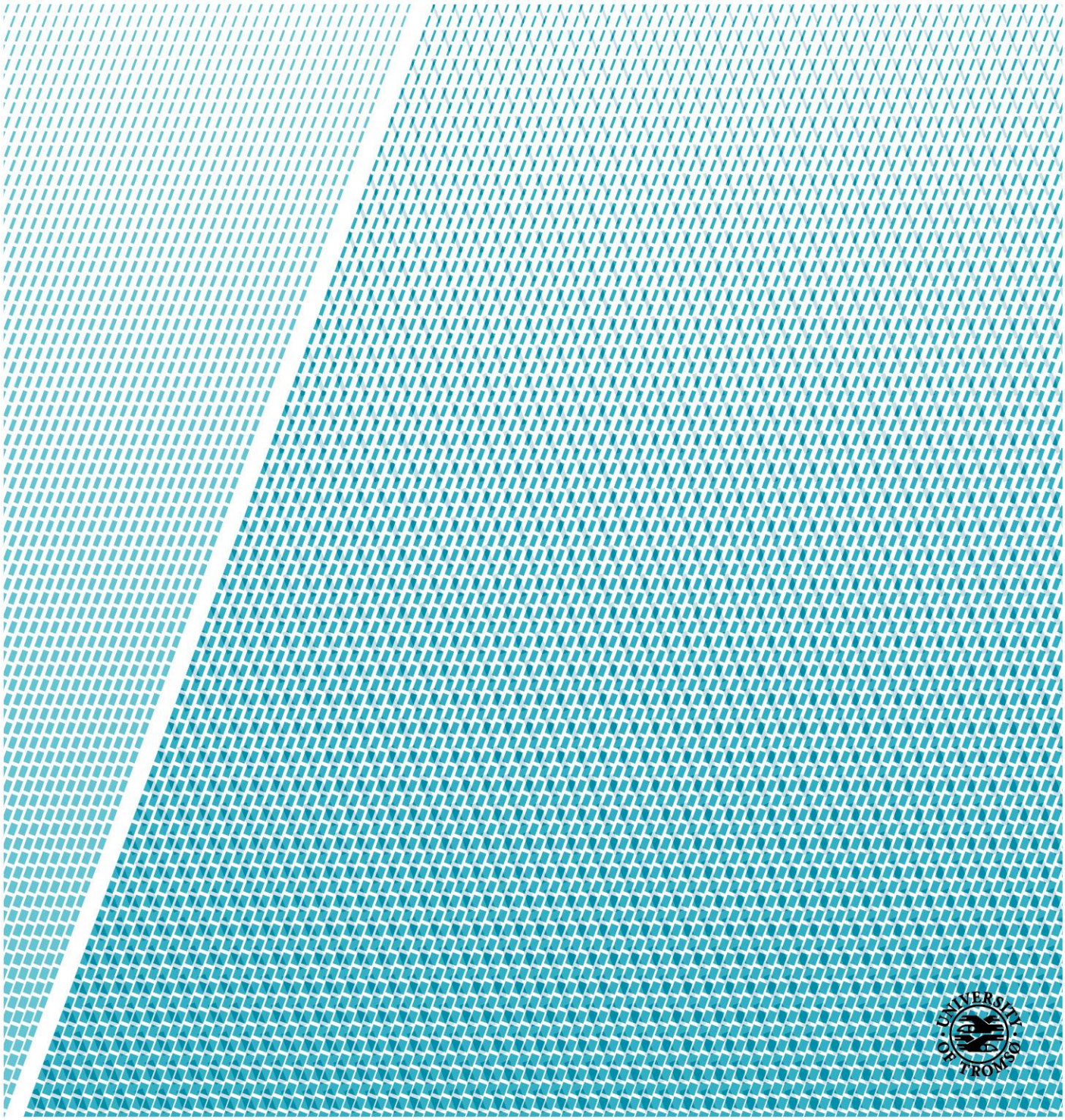
Swirlwave

Cloudless Wide Area Friend-to-Friend Networking Middleware for Smartphones

—

Jo Inge Arnes

INF-3990 Master's Thesis in Computer Science—May 2017



“We will become all-seeing, all-knowing.”
—Dave Eggers, The Circle

Abstract

Swirlwave is a middleware that enables peer-to-peer and distributed computing for Internet-connected devices with the following characteristics: The devices lack publicly reachable IP addresses, they can be expected to disconnect from the network for periods of time, and they frequently change network locations. This is the typical case for smartphones.

The middleware fits into the friend-to-friend subcategory of peer-to-peer systems, meaning that the overlay network is built on top of already existing trust relationships among its users.

It is independent of clouds and application servers, it has built in encryption for confidentiality and authentication, and it aims to be easily extensible for new applications. The solution described in the thesis was implemented for smartphones running the Android operating system, but its principles are not limited to this.

Acknowledgements

I would like to thank professor Randi Karlsen for being the thesis supervisor, and Jan Fuglesteg for being an invaluable student advisor. I would like to express my gratitude to Kjersti for her support, and my son for always being an inspiration.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Two trends have changed the landscape of personal computing, namely smartphones and cloud computing.

It may not be immediately obvious that smartphone apps have fundamental differences from traditional PC-applications. The differences are not so much observed in functionality, as in software architecture. The shift is that data that used to be stored and processed on local computers, now more often reside in a cloud, and users access their data through apps connected to the Internet.

This can pose challenges to privacy and consumer influence. Data breaches that affect millions, unclear ownership of personal data, and the surveillance economy, are all relevant in this respect. This is one of the motivations for proposing an alternative to the cloud based approach to data sharing.

Another motivation is to make it easier to use smartphones in peer-to-peer or distributed systems. Smartphones are not merely phones, they are computers. The smartphones of today have as much processing power, memory, and storage capacity as a typical desktop PC a decade ago[1,2,3]. Smartphones are increasingly popular, while PC sales decline [1]. Considering this, it seems sensible to explore ways to harness more of these devices' potential.

A third motivation is the Internet's inherently distributed architecture. The success of the Internet was probably possible due to its openness and lack of centralized ownership. In some ways, the move towards cloud computing can be seen as the opposite. Three large companies, Amazon, Google, and Microsoft, are becoming more and more dominant in the cloud provider market [2]. Undoubtedly, there are many advantages to cloud computing, but as popularity greatly increases a danger exists that the Internet may become more centralized and less open, as server software and data are moved to datacenters owned by a handful of companies.

There is little focus on using smartphones as *servents* in wide area peer-to-peer systems. The paradigm of today is that smartphone apps are clients to a backend in the cloud.

1.2 Problem and Goal

1.2.1 Statement of the Problem

Smartphone apps that let users share and communicate with others over the Internet are commonly backed by cloud services. In this model, communication does not go directly between two smartphones, but must pass through the cloud datacenters. This can be a source of problems concerning privacy.

[1] http://www.samsung.com/global/galaxy/galaxy-s8/specs
[2] http://vbcity.com/forums/t/133493.aspx
[3] https://forums.overclockers.co.uk/threads/average-pc-spec.17644377

On the technical side, smartphones have become relatively powerful computers. Considering that there are billions of smartphones in the world, would it not be sensible to make better use of their resources instead of just using smartphones as clients to cloud backends?

1.2.2 Goal

The goal of the thesis is to research, design, and implement an alternative to the traditional, cloud centric approach to smartphone app development.

The specific research question is: How can a middleware be designed to enable wide area peer-to-peer communication for smartphones—and other devices that frequently change networks and lack publicly reachable IP addresses—without the need for clouds or application servers as middlemen for storing, processing, and sharing data?

An additional goal is to make it easier to incorporate smartphones as nodes in a peer-to-peer or distributed system, so that the storage and processing capacities of the smartphones can be utilized as part of a bigger whole.

The focus is on suggesting an alternative model for developing smartphone apps, to design and implement a realistic (though possibly incomplete) version, and then explore it by experimentally testing and measuring its properties, before concluding whether it the solution is practicable for use.

1.2.3 Non-Goals

The goal is to create a proof-of-concept, not a production ready system. Neither is it a goal to implement complete applications using the middleware.

1.3 Approach

The system designed for this thesis is a middleware enabling smartphones to communicate directly in a peer-to-peer fashion over the Internet. This is not as trivial as it may seem at first glance, since smartphones usually lack publicly reachable IP addresses and often change networks. Most smartphone apps of today depend on clouds or application servers as middlemen, whereas the system described in this thesis does not. The system is implemented for the *Android*[4] operating system as an app running a background service.

The system's working title is Swirlwave.

People carry smartphones with them. Consequently, peers are expected to change networks at unpredictable times. When a peer changes its address, this must be announced to its peers. If not, they will lose contact. Each installation of the app keeps a contact list of peers. A peer is simply a smartphone running the Swirlwave middleware. Peers have addresses that can be connected to, but these addresses are not IP addresses. *Tor hidden service* [3] addresses are used instead, because these addresses are reachable without the need for a public IP address. One of the systems main challenges is to keep the peer contact lists up to date with correct addresses.

The system is designed as a middleware, which means that it is not a complete application by itself, but that it facilitates communication between applications. For this to be possible, the system must be

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

extensible in such a way that existing applications and libraries easily can make use of it. This is achieved by creating two proxies, one for the client and one for the server side. Any client library using TCP connections can connect to the locally running client proxy, and the middleware will automatically route the traffic to the correct peer. On the other end, a server side proxy will accept the incoming connection and route its traffic to the correct service running locally on the receiving peer. This is implemented in a location and protocol transparent manner. It is location transparent in the sense that an application connecting to a peer via the middleware does not need to know the peer's address. It only needs to connect to a local port. The system is protocol transparent in the sense that an application does not need to know the specific protocol used to connect to a peer. The underlying protocol in the implemented system is SOCKS4a, but the peer only connects locally by regular TCP.

The app's background service manages a Tor proxy and publishes hidden services as an integral part of communicating between peers. Tor hidden services are encrypted end to end, which provides communication confidentiality.

Each installation is additionally equipped with its own keypair for public-key encryption. This is used for authentication purposes. It is also used for ensuring confidentiality, integrity and non-repudiation of data when communicating over other channels.

All peers can be clients and servers at the same time; they can expose several services for peers to consume, and they can be clients to services published by other peers.

As an example, one type of service can be a web server. To connect to a web server, one needs a client software that understands the HTTP protocol. Another type of service can be a message queue. To connect to a message queue, a different type of client software is needed, one that knows the protocol of the message queue. Thus, servers and clients are two sides of the same story. The correct client software is needed to make use of a service on a server.

For the system in this thesis, a plugin approach is taken. It is possible to implement a service and its corresponding client software as an app that can be installed as a package separate from the Swirlwave middleware. The app can register itself with the installed instance of Swirlwave, and peers can then discover that the given peer exposes that service. The idea is that if a peer does not have the plugin installed, and thus is missing the necessary client software, it can potentially download the plugin package from Google Play. The latter is not implemented for the version of the system in this thesis.

1.4 Contribution

The main contribution of the thesis is to design and implement wide area, location transparent peer-to-peer communications to devices that often change network locations or disconnect from the network, and that lack reachable IP addresses. The focus of the thesis is mainly on smartphones. A system integrating this functionality is designed and implemented as a middleware on which other applications can be built.

A variety of apps for smartphones that run as servers exist, for example web servers for Android, but they only work as part of a local area network. Clients must be on the same local area network as the server to connect. With such a constraint, much of the point of using a smartphone disappears. A desktop or laptop will probably do a better job. What Swirlwave adds is enabling smartphone server

apps to be available outside the local area network, even if a user carries the smartphone with her and it changes networks.

The system is friend-to-friend based. Only devices that are explicitly registered as friends can communicate directly as peers. This is to make it possible to define smaller overlay networks of pre-approved devices, which for instance is important if the system is to be usable for building a distributed system using a pool of trusted devices.

The use of the Tor hidden service protocol make it possible to reach smartphones outside local area networks. Orbot[5] is an existing official app for smartphones that makes this possible, but it has no mechanism for changing and announcing addresses when the smartphone changes its location. Because of how the Tor hidden services work, clients are not able to connect anymore when the smartphone changes its location. Tor does not include any means to announce new addresses to clients, and no peer contact lists. Also, there is no protocol transparency, so a client connecting to the smartphone server app must understand the protocol used by Tor. The system in this thesis is designed to solve these problems transparently as a middleware.

To programmatically use Tor hidden services an existing library is used. This library was originally made for a Microsoft funded research project named *Thali*[6]. Because of the problems with hidden services and smartphones' frequent change of network locations, the Thali Project states on their site, "Right now using Tor Hidden services is hard for us because HSs are clearly designed for stationary services not mobile ones."[7] The Thali Project has not solved the problems, but merely suggests changes to the Tor hidden services protocol. At the time of this thesis, the library seems abandoned, and the code has not been updated in years. It is clear from the project homepage that they instead have chosen to concentrate on communications over BLE, Bluetooth, and Wi-Fi direct—none of which are wide area communications. The system in this thesis tries to overcome the same hurdles by accepting that addresses have to be changed when network locations change and by adding a system for keeping peers up to date with correct addresses.

1.5 Ethical Considerations

Privacy is a human right. In the *Universal Declaration of Human Rights*[8], Article 12, it is stated:

> *No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.*[8]

Privacy is important, not solely for the individual, but for the society. As a reaction to the 2013 disclosures of U.S. electronic surveillance, based on leaks by former NSA contractor Edward Snowden, Brazilian President Dilma Rousseff said in a speech in the U.N. General Assembly: "Without the right of privacy, there is no real freedom of speech or freedom of opinion, and so there is no actual democracy." [4] Without privacy, democracy cannot exist.

---

[5] See section 3.2

[6] See section 3.3

[7] http://thaliproject.org/ThaliAndTorHiddenServices

[8] http://www.un.org/en/universal-declaration-human-rights

There are good reasons for why elections are held by secret ballots, so that votes are anonymous.[9] Public voting would leave voters vulnerable to retribution, bribery, discrimination, peer pressure, groupthink, and other undesirable situations that would hurt free and fair elections. This would be bad for democracy. For similar reasons, we need privacy to communicate freely with others.

On the other hand, privacy comes with a grave cost. It is well known that the strong privacy provided by some systems is being misused by criminals.[10] This is not to be taken lightly when considering the sufferings that certain types of crimes inflict on their victims.

Therein lies the dilemma: How do we choose between protecting privacy and preventing crime?

Perhaps the answer is not either-or. Perhaps a tradeoff can be accepted. A system could be designed to ensure privacy to an extent that makes mass surveillance too impractical and expensive, but at the same time make it insecure enough for authorities to conduct targeted investigations of serious crimes. The intended effect is that criminals, knowing that the security is too weak to hide their criminal activity from investigation, would avoid using the system. One could think of such a system as "*felony unfit"*. At the same time, innocent users would know that the level of privacy is good enough to make it infeasible or too costly to monitor users on a large scale or spy on single individuals without a very good reason.

Schneier [5] argues that mass surveillance does not prevent crime and that targeted investigations are far more effective:

> *Adversaries vary in the sophistication of their ability to avoid surveillance. Most criminals and terrorists—and political dissidents, sad to say—are pretty unsavvy and make lots of mistakes. But that's no justification for data mining; targeted surveillance could potentially identify them just as well. The question is whether mass surveillance performs sufficiently better than targeted surveillance to justify its extremely high costs. Several analyses of all the NSA's efforts indicate that it does not.*

The system in this thesis is based on creating networks of trusted friends and acquaintances. The system does not rely on central servers that can be monitored, and the different overlay networks do not need to be interconnected at all. These factors contribute to making large scale monitoring of the system difficult. Nevertheless, communication and data are not encrypted locally on a single device. By employing more sophisticated techniques for targeted surveillance, software could stealthily be installed on a specific smartphone to eavesdrop on communications and obtain information about a specific criminal network.

1.6 Organization

The rest of this thesis is organized as follows: Important background information is presented in chapter 2. This is followed by related work in chapter 3. The architectural design is presented in chapter 4. Implementation specifics are presented in chapter 5. Experiments, results, and discussions are found in chapter 6. Potential future work is presented in chapter 7. The thesis is concluded in

[9] http://aceproject.org/ace-en/topics/ei/eif/eif09/eif09a

[10] http://www.bbc.co.uk/guides/z9j6nbk

chapter 8. Cited work comes after chapter 8, followed by the appendices, where a selection of source code listings can be found.

2 Background

This chapter presents background information that is relevant for understanding the motivations, goals, and design decisions made for the system in this thesis.

2.1 The Rise of Smartphones

This section gives a background on smartphones, how they are used, how applications (*apps*) typically are designed, and some problems concerning privacy.

Only ten years have passed since the first *iPhone*[11] saw the public light in 2007. One year later, the first Android phone was released. As of today, Android dominates the marked with sales surpassing a billion devices per year [6].

With smartphones and improved cellular networks, people no longer sit steadfast in front of a PC to be online. The Internet can be accessed from wherever the users are, even while being on the move. In fact, fifteen percent of American smartphone owners have said they have limited other options for accessing the Internet [7]. And people use their smartphones extensively. In 2015 forty-six percent of American smartphone owners said it was something they could not live without [7].

People are using their smartphones' Internet features in a wide range of areas in their personal lives. Examples are online banking, keeping up with the news, education, career, looking up health information, sharing or coordinating with others, navigating in traffic, just to mention a few. One important use is sharing personal pictures and videos. Another is participation in communities or social networks. In the U.S. sixty-seven percent used their phone to share pictures, videos, or commentary about events happening in their community [7]. Many also use their smartphone for navigation. Nearly one-in-three smartphone owners in the U.S. frequently used their phone for navigation or turn-by-turn driving directions [7]. This is mentioned because finding one's location on a map by using GPS and other positioning technologies[12] has become commonplace, a feature that was rare before the smartphone era.

The way a user accesses the services mentioned above is generally through apps. Most of these apps are backed by an online service. Even though smartphones generally have a relatively decent storage and processing capacity, these apps upload and store data in the service provider's datacenters or cloud services. This makes it possible to access the data, for instance pictures and videos, from other devices. It also makes it possible to share the data with others, for example by sending a mobile camera picture of oneself to friends through *Snapchat*[13]. In this case, the picture is uploaded to Snapchat's servers, and the recipient friends are notified about the picture. The friends can then see the picture by viewing it in their Snapchat app, which downloads the picture from the Snapchat datacenter.

It is important to understand that data is not sent directly from one smartphone to another, but rather the data goes via a datacenter. In addition to storing data that the user uploads, friends list and many other types of data concerning the app are managed there. The provider of the app usually gathers

[11] http://www.apple.com/iphone
[12] https://en.wikipedia.org/wiki/Mobile_phone_tracking
[13] https://snapchat.com

metadata about user activities, such as who they communicate with, when, where, how often, about what, and so on.

2.1.1 Privacy Concerns

Sir Tim Berners-Lee is an English computer scientist, best known for being the inventor of the World Wide Web. On the 28th anniversary of his original proposal, March 12, 2017, he published an open letter on the Web Foundation's website [8]. In the letter, he lists three challenges for the web. He writes that over the last 12 months, he has become increasingly worried about three new trends, which he believes we must tackle in order for the web to fulfill its true potential as a tool which serves all of humanity. The first challenge is that we have lost control of our personal data. He writes that the current business model is to offer free content in exchange for personal data. As our data is then held in proprietary silos, out of sight to us, we lose out on the benefits we could realize if we had direct control over this data, and chose when and with whom to share it. What's more, we often do not have any way of feeding back to companies what data we'd rather not share—especially with third parties. Under the same section, he also addresses the negative impacts of widespread data collection by companies. Through collaboration with—or coercion of—companies, governments are also increasingly watching our every move online. This, he continues, is not only harmful in repressive regimes. Even in countries where we believe governments have citizens' best interests at heart, watching everyone, all the time is simply going too far.

The challenges associated with losing control over personal data also applies to smartphone apps. Some topics concerning privacy is elaborated on in this thesis in the following subsections, 2.1.2–2.1.4.

2.1.2 Ownership of Data

When storing or sharing data online via a company's datacenter or cloud services, ownership of the data can be unclear. In October 2015, there were news articles claiming Snapchat had changed their terms and conditions, so that the company formally owned the pictures and videos uploaded by the users [9]. At the time, Snapchat had about 100 million daily users [10]. There was some confusion about whether this was correct, but there have been similar worries concerning Facebook and other social media services.

2.1.3 Data Breach Incidents

There have been several incidences where sensitive personal information has leaked from datacenters and cloud services. October 2014, hundreds of thousands of photos originating from Snapchat were leaked [11]. That same month photos of very private character showing some of the most famous actresses and female artists were leaked from iCloud and published on the Internet, which caused a lot of distress for the victims [12].

Numerous popular services have been hacked the last years, including *LinkedIn*[14], *Adobe*[15], *Tumblr*[16], *MySpace*[17], and *Dropbox*[18]. Secret passwords, birthdates, and credit card numbers that were stolen as part of these incidents, and that affected millions of users, have later been found in openly available databases on the Internet. February 2017, the Norwegian newspaper VG [13] brought to light that emails and password belonging to 727 Norwegian politicians, members of the government, employees at departments and embassies had been found by searching in such databases. This included password to accounts of the prime minister, the minister of health, and the EU-minister. The situation is hardly unique to Norway, and it is not difficult to imagine consequences for the exercise of internal politics in a country, as well as in international matters, bilateral negotiations, and matters of national security.

Active attacks are not always the cause of data breaches. February 2017, a Google engineer discovered a massive web leak caused by a defect in the company *Cloudflare*'s[19] systems, which hosts and serves content for 6 million sites. It was soon dubbed *Cloudbleed*[20]. At the time, it was said that the Cloudbleed defect could have leaked secrets from major web companies, like *Uber*[21], *FitBit*[22], and *OKCupid*[23]. No active attack was needed to obtain secret data. For instance, anyone visiting a Cloudflare-hosted web site could end up having someone else's data in their web browser cache [14]. In retrospect, investigations conducted by Cloudflare showed that the impact of the leak probably was lower that first feared. Cloudflare's CEO Matthew Prince said in an interview with Forbes [15] that they got lucky, and that it could have gone exceptionally bad. Though the impact was likely minimal, the fact that companies with non-vulnerable sites were still exposed highlighted a real downside in centralizing so much of the web, he admitted.

2.1.4 Surveillance Economy

Most of the companies behind the popular free apps have a business model where they actively monitor and gather information about their users. This can be sold to data brokers. User data is valuable because it can be analyzed with advanced big data algorithms and used in marketing and advertising. Data brokers who buy this data, can use or resell it to other companies. [5, 16]

The vast array of user data that is managed and sold by data brokers can be hard to grasp for most people. To shed some light on the subject, it can be mentioned that the data broker company *Acxiom*[24] has information about 700 million consumers worldwide with over 3000 data segments for nearly every consumer. [16]

---

[14] https://linkedin.com
[15] http://adobe.com
[16] https://www.tumblr.com
[17] https://myspace.com
[18] https://www.dropbox.com
[19] https://www.cloudflare.com
[20] https://en.wikipedia.org/wiki/Cloudbleed
[21] https://www.uber.com
[22] https://www.fitbit.com
[23] https://www.okcupid.com
[24] https://www.acxiom.com

In a survey by the Norwegian Data Protection Authority[25] (Datatilsynet) it was found that seventy-nine percent of those who responded either strongly or somewhat agreed that the collection, analysis, and sharing of personal data by online agencies for commercial purposes made them uncomfortable. [17]

2.2 Client-Server

In section 2.1 it was explained that smartphone apps often are backed by an online service. This is a type of client-server architecture.

A system with a client-server architecture is split in a client part and a server part, each with distinct roles. The server part, or server side, of the system is a central that provides services to clients. A client is a software that contacts the server for its services.

In a typical scenario, a server can provide services to many clients at a time. The clients contact the server over a network connection. The client software is run on the user's computer, and the server software runs on a dedicated computer—a server—somewhere in the network. Note that the terms client and server often are used interchangeably for the software and the hardware depending on the context under discussion. The server does not contact the client, it only acts as a response to a request from a client. For instance, a web server resides on a server reachable from the Internet. It serves a web site. Users can access the site by starting a browser on their computer, and request web pages from the server. In this example, the browser is the client.

The term backend is used for servers that provide clients with essential services that are needed for the client to work. It is very common to split an application in a frontend and a backend. The frontend has the responsibility for providing user interfaces and processing that must be done locally, and it acts as a client to a backend. The backend is often a server that has the responsibility for heavier processing, storing data, and sharing of data.

2.3 Cloud Computing

The online services and datacenters in section 2.1 are more specifically cloud services running in a cloud. In this section, cloud computing is described.

Cloud computing is truly transforming large parts of the IT industry. It affects many aspects of computing, from software to hardware, from development to IT management. Instead of companies owning hardware, managing network infrastructure, and running IT systems by themselves, the trend is to outsource to large datacenters owned by third parties. Resources can be provisioned on demand and payed per use. Amazon, Google, and Microsoft are now cementing their dominance as cloud providers, at the expense of smaller companies. [2].

Microsoft explains cloud computing this way on their Azure site:

> *Simply put, cloud computing is the delivery of computing services—servers, storage, databases, networking, software, analytics, and more—over the Internet ("the cloud"). Companies offering these computing services are called cloud providers and typically*

[25] https://www.datatilsynet.no/English/

charge for cloud computing services based on usage, similar to how you're billed for water or electricity at home. [18]

They continue with,

You're probably using cloud computing right now, even if you don't realize it. If you use an online service to send email, edit documents, watch movies or TV, listen to music, play games, or store pictures and other files, it's likely that cloud computing is making it all possible behind the scenes. [18]

According to the *National Institute of Standards and Technology*[26] (NIST) [19], cloud computing has five essential characteristics: On-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. The types of services provided by clouds can usually be classified as: Infrastructure-as-a-service (IaaS), platform as a service (PaaS), or software as a service (SaaS).

In the report "Above the Clouds: A Berkeley View of Cloud Computing", the scope of the term cloud computing is defined as this:

Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud. When a Cloud is made available in a pay-as-you-go manner to the general public, we call it a Public Cloud; the service being sold is Utility Computing. We use the term Private Cloud to refer to internal datacenters of a business or other organization, not made available to the general public. Thus, Cloud Computing is the sum of SaaS and Utility Computing, but does not include Private Clouds. [20]

2.3.1 Social Media

Some of the world's most popular apps are social media apps. *Facebook*[27], *Twitter*[28], *YouTube*[29], *LinkedIn*[30], *Flickr*[31], *Snapchat*[32], and *Instagram*[33] are all examples of that. They are all Software-as-a-Service (SaaS) applications that depend on clouds for storing and processing data, and fit the NIST definition of SaaS:

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser, or a program interface. The consumer does not manage or control the underlying cloud infrastructure including

---

[26] https://www.nist.gov
[27] https://www.facebook.com
[28] https://twitter.com
[29] https://www.youtube.com
[30] https://www.linkedin.com
[31] https://www.flickr.com
[32] https://www.snapchat.com
[33] https://www.instagram.com

> *network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.* [19]

Instagram is a hugely successful social networking app made for sharing photos and videos from a smartphone. Instagram stored all photos in Amazon's cloud platform[34] (*AWS*) between 2010-2014. Facebook, which acquired Instagram in 2012, decided to move all data to their own datacenters. While 200 million people were using the app, they managed to move 20 billion digital photos to their new location at Facebook. [21]

Other social media companies are moving from one public cloud to another. The smartphone app Snapchat had 161 million daily users in the fourth quarter of 2016, and each user visited around 18 times a day [22]. Snapchat is heavily dependent on Google's cloud platform, but the company is now trying to become more independent by moving activities to Amazon's public cloud. In the process, they made a deal with Amazon to buy cloud services over a five-year period for $1 billion. For the same five-year timeframe, Snapchat already had a cloud services contract with Google. That contract was worth $2 billion. [23]

Another example is Uber, an online transportation network company with operations in 528 cities in 69 countries. It is also considering moving parts of its infrastructure to public clouds, most likely Amazon, Google, or Microsoft. [24]

2.3.2 The Mobile Cloud Era

In 2011, the keynote for ASPLOS XVI conference was titled "The Cloud Will Change Everything". The abstract started with the following words:

> *"Cloud computing" is fast on its way to becoming a meaningless, oversold marketing slogan. In the midst of this hype, it is easy to overlook the fundamental change that is occurring. Computation, which used to be confined to the machine beside your desk, is increasingly centralized in vast shared facilities and at the same time liberated by battery-powered, wireless devices.* [25]

It was already clear that battery-powered wireless devices, such as smartphones, would become an important driver for the paradigm shift towards cloud computing. As of today, it has become very common for smartphones apps to act as clients using a cloud service as its backend.

On the information page about mobile solutions for the Google Cloud Platform, it reads: "Mobile devices backed by scalable machine intelligence in the cloud is the defining computing paradigm of our time." [26] In the developer documentation for the same platform [27], it is explained that most mobile apps and games need a backend service for things that cannot be done solely on-device, such as sharing and processing data from multiple users, or storing large files. It is also explained that building a backend service for a mobile app is similar to building a web-based service, with some additional requirements.

---

[34] https://aws.amazon.com

One important aspect of using the cloud as a backend, is that user data from all users, and metadata about their activities, are collected and stored in a central backend. This opens for large scale analytics of user data and metadata, called big data analytics. Google Cloud Platform, and other cloud providers, offers analytics as an integral part of their services: “With your data on Cloud Platform, unlock insights with Google’s pioneering big data analytics products. Query petabytes of data with BigQuery, all without managing any infrastructure. Process both offline and real-time streaming data from your mobile apps using a unified programming model with Dataflow.” [26]

The collected data and analytics about users are what many app companies sell to data brokers to generate their revenues, and it is part of what has become known as the surveillance economy. [5, 16, 17]

This thesis explores an alternative approach to designing apps, without the need for a cloud or server as an app backend.

2.4 Middleware

The system in this thesis, Swirlwave, is referred to as a middleware. In this section middleware is described.

December 1998, the Internet Engineering Task Force (IETF) held an ad hoc workshop about middleware at the International Center for Advanced Internet Research. This was documented in RFC 2768 [28]. During the workshop, it became clear that the definition of middleware was dependent on the subjective perspective of those trying to define it. This resulted in the rough characterization of middleware as those services found above the transport (i.e., over TCP/IP) layer set of services but below the application environment (i.e., below application-level APIs).

Swirlwave is a middleware in that it is not itself a complete application. Rather, it is a software glue binding together applications on devices separated by a network. It resided above the transport layer and provides services to applications for transparently routing of traffic between peers and forming friend-to-friend overlay networks.

2.5 Peer-to-Peer

Peer-to-peer (P2P) is an architecture where the participants, called peers, can act both as clients and servers at the same time; they are said to be acting as *servents*. From a high-level perspective, the nodes in a peer-to-peer system are all equal. The peers connect to each other forming an overlay network. The topology of this network can be structured or unstructured. [29]

In a structured peer-to-peer architecture, the nodes are interconnected according to a deterministic algorithm. Examples of structured peer-to-peer architectures include distributed hash tables (DHT) and content addressable networks (CAN). [29]

Unstructured networks are formed in more of an ad-hoc manner, where new peers are connecting to arbitrary peers already in the network. For public unstructured peer-to-peer networks it is often a goal to construct an overlay that resembles a random graph, and algorithm exist to randomize which peers know who. [29] This will not be true in the same degree in architectures for private networks. [30]

Swirlwave as a middleware is based on an unstructured overlay, but an essential idea behind the system in this thesis was that both structured and unstructured architectures could be applied on top of it. Swirlwave combined with an application defined layer on top could form a two-layered architecture, where the lowest layer constituted an unstructured peer-to-peer system, and the higher layer could manage a structured topology.

Another concept in peer-to-peer networking, is the one of *superpeers* [29]. A superpeer is special role where the peer is given a responsibility on behalf of a group of peers, for instance it could be responsible for maintaining an index or directory used by several peers, or it could act as a common broker between a group of peers.

2.6 Friend-to-Friend

Rogers et al. [30] categorize peer-to-peer networks as public or private. Public peer-to-peer networks are open for anyone to join. Most peer-to-peer networks are public. A private peer-to-peer network, on the other hand, is defined as an Internet overlay in which the resources and infrastructure are provided by the users, and new users may only join the network by personal invitation. Private peer-to-peer networks are further divided in group-based and friend-to-friend. *Group-based* allows any pair of users to directly connect. *Friend-to-friend* only allows direct connections between users who know one another.

The term friend-to-friend networking was first coined by Dan Bricklin [31], in 2000. He described how explicit lists of network nodes that the user controlled in some way and trusted (and how much) could lead to useful systems without central control. He called this small list of cooperating computers friend-to-friend networking (F2F), and noted that F2F could lead to communications that did not depend on some large company's—or any other company's—servers (other than for plain old IP connectivity if the communication were at a distance).

Popescu et al. [32] described a friend-to-friend architecture for safe sharing of sensitive data, called *Turtle*. The system was inspired by the way people shared information in oppressive regimes, by only directly sharing with people they knew. The idea behind Turtle was to take this friend-to-friend exchange to the digital world, and come up with a peer-to-peer architecture allowing private and secure sharing of sensitive information between a large number of users, over an untrusted network, in the absence of a central trust infrastructure. In the paper, it is stated that existing peer-to-peer architectures with similar aims build trust relationships on top of a basic, trust-agnostic, peer-to-peer overlay. Turtle took an opposite approach, and built its overlay on top of pre-existent trust relationships among its users. They assumed that friendship relations were commutative. However, friendship was not transitive (the friend of a friend was not automatically a friend).

Swirlwave is based on a friend-to-friend architecture.

2.7 Distributed Systems

A distributed system is a collection of independent computers that appears to its users as a single coherent system. [29]

The system in this thesis can facilitate the creation of distributed systems that use smartphones as nodes, by letting smartphones form an overlay network of peers, and enabling these smartphones to act as servents, not just clients. The system in this thesis does this without the need for clouds or

application backend servers, and can be used for wide area connections, in contrast to most peer-to-peer solutions for smartphones that rely on LAN, Wi-Fi Direct, Bluetooth, or other narrow area solutions.

2.8 Internet of Things

The Internet of Things (IoT) is a field that has drawn significant research attention in recent years, but the exact definition is still forming. [33]

The basic notion is that things, devices that we usually not think of as computers, can be wirelessly interconnected to form a network and perform machine-to-machine communications. The things, or devices, can have sensor technologies. The network of devices can then be used for monitoring the environment, ambient intelligence, and autonomous control.

The system in this thesis does not focus on IoT, but its principles could be used. It is not unlikely for Internet connected IoT-devices to both lack publicly reachable IPs and often change locations, for instance if the devices are located outdoors and maybe even are in motion. A different scenario could be a smartphone acting as a superpeer for several devices with wireless capabilities. A smartphone usually has several short range, wireless channels that IoT-devices could connect to and it can have an Internet connection over the cellular network. In this way, an outdoor, moving IoT-network of devices could be tracked, monitored, accessed, and controlled via the Internet. Possibly, several networks of things could inter-communicate via their respective superpeers. These ideas are outside the scope of this thesis.

2.9 IPv4 and IPv6

When the term *IP* is used in this thesis, it refers to IPv4 [34], which is by far the most widespread IP version as of today. However, the most recent version is IPv6 [35]. Originally from 1998, it is intended to replace IPv4.

With IPv6, each device will be given an address that is public, and NAT will no longer be needed. An addition to this protocol, called mobile IPv6 is also promising [36]. It is designed to let devices keep their address even when changing networks. In this scenario, Swirlwave would not need to use Tor for connectivity because many of the connectivity problems would be solved.

However, the adoption of IPv6 is still low. Per April 2017, it is estimated to be about 21.7% in the U.S., 12.5% in the U.K., and 9.7% in Norway.[35]

2.10 Unreachable Addresses and Network Changes

It is important to be aware of the differences between designing a system for servers running in a local area network and a system for smartphones connected to the Internet.

Two distinct problems arise when working with mobile devices connected to the Internet. Firstly, the devices generally do not have IP addresses that are reachable from the Internet. Secondly, when devices frequently disconnect from the network and change addresses, how do we keep track of the

[35] https://www.akamai.com/uk/en/about/our-thinking/state-of-the-internet-report/state-of-the-internet-ipv6-adoption-visualization.jsp

devices' addresses. In Swirlwave, the goal is to do this in a peer-to-peer fashion. The use of separate servers for solving this task is avoided.

2.10.1 Unreachable Addresses

Usually, when someone uses a personal computer connected to the Internet, other computers cannot directly contact it. The personal computer can initiate contact with a server, but it cannot act as a server itself. The same is true for smartphones.

The reason is that the computers are not directly connected to the Internet, but are part of a local area network (LAN) that communicates with the outside world through a router connected to the Internet. The devices on the local area network are assigned IP addresses that are only valid inside the local area network. In the most common configuration, IPs are assigned by a DHCP-server. Devices will be given an address when they connect to the LAN, but this address can be different the next time the device connects to the same network.

When a computer connects to a server on the Internet, the server will see the IP of the router. The server sends its replies to the router. The router performs network address translation (NAT), and routes the traffic to the correct computer on the local area network. Only the address of the router is directly reachable from the Internet. The IPs of the computers inside the LAN are not reachable from the Internet. This is also the case for smartphones connected to the Internet via local Wi-Fi or cellular data such as 4G.

The router gets its IP from the Internet service provider (ISP). The ISPs usually do not reserve a specific address for a customer's router, which means that also the publicly visible address of the router can change. It is publicly reachable, but it is not static. Some ISPs offer static IPs to their customers for an extra fee. For technical reasons, it is not feasible for mobile operators to give out publicly reachable, static IPs for mobile phones using cellular data.

If one have a static IP to the router, one can make a computer on the network directly reachable from the Internet. One way of doing this is by assigning a static, local IP to the computer on the LAN. The computer will then always have the same IP on the LAN. It is still not reachable from the Internet. The next step is to configure port forwarding on the router, so that connections to certain ports on the routers IP is forwarded to the computer on the LAN. The computer can now be used as a permanent server available from the Internet.

2.10.2 Network Changes

The problem with smartphones, and other mobile devices, is that the users carry them around. As soon as the smartphone leave the local area network, and for instance changes to cellular data, it is on a completely different network with a different IP.

2.11 Network Address Translation

Network Address Translation (NAT) is a technique used by routers to let computers in a LAN connect to the Internet. Comer [37] summarizes NAT as technology that provides transparent IP-level access to the Internet for a host that has a private, nonroutable IP address.

The traffic between computers in the LAN and the Internet passes through the NAT, which modifies IP addresses. Before routing packages to the Internet, the local IP addresses in the packages are

changed to the routers public IP. When the server on the Internet sends data back to the router, the router's address in the packages are changed to the local IP of the computer that should receive the data. [37]

A problem that emerges with the use of NATs, is that hosts on the Internet cannot directly initiate a connection to hosts behind the NAT (except if manually configured, or in particular cases regarding handling of domain name lookups). Zhou [38] states that: "According to common firewall and NAT rule, hosts in a private address realm cannot be reached directly from public Internet." The computer behind NAT can connect to the Internet as a client, but communications cannot be initialized the other way.

Smartphones on cellular data are usually behind NATs beyond the control of the user. When two smartphones are behind separate NATs, they cannot directly connect.

Figure 1 shows that Device A is connected to the Internet. It is unable to initiate direct contact with Device B or Device C, because these are behind a NAT and thus only have a private IP address. The private IP address is only valid on the LAN. Note that the device B and C can contact each other. They can also contact Device A, if it has a public IP address.

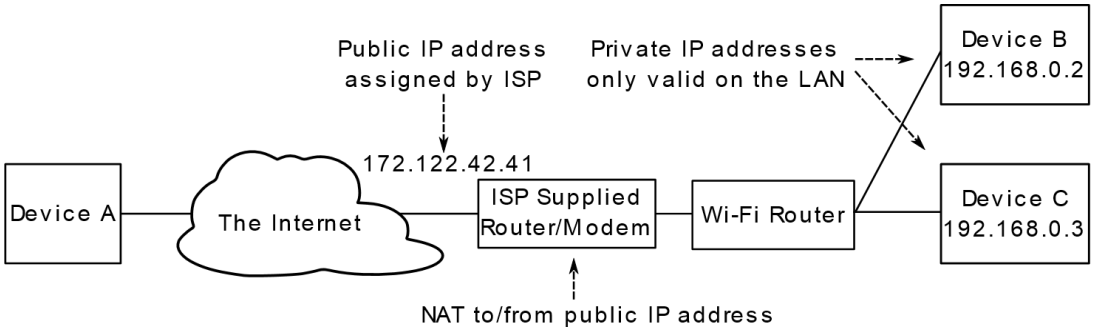


Figure 1 - Network Address Translation

2.12 NAT Traversal

To establish connections from the Internet to hosts behind NAT, a technique is needed for circumventing the problem associated with address translations and private IP addresses. This is called *NAT Traversal*. According to Zhou [38], NAT Traversal is to enable end-to-end protocol and application packets through NAT gateway directly or indirectly. It is crucial to Swirlwave because all peers must be able to act as servers. For a peer to be usable as a server in the Swirlwave system, it must be possible to connect to it from the Internet, even if it is behind NAT. The following subsections discuss some NAT traversal alternatives that were considered.

2.12.1 VPN

A *Virtual Private Network* (VPN)[37] lets client computers join a private network over the Internet, or other public network, and become included in the private network. This is done by setting up a VPN server allowing the client to establish a point-to-point connection, or tunnel, to transmit data over the public network to the private network, usually encrypted. There are several VPN-protocols, some which are NAT friendly (PPTP) and others that are less so (IPSec and L2TP).

VPN could thus be used to let several client computers, all behind NATs, connect to each other by connecting to the same private network.

Mobile VPN (mVPN) is suitable for devices that change networks. Conventional VPN tunnels will be disrupted when a device disconnects or changes to a new network. Mobile VPN makes it possible to keep the same application sessions open across network changes, so that the connected applications do not fail. It is otherwise like normal VPN.

There are several drawbacks when trying to use VPN for friend-to-friend based systems. Firstly, setting up VPN servers require publicly valid IPs. Secondly, virtual private networks require an amount of manual work for configuration and management. Thirdly, for client computers to act as servents, they either need reserved IP addresses, which means that they need to be known in advance, or alternatively some other mechanism for locating peers must be implemented. Fourthly, since F2F-networks are private P2P-networks, and since friendships are not transitive, complications arise when either trying to interconnect separate virtual private networks or letting every possible peer be part of the same VPN. The latter could also lead to scalability problems. These restrictions made VPN unsuitable as a basis for the system in this thesis.

2.12.2 UDP Hole Punching

UDP hole punching [38] is a technique to establish peer-to-peer communications even in the presence of NATs. The technique generally needs a server as a middlebox between the devices. For this a dedicated server with a publicly valid IP must be deployed. The technique is not applicable in all scenarios or with all types of NATs, as NAT operating characteristics are not standardized.

As the name reveals, the UDP hole punching is based on the User Datagram Protocol (UDP), which have no delivery guaranties, ordering of packets, or duplicate detection. For this reason, UDP was not considered an option for the system in this thesis, which is based on TCP-connections.

TCP hole punching [39] is possible given certain conditions but is less understood and needs to be supported by NAT servers, which makes it unsuitable for the system in this thesis.

2.12.3 SSH and Port Forwarding

Secure Shell (SSH)[36] is a network protocol operating in the upper layers of the OSI model. It allows users to securely connect to another computer over an unsecured connection, as the Internet. SSH can be used for a client to connect to a server, usually to login and execute commands in a shell, but also for tunneling, forwarding TCP ports, secure file transfers, and graphical user interfaces (X11).

SSH tunneling can be used to forward data to a port on a local computer to a port on a remote computer. The traffic will be sent between computers inside an encrypted tunnel. There are three types of port forwarding:

- Local
- Remote
- Dynamic

[36] https://www.ssh.com/ssh

Local port forwarding is the most straight forward example. It allows a local client to create a tunnel to a remote SSH server and then transparently forward traffic to the remote server, which then forwards to a specific port and address.

Remote port forwarding, or reverse port forwarding, is more interesting in the context of this thesis. Let us say that a server A is running behind a NAT, so that it cannot be reached from the Internet. With remote port forwarding, a tunnel can be initiated from computer A to a computer B. Computer B is reachable from the Internet. Traffic to a specific port at computer B, will the then be forwarded through the tunnel that computer A initiated earlier. In this way, a server behind a NAT can indirectly be made available from the Internet.

The two first alternatives can only forward to one specific host and port. Sometimes it is desirable to set up a tunnel that lets the client dynamically tell the server what host the incoming traffic should be forward to. This can be achieved by combining SSH with the *SOCKS* [40] protocol. On the SSH server, a SOCKS proxy will receive the traffic that comes through the tunnel from the client. The client must use the SOCKS protocol when communicating with the proxy at the server end of the tunnel. The protocol enables the client to dynamically specify addresses that the server side should proxy traffic to.

For all three alternatives, a server with a valid IP address or domain name must exist. SSH tunneling could thus not be used for the system in this thesis.

2.12.4 Tor Hidden Services

The purpose of the Tor [3] hidden service protocol is not NAT traversal, but nevertheless it is the solution for solving the problem of NAT and unreachable addresses for the system in this thesis.

The protocol's real purpose is to ensure responder anonymity, which means that for a server neither IP address or domain names are known to the client. Traffic is routed through the Tor onion routing network. A custom type of address is used to reach the server, called an onion address. As it turns out, this can be exploited to reach servers behind a NAT. Since the Tor routing network is open for all and available from the Internet, the wanted functionality was ready to use.

Tor hidden services are described in section 2.13.

2.13 Cloud Push Notifications

Because of the difficulties connecting to a smartphone behind a NAT, and because of the current cloud computing trend, the typical way of communicating with smartphone devices over the Internet is by using *cloud push notification/messaging* [41] technologies. Smartphone apps based on cloud push will open a TCP connection to a cloud and keep it open. The cloud can then send data—called notifications or messages—to the app through this long running connection.

Often the size and types of messages are restricted by the cloud notification service. Cloud push messaging is not fit for transferring files and larger data sizes from one smartphone to another. One example of this is *Firebase Cloud Messaging*[37] (FCM), which is Google's cloud push framework.

[37] https://firebase.google.com/docs/cloud-messaging

FCM allows notification messages with a maximum size of 2KB, and data messages with a maximum size of 4KB.

Push notification services are typically used by servers to reach mobile clients [41]. To allow mobile devices to communicate with each other, developers must usually manage the specifics themselves by implementing server software that communicate with the cloud's push notification service and orchestrate communications between the smartphones.

For smartphone A to send a message to smartphone B, a typical pattern of communications will be as in the following example:

1. A smartphone sends data to the cloud service
2. The cloud service sends data to the app server software
3. The app server software interprets the message and creates a message that will be sent to smartphone B.
4. The app server software sends the message to the cloud push service.
5. The cloud push service sends the message through the connection to smartphone B.

Microsoft Research Project Hawaii[38] developed a relay service that made it easier to implement this type of communication for *Windows Phone*[39] devices. The Project Hawaii program was discontinued on October 8, 2013.

2.14 Encryption

The term *encryption* belongs to the field of cryptography. It is a means to allow for authorized parties to establish confidential communication over an insecure channel that is subject to eavesdropping [42]. It can also be said to be the processing of a message by a sender in order to render it unintelligible to other than authorized recipients [43]. The unencrypted message/data is referred to as *plaintext*, and the encrypted version is referred to as *ciphertext*.

Encryption is used in network communications to preserve confidentiality, and to authenticate parties. *Confidentiality* is the avoidance of the unauthorized disclosure of information, and *authentication* is the act of determining the identity or role that someone has [42]. The system in this thesis uses encryption for confidentiality and authentication.

Many well documented encryption algorithms exist. The most common ones use secret keys when encrypting plaintext to ciphertext. Keys are also used to decrypt from ciphertext to plaintext. Only by having the correct key can encrypted data be correctly decrypted. Keeping keys secret is the crux of key-based encryption security.

A couple of algorithms have become de-facto standards in network communications. They fall into two main groups: Symmetric-key and public-key.

[38] https://www.microsoft.com/en-us/research/project/project-hawaii
[39] http://www.microsoft.com/windowsphone

2.14.1 Symmetric

In *symmetric cryptography*, the same key is used for encrypting and decrypting data. Two categories exist: Stream ciphers and block ciphers. The first encrypts one byte or digit at a time, the second encrypts larger blocks of data of a time. [42]

Symmetric-key algorithms can be as secure as the public-key algorithms described in the next subsection, and generally they are faster. The problem with symmetric-key algorithms is exchanging or agreeing on keys in such a way that an eavesdropping party does not get hold of the key. One popular protocol for securely agreeing on keys is the *Diffie–Hellman key agreement* algorithm. It lets parties generate the same keys by exchanging some information publicly, and combining it with information that is kept private at each end. [42]

One example of a symmetric encryption algorithm is *Rijndael*, which is the basis for NIST's *Advanced Encryption Standard* (AES) [44]. AES is secure and fast, and is enhanced further when accelerated by hardware.

2.14.2 Public-key

Public-key cryptography, also called asymmetric cryptography, operates with two different types of keys; one key is used for encrypting data, another key is used to decrypt the data. Together the two keys form a key pair. *RSA* is a widely used public-key algorithm. [42]

By having two keys, one can be kept private and one can be given to other parties. The first is then called the private key, the latter is called the public key. The advantage of private and public keys is that the private key is not shared with other parties, thus the risk of unauthorized third parties (*adversaries*) stealing the key during network communications is removed. The public key can be shared without any restrictions, given the important assumption that it is infeasible to derive the private key from the public key. [42]

When a communication party encrypts data with a private key, everyone with a public key can decrypt the data. Consequently, they know that only the communication party owning the private key could have encrypted the data. It is then taken as proof that the data originated from this party. Vice versa, when a communication party encrypts data with a public key and sends it to the party owning the private key, then it will know that only the owner of the private key will be able to decrypt the data. The combination of the two cases forms the basis of widely accepted and used network communications schemes for ensuring integrity, authentication, and non-repudiation of data.

2.14.3 Public-Key Encryption Use Cases

Public-key encryption is slow compared to symmetric, but it is difficult to safely transfer a symmetric key from a sender to a receiver. If anyone gets hold of the symmetric key, then they can decrypt all exchanged data, and even forge data by encrypting with the stolen key. A solution is to use public-key encryption for securely exchanging symmetric keys. When a symmetric key has been securely transferred to a recipient, the rest of the data can be encrypted using this key for the rest of the session.

Another use of public-key encryption is *signing* [42] a hashed value calculated from a document or block of data for non-repudiation purposes. A *hashing algorithm* [42] is a one-way function that calculates a number, called a hash value, from a block of bytes, for instance a document. This function is usually so that even a small change in the original data will produce a very different hash value, and

so that it is very unlikely for different data to produce the same hash value. Hashing is not encryption, because the hash value cannot be decrypted into its original content. If a sender encrypts the hash value with its private key, and sends it together with the data, then anyone having the public key can decrypt the hash value and know that it was encrypted by the originator. They can then themselves calculate the hash value of the data and compare it to the decrypted hash value. If the values are not equal, then the data has been tampered with.

Public-key encryption is used for authentication as well. The specifics vary from protocol to protocol, but basically it involves the sender signing authentication data or hash values with its private key and including it in a message that is encrypted with the receiver's public key.

2.14.4 Certificates

Certificates are part of a *public-key infrastructure* (PKI), and contain sets of keys, typically one for encryption and one for signing. The certificates are created by a trusted third party called a *certificate authority* (CA). The certificate come in to versions, one that includes private keys, and one with only public keys. The public certificate can be distributed to anyone. The receiver of a certificate can validate the authenticity of the certificate by contacting the CA. [42]

The use of certificate authorities is considered standard, but alternatives exist: OpenPGP-compatible systems, for instance, rely on a decentralized trust model called a *web of trust*[40].

2.14.5 Man-in-the-Middle

As seen in this section, one purpose of encryption is to prevent adversary parties from eavesdropping. It is also crucial to preventing adversaries from modifying, substituting, or replaying old data sent between communication parties. A man-in-the-middle is an adversary that can be expected not only to eavesdrop, but to actively carry out that type of actions. When constructing a system for confidential communications between parties, the question of how to prevent eavesdropping and man-in-the-middle-attacks must always be taken into consideration.

2.15 Tor

Swirlwave uses *Tor*[41] as a means to reach devices that are lacking publicly visible IP addresses. More specifically, it uses the *Tor hidden services protocol*[42]. The Tor hidden services protocol was found suitable for use in Swirlwave after considering possible alternatives, which was described in section 2.12.

However, the real objective of Tor is not connectivity. Tor is a network built for anonymity. It is designed to conceal online traffic from surveillance and monitoring by relaying through several nested proxies, compared to the layers of an onion. All traffic between relays are encrypted. Per February 2017 the network consisted of over 7000 volunteer relays[43]. The network is free and available for all to use.

---

[40] https://gnutls.org/openpgp.html
[41] https://www.torproject.org
[42] https://www.torproject.org/docs/hidden-services.html.en
[43] https://metrics.torproject.org

The name Tor was originally an acronym for The Onion Router after the original project that developed the protocol. Its core principles were developed in the mid-1990s by the United States Naval Research Laboratory, by Paul Syverson, Michael G. Reed, and David Goldschlag, with the purpose of protecting U. S. intelligence communications. The Defense Advanced Research Project Agency (DARPA) developed onion routing further in 1997. Syverson went on to implement the first version of The Onion Routing Project in 2002, together with Roger Dingledine, and Nick Mathewson. The paper "Tor: The Second-Generation Onion Router" was released in 2004 at the 13th USENIX Security Symposium. [3]

Tor is a public overlay network where traffic is routed through *onion router*s (OR). Traffic is encrypted and relayed via at least three onion routers before reaching its destination. On each end of the network there are *onion proxies* (OP). The onion proxy on the client side runs as a process on the client computer. It has access to a directory of onion routers. The client side onion proxy chooses the onion routers it wants to use, and then a *circuit* is built. The circuit defines how the routers are connected to each other. Each onion router in a circuit knows only its predecessor and successor, but no other nodes in the circuit. When an onion router relays traffic to another router in the circuit, then they will use encryption keys that are only valid between the two. The keys are deleted when no longer used. At the destination end of the circuit, an onion proxy receives the traffic and sends it to the destination server. The server that receives the traffic, does not know where the traffic came from. It only knows about the onion proxy where the traffic exited the Tor network.

Note that the destination server is just an ordinary server reachable from the Internet. It is not aware of Tor. The Tor network protects the privacy of the client. Tor hides what the client connects to, it encrypts traffic, and it hides who the client is from the server. It does *not* hide the server's location from the client or anyone else. It is just a server like any other, unaware of Tor.

2.15.1 Tor Hidden Services

The *Tor hidden services protocol* [3] is designed for responder anonymity. *Responder anonymity* means that the server's location will be hidden from clients.

As described earlier, a client connecting through the Tor network can contact any server that is reachable from the Internet. For a server to be reachable from the Internet, it must at least have a publicly valid IP-address, which means that its network location is publicly available. Thus, its location is not hidden or anonymous.

If a server wants to hide its location, the Tor hidden services protocol can be used. In this protocol, an onion proxy on the server side will register a hidden service in the Tor network. It will then get a special type of address, called an onion address, which is valid inside the Tor network. Clients can then reach the onion proxy via Tor by using the onion address. When traffic reaches the onion proxy on the server side, it will route the traffic to the server. The protocol makes it possible for traffic from a client to be routed through the Tor network to the server, without letting the client know the server's real location.

As a side effect of hiding the server's location, the server becomes available without a public IP-address. Also, it is important to note that the onion proxy that registers the hidden service initiates a connection from the server side to the Tor overlay network. This means that the server side can be behind a NAT, because NAT only prevents connections from the Internet to the server, not in the

opposite direction. An unintended result of the protocol is therefore that hidden services can be used to reach servers behind NAT. This is the reason why Swirlwave uses it, not because of responder anonymity.

2.16 Other Smartphone Channels

Smartphones usually have several channels of communications. In this section, the most important ones are briefly described.

2.16.1 The Internet

Smartphones in general can connect to the Internet, and the Internet is the primary communication channel used by Swirlwave. In most cases a smartphone will connect to the Internet wirelessly through Wi-Fi or a cellular data channel.

2.16.1.1 Wi-Fi

Wi-Fi[44] (IEEE 802.11)[45] is used to connect wirelessly to an *Ethernet* (IEEE 802.3)[46] local area network at home, at work, at campus, etc. When the smartphone is connected to a local area network with Internet access, this is usually a high bandwidth connection to the Internet. The drawback is that the range of the Wi-Fi is limited, usually inside a building, and the connection is broken as soon as the user leaves this area.

2.16.1.2 Wi-Fi Direct

Wi-Fi Direct[47] is a standard that lets devices connect peer-to-peer without the need for a wireless access point. This type of connection is not for accessing the Internet, but for direct connections between devices.

Wi-Fi Direct supports typical Wi-Fi speeds, which can be as high as 250 Mbps. Even at lower speeds, Wi-Fi provides plenty of throughput for transferring multimedia content with ease.

2.16.1.3 Cellular Data Channel

Besides Wi-Fi, smartphones can connect to the Internet by using the cellular data channel. This is an Internet connection provided by the mobile carrier network. The bandwidths and underlying technologies can vary. Mobile coverage can be a factor. As of today, 4G *LTE*[48] is considered standard.

The advantage of cellular data compared to Wi-Fi is that cellular data can be used when there are no available Wi-Fi networks, for instance outside, or a place where a there is a Wi-Fi network in range, but the user do not have the credentials to access it. Cellular data can be a more secure choice than Wi-Fi in cases when the Wi-Fi network is public and should not be trusted.

2.16.2 Short Message Service

Nearly all smartphones have SMS capabilities. When we think of SMS, we mostly thing about texting, but in smartphones the SMS messages can be used for sending data that apps can processes. A data

---

[44] http://www.wi-fi.org
[45] http://www.ieee802.org/11
[46] http://www.ieee802.org/3
[47] http://www.wi-fi.org/discover-wi-fi/wi-fi-direct
[48] http://www.3gpp.org/technologies/keywords-acronyms/98-lte

SMS will be sent over the cellular data channel or over the same channel as text SMS messages (GSM). The drawback is that it can take a while from a message is sent until it is received, and the maximum length of a single data message is short; 140 bytes including the User Data Header (UDH). It is possible to send concatenated SMS messages, which consist of several single messages. A theoretical maximum is 255 SMS messages concatenated to one, but this varies from operator to operator, and 3 is considered the reliable maximum. If a data SMS is sent over the GSM network, the cost can be different from sending data over the cellular data channel. It can be higher or lower, depending on the user's subscription.

An important aspect of SMS messages for the system under discussion in this thesis, is that SMS messages are sent to a phone number. The phone number can be seen as a static, publicly reachable address to the device. The system in this thesis uses SMS as a last resort of communications, if a peer cannot otherwise be reached.

2.16.3 NFC

Near Field Communication (NFC)[49] is a technology that can be used for transferring data within a range of approximately 4cm. Data can be stored on a tag, often in the form of a sticker, and be transferred to an NFC-capable device when it is close enough. It is also possible to transfer small amounts of data from one NFC-enabled device to another. Many smartphones are equipped with NFC. This thesis uses *Android Beam*[50], an NFC protocol, to transfer contact information between smartphones.

2.16.4 Bluetooth

Bluetooth[51] is a wireless communications technology found in a wide range of products, including PCs and smartphones. It uses short-wavelength radio signals (UHF). There are many versions and classes of Bluetooth. Bandwidths is 25 MBits/s for versions up to and including version 4.0, and 50 MBits/s for Bluetooth 5. Class 3 has the shortest distance with less than 1m. Class 1 is mostly used for industrial use cases, and can have a range up to 100m. Most smartphones have Bluetooth class 2 with a range of about 5-10m.

Bluetooth 5 is the newest standard, and can have ranges up to 240m. It is designed with peer-to-peer and IoT in mind.

Bluetooth low energy (BLE, Bluetooth LE, Bluetooth Smart) is another type of Bluetooth, designed for low power consumption and personal area networks.

2.16.5 Radio

Products for smartphones exist that use radio signals with a longer range than Bluetooth, for instance *goTenna*[52]. GoTenna is made for off-grid communications when there is no cellular network signal or wi-fi available. It uses very high frequency (VHF) radio signals. Its range depends on the environment. According to the company's homepage, the range in a congested environment can be up

[49] http://nfc-forum.org/what-is-nfc/
[50] https://support.google.com/nexus/answer/2781895
[51] https://www.bluetooth.com/what-is-bluetooth-technology
[52] https://www.gotenna.com

to 1 miles (1.6 kilometers), and in an open environment it can be up to 5 miles (8 kilometers). With goTenna Mesh, users can extend the range by relaying messages from one device to another.

2.16.6 Mobile-Edge Computing

Mobile-edge Computing (MEC) [45] is a term often mentioned in conjunction with the Internet of things (IoT). Technical standards for this technology is being developed by the *European Telecommunications Standard Institute* (ETSI)[53].

MEC is not a wide area solution, but can nevertheless become important for peer-to-peer ad-hoc networking in the future. It is implemented in the cellular base stations, owned by telecommunications companies, and makes it possible to offer services for mobile devices connected to the base station. This can be used for creating peer-to-peer communications between devices in range of the same base station, and is the basis for services as mobile advertising, 'Smart City', footfall analysis, campus management, augmented reality, and video analytics.

2.17 Android Operating System

Android[54] is an operating system developed by Google based on the *Linux*[55] kernel. It was made primarily for smartphones, but it is now used for a range of products, for instance tablets, smartwatches, and smart TVs. It is estimated that Android's share of the global mobile operating system market was about 87% the last quarter of 2016 [46]. In September 2015 Google reported that there were 1.4 billion active Android devices in the world [47].

2.17.1 Android Application Development

Android applications, or apps, follow a different structure than traditional desktop applications. Most notably, the ability to share graphical user interface (GUI) components between apps is a central idea in the development framework.

Official documentation for developing for Android is available online[56]. There is also a vast array of books on the topic, for instance *Professional Android 4 Development* [48] and other books that were used as material for this thesis.

Another difference is that the operating system takes greater liberties when it comes to automatically shutting down applications to reduce the usage of resources as battery power and network traffic. Android has a comprehensive regime with respect to resource management. For example, Android can restrict network access to periodic windows of time, when the apps can sync with online servers. This poses extra challenges when designing a peer-to-peer system, such as Swirlwave, that should be available as much as possible.

2.17.1.1 Activities and Services

A crucial building block for creating Android apps are *activities*. An activity generally corresponds to one, single screen in an application. An app can be composed of many activities. A special feature in

[53] http://www.etsi.org
[54] https://www.android.com
[55] https://www.linux.com/what-is-linux
[56] https://developer.android.com

Android is that activities in one app can be started by another app. This enables user interfaces and functionalities that are useful across apps to be shared.

In addition to visual components, apps can have *background services*. A background service is part of an app, but can be configured to run even if no graphical user interface is shown. The Swirlwave app uses a background service to do the bulk of its work.

2.17.1.2 Intents

Another important building block in Android's app framework, is its message passing infrastructure that is used inside apps, between apps, and between operating system and apps. The messages are called *intents*. An intent can be used to initiate an activity that reside in another app. Intents can also be used to share data and communicate between apps. Intents are also used as part of broadcasting that an event has occurred.

Swirlwave uses intents different purposes. For instance, it is used to open the main screen (an activity) when the user taps on a Swirlwave notification in the notification bar. Also, on NFC activation, Swirlwave receives an intent that exchange contact information between peers.

2.17.1.3 Broadcast Receivers and Intent Filters

An application can implement *broadcast receivers*. A broadcast receiver handles events that have been broadcasted as intents. To specify which types of events that a specific broadcast receiver should handle, *intent filters* are used.

The Android OS will broadcast certain events, such as when an SMS is received, or the network connection changes. Swirlwave implements broadcast receivers for receiving data SMS, and for monitoring network changes.

2.17.1.4 Android Manifest

All Android apps have a manifest file, *AndroidManifest.xml*, which defines the structure and metadata of the application, its components, and its requirements. It contains activities, services, content providers, broadcast receivers, intents, permissions, icon, app version number, etc.

2.17.1.5 Permissions

Diverse types of functionality provided by Android are restricted. An app must be given explicit permissions to use this functionality by the user. The permissions that an app will request is found in the Android manifest file.

The permissions needed by Swirlwave are for using Internet, NFC, and SMS. The user will be asked to grant these permissions when installing the app.

2.17.1.6 Distributing Apps

The usual way to install Android apps is via *Google Play*[57], an online service integrated in the operating system, where developers can publish their own apps.

The app goes through a validation processes, in which Google can permit or deny publishing it to their directory. Apps can be payed or free. The system includes an infrastructure for downloading, installing

[57] https://play.google.com/store

and upgrading apps. As part of the validation process, Google has built in automatic controls of which types of functionalities exist in an app. If an app does not follow the guidelines, which can be changed from time to time, the app can be removed from Google Play.

This poses a problem for apps running background processes that must run even when the screen is off or locked, which is the case for the Swirlwave app. In newer versions of Android, the app must ask the user for permissions to prevent a background process from being automatically shut down by Android as part of power management. A problem with this, is that apps that ask for this type of permission, have been reported by developers to have been removed by Google Play after a change in the guidelines, even if the app had been cleared for having valid reasons for running background processes this way. Another problem is that Android 6.0, Marshmallow, has a bug. Even if a user has answered yes to allowing the background service to run even when the screen is off or locked, this permission is not correctly set. The solution is to manually open the Android configuration settings and give the Swirlwave app permissions to run when the screen is locked or off.

3 Related Work

3.1 Turtle

Popescu et al. [32] describes a friend-to-friend (F2F) architecture for safe sharing of sensitive data, which is called *Turtle*. As Swirlwave, it builds the overlay network from pre-existing trust relationships:

> *What makes Turtle unique is the bottom-up way it builds its overlay starting from pre-existing trust relationships among users. To the best of our knowledge, there are no other peer-to-peer systems that employ this technique.* [32]

Turtle differs from Swirlwave by being a theoretical description of a file sharing network architecture where search queries were flooded through the network. Swirlwave is a middleware enabling friend-to-friend networking without being tied to specific applications.

Turtle relies on cryptographically secure connections, but the problem of encryption key agreement in the absence of a central trust infrastructure is solved differently from Swirlwave. Turtle uses questions and answers that are assumed to be known by both users, because they are friends in real life. In the version of Swirlwave implemented for this thesis, public keys are exchanged by initially meeting in person and transferring contact information via NFC. These keys are not needed for creating cryptographically secure connections, Tor is used for this, but they are primarily used for authentication and additionally for communications outside of Tor, for example when sending SMS.

Turtle is designed for small data items and described as a packet routing overlay, while Swirlwave is designed for larger data transfers. However, it is mentioned in the paper that it should be straightforward to re-design Turtle as a circuit switching network, so it could accommodate large data transfers.

This means that Swirlwave possibly can be used as a middleware for implementing a version of Turtle that runs on smartphones and support large file transfers.

3.2 Orbot

Orbot[58] is the official version of the Tor onion routing service on Android.

It can transparently proxy all Internet traffic through Tor, or let users configure which apps that should be proxied. Unfortunately, this requires some advanced modifications to the smartphone's operating system installation. The user must root the device and install a custom iptables-capable ROM, such as the discontinued *CyanogenMod*[59]. Without this modified version of the Android operating system, apps must implement support for connecting to the local Tor proxy. One of the difficulties with this, is that Tor requires the client to use the *SOCKS4a* [49] protocol, which is not supported out of the box in

[58] https://guardianproject.info/apps/orbot

[59] http://web.archive.org/web/20161224194030/https://www.cyanogenmod.org

the *Java*[60] implementation of sockets on Android. This is because the socket implementation in Android is based on *Apache Harmony*[61] instead of the stock Java-libraries.

Orbot is mostly used in conjunction with a couple of Tor-capable client applications, for example the *Firefox*[62]-based browser *Orfox*[63] and the chat app *ChatSecure*[64].

Orbot can be used to publish hidden services, but the app is not aware of changes in network locations. Clients will not be able to connect to the service when the network location changes, until the client shuts down and starts their Tor connection again. For someone implementing a peer-to-peer system, the consequence is that all peers would have to restart their Tor proxies every time one of them changed its location. This would break ongoing connection to other peers. Swirlwave instead monitors network connections on the devices, and assigns a new hidden service address when the device connects to a new network. This address is then announced to the peers in the contact list. Orbot is purely client-server based and does not have a notion of peers.

Orbot cannot be used directly for Swirlwave, because programmatic control over the Tor proxy and hidden services is needed. Still, the same official libraries are used through a library from the Thali Project.

3.3 Thali Project

Thali[65] is an experimental platform for building peer web sponsored by Microsoft. It is described as an open-source software platform for creating apps that exploit the power of personal devices and put people in control of their data.

Swirlwave uses a fork of a library from this project for controlling Tor and hidden services. Thali did plan to use hidden services as a means of communication between smartphones, but they seem to have abandoned the idea, and their library had not been worked on for several years when this thesis began using it.

The reason for abandoning the idea of using hidden services, is the problem with devices changing network locations:

> *Right now using Tor Hidden services is hard for us because HSs are clearly designed for stationary services not mobile ones.*[66]

At the time being the project focus on BLE, Bluetooth, and Wi-Fi Direct. These technologies are not wide area technologies.

---

[60] https://java.com
[61] https://harmony.apache.org
[62] https://www.mozilla.org/en-US/firefox/new
[63] https://guardianproject.info/apps/orfox
[64] https://chatsecure.org
[65] http://thaliproject.org
[66] http://thaliproject.org/ThaliAndTorHiddenServices

4 Architectural Design

4.1 Introduction

This chapter presents the *system design* and *architecture* of Swirlwave. Architecture in this context means software architecture or systems architecture. Before proceeding, the meanings of the terms are explained.

Architecture is described in *The IEEE Standard Glossary of Software Engineering Terminology* simply as: "The organizational structure of a system or component." [50] The term usually refers to a high-level view of a system's organization, its most important structures, and how these structures interrelate with each other and the environment. The purely internal workings of a structure, which consist of the parts and properties that are not important outside of it, are not considered architectural.

Bass *et al.* defines software architecture as:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural. [51]*

In computer science, the term systems architecture is often seen as broader than software architecture, and can include both software and hardware. For instance, for distributed systems it will often include physical nodes and networks in its descriptions.

Creating an architecture is part of the process called system design. In *A Dictionary of Computer Science, 7th edition,* system design is defined as follows:

> *The activity of proceeding from an identified set of requirements for a system to a design that meets those requirements. A distinction is sometimes drawn between high-level or architectural design, which is concerned with the main components of the system and their roles and interrelationships, and detailed design, which is concerned with the internal structure and operation of individual components. The term system design is sometimes used to cover just the high-level design activity. [43]*

The term *architectural design* can thus be used to describe just the high-level design of the system. IEEE also has a definition of architectural design: "The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." [50]

In the design process, the goals and requirements of the system is used as a basis for creating the architecture. In this thesis, the goals are defined in the first chapter; these goals are derived from the problem statement, which again are connected to the motivation. The system requirements also stem from the background chapter, where for instance technical challenges are uncovered and discussed.

The rest of chapter four is organized as this: First an overview of the architecture is presented. Then architectural choices are explained in light of requirements and goals. Next, parts of the architecture are explained in more detail in separate sections.

4.2 Overview

This section gives an overview of Swirlwave's architecture.

A system based on Swirlwave has a shared nothing architecture that consists of independent and self-sufficient nodes that do not share memory or disk storage. The nodes will generally run on separate physical devices, most notably smartphones connected to the Internet through Wi-Fi or cellular data. It is a peer-to-peer system that does not rely on cloud computing or application server middleboxes. This contrasts with the most common architectures used for communicating between smartphones, which generally rely on some form of *cloud push* technology.

4.2.1 Friend-to-friend

Swirlwave is based on friend-to-friend networking between devices. Friend-to-friend networking is a category of unstructured, private peer-to-peer networking where peers only connect directly to already known peers (friends). This is illustrated in Figure 2. Friendships are commutative, which means that friendships are mutual. This is illustrated as lines between the peers with arrows pointing in both directions. Several peers can share the same friends—Peer A and Peer B both have Peer D as a friend—but friendships are not automatically transitive; a friend of a friend is not necessary a common friend. Peer C is a friend of Peer B, but neither Peer A, nor Peer D, know Peer C. Peer A cannot contact Peer C directly, because they do not know about each other. If this seems as too limiting, it is important to appreciate that this structure still makes it possible to reach unknown, faraway peers indirectly: Let us say that Peer A wants to make a far-reaching query, which is not limited just to its friends. Peer A could then ask Peer B to relay a query to its friends. Peer B could then relay the query to Peer C, which again could relay the query to its friends, and so on.

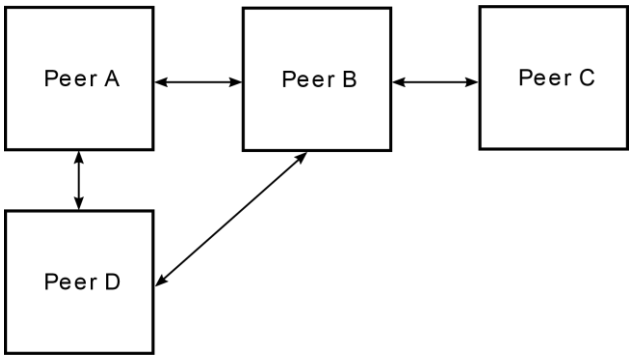


Figure 2 - Friend-to-friend (F2F): Peers with commutative, but not transitive, friendships

4.2.2 Unstructured and Structured

Tanenbaum (p. 49) [29] explains that although it would seem that structured and unstructured peer-to-peer systems form strict independent classes, this need actually not be the case. A two-layered approach can be used to create and manage a specific, structured topology on top of an unstructured overlay. Figure 3 is a modified version of Figure 2-10 from Tanenbaum [29]. It shows Swirlwave as a

lower layer that provides the unstructured overlay, with an application on top that creates and maintains a specific topology.

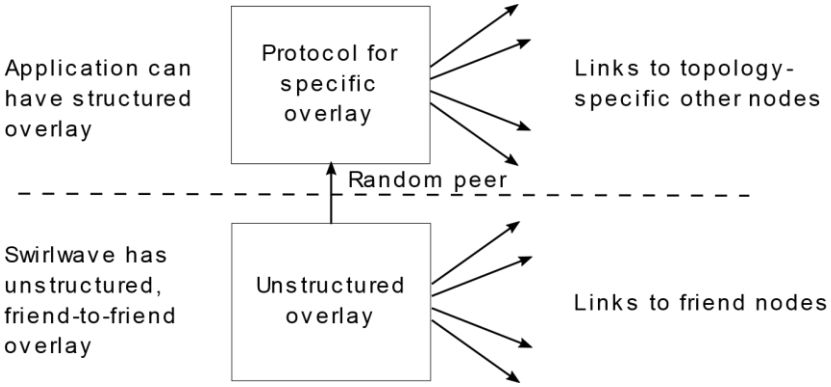


Figure 3 - A two-layered approach for constructing and maintaining specific overlay topologies on top of unstructured friend-to-friend overlays

4.2.3 Middleware

Swirlwave is not a complete application on its own, rather it is a middleware that application developers can use to overcome the difficulties met when communicating between devices (e.g. smartphones) that lack public IPs, and that frequently change locations. Figure 4 shows two devices illustrated as boxes. Swirlwave is shown as a layer that fits between the application and the operating system (including transport layer services, such as TCP/IP). The devices are connected to the Internet, and traffic is routed through the Tor overlay network.

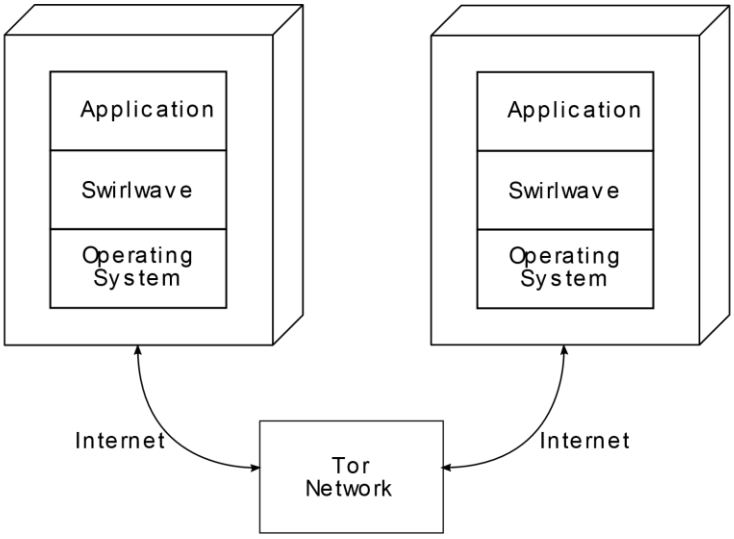


Figure 4 - Swirlwave is a middleware

4.2.4 Client and Server Proxies

Each peer is a *servent*, which can act both as server and client. Applications can be seen as having two parts: A client part, and a server part. A variety of different applications can run on top of the same Swirlwave installation.

In Figure 5, more details have been added. The illustration shows a peer acting as a client to a peer acting as a server. On the left side, the client part of an application contacts the server part of an application on another peer. To do this, the client does not need to know the exact location or address of the server peer. Neither does it need to know how to connect to Tor. It simply connects to a local proxying service provided by the Swirlwave installation. The Swirlwave client proxy listens to different ports and accepts ordinary TCP-connections from the client. Each port will be mapped to a specific friend peer. The client proxy, which knows the current onion address of the other peer, then connects to the locally running onion proxy. The onion proxy sends the traffic through the Tor network to the onion proxy running on the server side peer. A Swirlwave server side proxy receives the traffic from the onion proxy. Lastly, the Swirlwave server proxy sends the traffic via ordinary TCP to the locally running server part of the application.

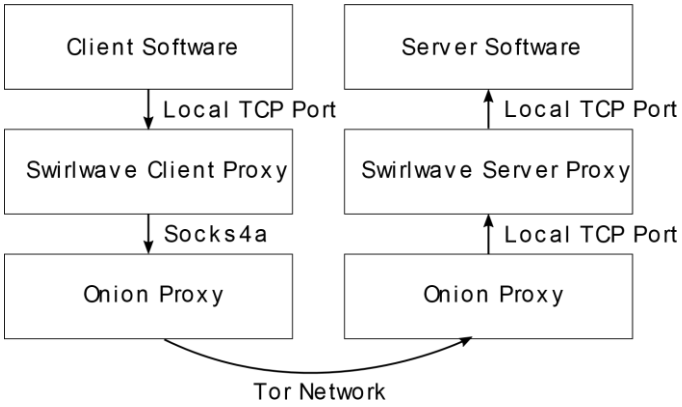


Figure 5 - Proxying from client to server

In addition to routing data traffic on behalf of the application layer, Swirlwave has a set of system messages that are sent between peers. One example is a message informing that a peer has changed its address. The Swirlwave proxies will differentiate between system messages and application traffic.

4.2.5 Contact List

A vital component of Swirlwave is the contact list. Each peer in Swirlwave keeps a contact list of its friends. An entry in the list contains data that is needed to communicate with the other peer and is used by the Swirlwave proxies. The contact list entries are stored in a local database on the device.

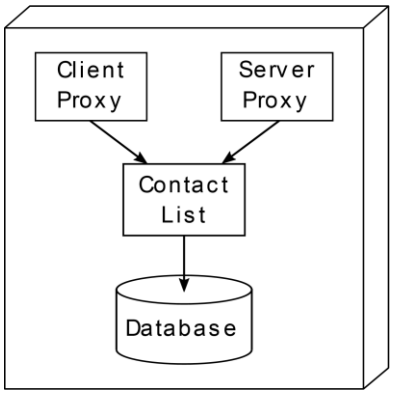


Figure 6 – The contact list is used by the client and server proxy, among others.

4.2.6 Locating Peers

Keeping track of peer locations is a central feature of Swirlwave. This is achieved without external directory services or single points of failure. Swirlwave provides a layer of location transparency to applications, and friends can be contacted even if they change networks.

Peers keep the addresses of their friends in their contact lists. When a friend changes its location, its address will change. The peers must thus obtain the friend's new address and update their contact lists. If not, the friend can no longer be contacted.

In Swirlwave, it is the responsibility of the peer that has changed its address to send a system message to all its contacts informing them about the change. If a contact cannot be reached, it will be the unreachable contact's own responsibility to obtain the new address in other ways at a later time.

Figure 7 shows the same peers as in Figure 2. This time Peer B has changed its address. It sends a system message to its three friends. This is illustrated as arrows marked with the number 1. Peer A successfully receives the address and updates the contact list entry for Peer B. The two other peers cannot be reached. When Peer C and Peer D later tries to contact Peer B, they discover that it cannot be reached. This is because they do not have the correct address. Note that Peer A is a common friend of Peer D and Peer B. Peer D therefore asks Peer A for the address of Peer B. This is illustrated with an arrow marked 2a. Peer C, on the other hand, does not have any other friends to ask. Instead, it uses an alternative channel to ask Peer B directly for its address. This alternative channel is typically slow and only supports lesser amounts of data, which renders it unsuitable as a primary communications channel, but its address is stable. Because the address is stable, it can be used to contact Peer B directly. Peer C is not dependent on having any other friends in its network. This is illustrated with an arrow marked 2b. For smartphones, SMS fits the requirements for being an alternative channel. SMS is supported by nearly all smartphones, and the phone number can be viewed as a stable address. The friends' phone numbers are kept in the contact list.

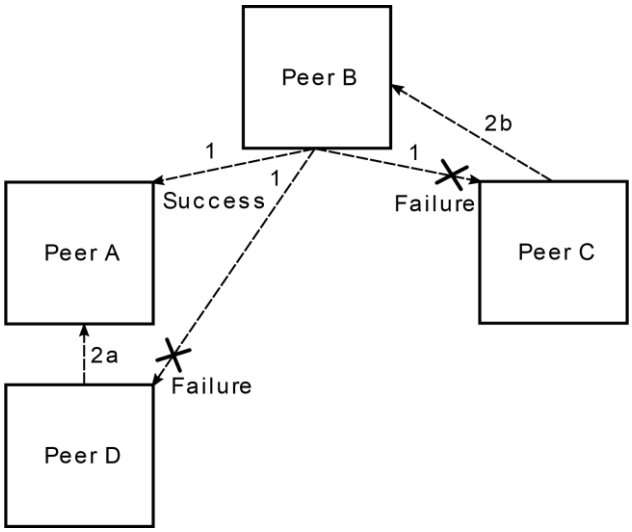


Figure 7 - Peer B changes address and tries to inform its friends, but two friends cannot be reached at the moment.

4.2.7 Authentication and Confidentiality

Swirlwave uses Tor hidden services. Hidden services are end-to-end encrypted between onion proxies. Figure 5 shows two onion proxies. One is running on the client device, and one is running on the server device. All data transmitted between the two devices are encrypted. Swirlwave does not encrypt the traffic that is only transmitted internally on the device. The client and server software, depicted as the highest layers in the illustration, can off course implement custom security mechanisms that involves encryption, if an extra layer of confidentiality is needed.

The Tor protocol is designed with anonymity in mind. Consequently, the onion proxy on the server side will not know the origin of incoming connections. In Figure 5Figure 2, the onion proxy on the right-hand side will not by itself be able to identify who is on the left-hand side. It could be anyone.

Swirlwave solves this by providing a built-in authentication mechanism for validating the identity of incoming connections from friends. This functionality is part of the Swirlwave client and server proxies, and it is based on public-key cryptography. Every peer in Swirlwave has a set of asymmetric encryption keys. The private keys are never shared with anyone, while the public keys are shared freely with friends. The public keys are exchanged out-of-band (e.g. via NFC) and stored in the contact list. When a client side proxy establishes a new connection to a server side proxy, a system message is first sent. The system message contains information that is encrypted with the client's private key. The server can then use the client's public key to decrypt the information as part of validating the identity. The connection attempt will be refused if the claimed identity of the client cannot be authenticated.

Public-key cryptography is also used for both authentication and confidentiality when communicating with friends via alternative channels, such as SMS, because these channels usually are unencrypted.

4.2.8 Extensibility

Swirlwave has a plugin architecture to allow for extensibility. Applications can register as plugins with Swirlwave to become integrated in the system. Different applications can register independently, and one installation of Swirlwave need not have the same set of applications as another installation.

The set of applications registered with an Swirlwave-installation is exposed to friends as a list of capabilities. Friends can look up which applications are registered, together with properties of the registered applications. An example of a property is whether client functionality, server functionality, or both are available. Application specific information can also be published as a property.

The contact list contains information about the capabilities of the different friends. Peers will inform their friends by sending a system message whenever their capabilities change. The friends will then update their contact lists accordingly.

As implied earlier, applications can include a client and a server part, but not necessarily both. It is possible to install a server-only application at one peer, and on other peers install different client-only applications capable of communicating with this server-only application. This is made possible by the architecture because each type of service has a unique identifier. A client must define which service types it supports by referring to these service type identifiers. Clients and servers can then be coupled by matching the service types they support.

4.3 Design Decisions from Goals and Requirements

In this section, the architecture is discussed in light of system goals and requirements.

4.3.1 Why Peer-to-Peer?

The main goal of the thesis is to suggest an alternative to the dominating, cloud centric approach to smartphone app development. The two main reasons for wanting an alternative can be summed up as:

1. Privacy and clearer ownership of data through the avoidance of cloud computing and backend servers for storage, processing, and coordination of communication between devices
2. Better utilization of processing power, storage resources, and networking capabilities of smartphones by enabling them to be used as nodes in distributed systems

The first and foremost requirement is thus to avoid clouds and backend servers. Additionally, decentralization is mentioned as a motivation in section 1.1. This is the rationale for choosing a peer-to-peer architecture, where processing and storage is primarily done locally on the devices, and data communication is more direct.

4.3.2 Why Friend-to-Friend?

Swirlwave is based on a type of private, unstructured peer-to-peer networking called friend-to-friend. This is reasonable for distributed systems with a predefined set of dedicated nodes that should not be available to everyone. For example, assume that a company owns many smartphones administered by their employees. The company develops a custom, distributed system that runs on all the smartphones. The system processes data collected and stored on the smartphones locally. The company would most likely want their network of smartphones to be private, so that it is not available to everyone else using Swirlwave. A comparable situation applies to a social network of friends that chat with each other, send messages, and share diverse types of data. They would probably want to restrict access to their network to the ones they know. These are the reasons for choosing a friend-to-friend architecture.

4.3.3 Reasons for Individual Contact Lists

Because there are no central directory servers, and since there will be many separate private networks, each peer keeps its own record of the peers it knows—the contact list.

In real life, social networks are not isolated silos. People are usually part of multiple social networks; they have networks of friends, family, colleagues, and so on. Two friends do not necessary know all the same people. This is the basis for the architectural decision that friendships should be commutative, but not transitive. This is illustrated in Figure 2. Assume that the peers are sharing documents, and that each peer can see a combined list of all documents located at their friends. Peer B will see a list of documents from all peers, but Peer A and Peer D will not see documents from Peer C. In the Swirlwave architecture, peers have individual contact lists. Keeping an individual contact list per peer makes it is possible to imitate this type of non-siloed networks.

A mechanism is needed that lets peers discover that they have common friends. One example of how the concept of common friends is useful, is found in section 4.2.6. As mentioned, a peer can be member of many networks. Because of this, it is not always so that a friend of a peer should know about the existence of all other friends of that peer. Based on these requirements, the support for common friends is included in the architecture as follows: When two peers become friends, they will exchange a list of friends they already have, and that they do not mind that the new friend gets to know about. The list will be part of the contact list entry for the new friend, and it will be updated when that friend adds or removes friends. When a peer adds a new friend, a decision must be made about who should be informed. When removing a friend, all friends that also know this friend must be informed. The set of common friends is found by taking the intersection of a peer's own friends and the other peer's known friends.

4.3.4 Extensibility and Flexibility Requirement

Swirlwave is meant to be a tool that can be used to build peer-to-peer and distributed systems. It must therefore be extensible to fit a variety of custom use cases, such as the examples in section 4.3.2. This is why the system is designed as a middleware. The architecture makes it possible to develop highly customized applications and deploy these without making any changes to Swirlwave itself.

By layering Swirlwave in a middleware and an application layer, it is possible to build structured peer-to-peer topologies on top of the friend-to-friend overlay. This is necessary to make Swirlwave usable when developing a distributed system for smartphones, which usually requires a specific organization of the nodes. As part of developing a distributed system, it must also be possible to assign specialized roles—or capabilities—to nodes. This is the purpose of Swirlwave's plugin architecture. An example follows: In section 2.5, the concept of superpeers is mentioned. In Swirlwave, a superpeer could be defined by installing a server-application with superpeer-functionality on a specific node. This application will then be registered as a plugin with Swirlwave. As part of the plugin registration process, capabilities are specified. One of these capabilities is a service type identifier. A service type identifier can be said to identify a service contract between server and client. A contract is a promise that certain features will be available. Swirlwave provides an infrastructure for publishing service type identifiers, but it does not define the nature of the contracts implied by the identifiers. That is open for the application layer to decide. Swirlwave announces the registration information to all nodes in the contact list. The other nodes will then be able to locate superpeers by looking in their contact list for friends exposing the superpeer service type. In some cases, additional information is needed by the nodes that are using superpeers. Maybe there are several superpeers, but with different priorities. One superpeer could be the primary one, and the others could be backups if the primary superpeer fails. Properties like this could be published during registration, because applications can register custom

information as part of the capabilities, which for instance can contain data in *JSON*[67] or *XML*[68] format. This will be part of the superpeer's published capabilities. Alternatively, other peers could ask the superpeer for extra information directly through a *REST Web service*[69] or similar. The peers know that a superpeer exposes a certain service because of the service type identifier. The case of superpeers is just an example, but it illustrates the reason for Swirlwave's plugin architecture and infrastructure for publishing capabilities.

4.3.5 Choosing a Primary Communication Channel

Many communication channels are available to smartphones. This topic is discussed in the background chapter. Several of the channels are suitable for peer-to-peer networking, but almost none of them are wide-area technologies. Since most smartphones are carried around by their users, and most likely will be spread out in a geographical area wide enough to require wide-area networking, most of the technologies are non-viable options for Swirlwave.

Social networking is an important use case that often involves sharing of personal multimedia, documents, and other relatively large types of data with friends. This requires a channel that can transfer arbitrary sized data at acceptable speeds. The same is true for distributed systems that transfer data between nodes. Combined with the wide-area requirement, this is the reason why the Internet is used as the primary communications channel.

Nodes in peer-to-peer architectures typically have client and server functionality at the same time. For a node to provide server functionality, it must be reachable by other nodes. In conjunction with this, two distinct problems arise for smartphones when communicating over the Internet:

- Smartphones are normally behind NAT.
- Smartphones frequently change networks.

4.3.5.1 NAT Traversal Requirements

Although the nodes are assumed to be connected to the Internet, they are expected to lack a publicly visible IP address. The reason is that they are connected to a network behind a NAT. This is the typical case for mobile devices that connect to the Internet through cellular data or Wi-Fi. The same is true for most personal devices connected to the Internet, whether it be at home, at university, or at work. The consequence is that these devices cannot be reached directly by other devices. They can connect to others, but others are unable to initiate contact with them.

In the background chapter, several NAT traversal alternatives are discussed in section 2.11. After considering the alternatives, a choice was made to use Tor hidden services. The purpose of the Tor hidden services protocol is responder anonymity, but the protocol has an unintended side effect: It can be used for letting mobile devices connect directly to each other over wide areas, with acceptable transfer rates for arbitrary amounts of data. This fits the requirements of Swirlwave. The real goal of the hidden services protocol is to hide a server's true location, which is why the protocol is used on the so called *Dark web* [52]. To hide the server's location, the protocol creates a custom type of address,

---

[67] http://json.org

[68] https://www.w3.org/XML

[69] https://www.ibm.com/developerworks/library/ws-restful/index.html

an onion-address. The interesting side effect in the context of Swirlwave is that the use of onion-addresses at the same time removes the need for a publicly visible IP. The protocol can thus be used to reach devices behind proxies that use NAT.

Some additional reasons for choosing Tor includes:

- It is globally available, and practical to use.
- It is open and free of charge.
- It is scalable.
- It is being maintained by an active organization, but it is not owned by any single company.
- It is built for privacy and security, which are important requirements of Swirlwave. This is Tor's main purpose.

4.3.5.2 Requirement to Handle Network Changes

The idea of using hidden services was important to the system, but it was only part of the solution. With mobile devices, other challenges arise compared to using stationary servers. A mobile device will be moved around geographically. The user carries the smart phone with her. The device will be susceptible to lose its network connection, and it will frequently disconnect from the current network and connect to another one. It can switch back and forth between cellular data and Wi-Fi. In the process its network address will change. The hidden services protocol is not designed for this, as mentioned in section 3.3.

Managing the peer's ever changing addresses is one of the main contributions of the Swirlwave system. How the architecture solves these requirements is briefly described in section 4.2.6, and is described more in detail later in this chapter.

4.3.6 The Need for a Fallback Channel

Peers are expected to change addresses and even disconnect from the network. If two devices have been offline for a while and then reconnects, then it is possible that they both get new addresses. If they do not have any common friends, then none of the peers will have the correct address to the other peer. They will not be able to contact each other. This is the reason for designing a fallback protocol that uses an alternative channel, as described in section 4.2.6. The same section explains why SMS is a used as a fallback channel for smartphones. A detailed description of the SMS fallback protocol for exchanging addresses is given later in this chapter.

4.3.7 The Necessity of System Messages

There are no central nodes as part of the Swirlwave architecture. The middleware is based strictly on peer-to-peer communications. There are no central directory services, so the peers manage their own contact lists. The contact list contains information about friends that can be expected to change. Especially information about current addresses must be kept up to date, but also peer capabilities, shared friends, and other information can change. This information must come from somewhere. In Swirlwave this is solved by sending messages between peers. Since this is an integral part of the Swirlwave middleware, it should not be part of the application layer. It is therefore part of the middleware architecture as system messages that is sent between peers. The system messages are generally transmitted in the same way as application traffic, though the Swirlwave proxies. The exception is when a peer cannot be reached through the primary channel and the fallback channel is used.

A system message is also sent when a peer connects to another peer to consume an application layer service. This message is always sent as an initial handshake, and is used for authenticating the client. The authentication message is handled transparently to the application layer, and is only of concern to the client and server proxy.

4.3.8 Solving Privacy Requirements

Providing privacy for its users is one of Swirlwave's main goals. Solving this involves integrating digital security into the architecture. Digital security is a specialized field, but even for experts it is hard to get right. Absolute security is said to be an abstract concept. It is more realistic to discuss degrees of security, and aim to achieve a level that is acceptable for the context in question. This section discusses the choices made regarding confidentiality and authentication of data transferred between peers in the Swirlwave system.

The Tor network is built for anonymity and confidentiality. It encrypts all traffic. This provides a sufficient level of security when communicating over the Internet. What Tor lacks is encryption of data at its boundaries, where data enter and leave the Tor overlay network. In Swirlwave's case, this boundary is only found locally at the server and client side. This is because Swirlwave has local onion-proxies on both sides, and because the hidden services are running on the same device as the server-side onion proxy.

This solves the requirement that data must be securely transferred between peers to ensure privacy. It does not, however, solve authentication of clients on the server side. Such authentication can be added in the application layer, but since Swirlwave uses system messages, it must be able to validate the identity of a client as part of the middleware. This should be transparent to the application layer. This also makes it easier to use Swirlwave when building custom applications.

Swirlwave uses standard public-key cryptography, for authentication purposes, as well as for encryption when communicating over the fallback channel. All peers have their own key pair. The key pair is generated when the system is being installed, and is tied to that installation, and consists of one private and one public key.

Private keys are never given to others. The owner must make sure to keep private keys secret. The purpose of public keys is the opposite, they are supposed to be shared. The problem that arise with public keys is verifying the true ownership of the keys. If a user can be tricked into believing a public key belongs to a someone they trust, when it in fact is owned by the trickster, then this can be exploited in all manner of ways. When given a public key, trust is therefore all-important. The receiver should be confident that the alleged ownership of the key is correct. A trustworthy mechanism for exchanging public keys is required.

As a rule of thumb, one should use industry-accepted security features instead of inventing one's own. The standard for exchanging public keys for encryption and signing, and proving the ownership of a public key is digital certificates signed by a certificate authority (CA). This is described in section 2.14.4. Certificates cannot be used in the context of Swirlwave. A certificate must be associated with a specific domain name, the registration process is manual, and the certificates cost money. This renders certificates unfit as an option. Other alternatives exist, for example by using chains of trust. For the first version of Swirlwave, a much simpler solution was chosen.

Swirlwave transfers keys out-of-band, which means that users meet in person to exchange keys. The media for transferring public keys in Swirlwave is NFC. This technology can be used to exchange small amounts of data over a very short distance. Modern smartphones usually have built in NFC. Swirlwave will take care of exchanging public keys and other peer information when the phones touch each other. An alternative that was considered was Bluetooth. This could be used for laptops, stationary computers, and devices that do not have NFC.

4.3.9 Client and Server Proxies

This section explains why the Swirlwave architecture includes a client and a server proxy. The proxies are there to provide location and protocol transparency to applications, they handle system messages, and they provide automatic authentication. Figure 5 illustrates how traffic passes through the client and server proxies.

4.3.9.1 Client Proxy

A requirement of the system is extensibility. It should be easy to integrate new types of services that a peer can offer. A vast array of libraries exists that can be used to develop services that accept connections over TCP. The protocols supported by these libraries can be anything from HTTP[70], FTP[71], SMTP[72], to any other thinkable protocol implemented on top of TCP. When creating a client that consumes these services, it is desirable to use existing libraries corresponding to the protocol used by the service. One example could be a queue service that ships with an accompanying client library, such as *JeroMQ*[73]. It can be very time consuming to develop custom versions of client libraries especially tailored for Swirlwave. Another example is that it should be possible to integrate a web browser component to view web applications residing on peers. The system should hence be flexible enough to support the use of client libraries without modification.

The challenging part is that client libraries commonly connect to a server by using an IP address and port number. Peers do not have such IP addresses available. Swirlwave uses onion-addresses to connect to services via the Tor network. Furthermore, SOCKS4a [49] must be used to connect to Tor as a client.

Swirlwave's solution is the client side proxy. It is a layer that runs on the client and listens on a range of ports locally. The client proxy maps these local ports to remote peers. When a client uses ordinary TCP to connect to one of the local ports, the proxy will connect it to the remote peer. All traffic between the client and the server will be proxied over a SOCKS4a connection. The Swirlwave client proxy is a type of port forwarding proxy that accepts ordinary TCP connections and forwards traffic to the correct peer over SOCKS4a—in other words, it provides location and protocol transparency to the application layer clients.

4.3.9.2 Server Proxy

In this section, the reasons for including a server side proxy in the architecture are explained.

---

[70] https://www.w3.org/Protocols
[71] https://tools.ietf.org/html/rfc959
[72] https://tools.ietf.org/html/rfc2821
[73] https://github.com/zeromq/jeromq

On the server side of a connection, an onion proxy receives incoming traffic and forwards it to a local port. The destination is a service made accessible on a port by an application-layer service. With Swirlwave, it should be possible to run several different application-layer services at the same time on many different ports. As mentioned in section 3.3, Swirlwave uses a library from the Thali Project to programmatically manage the onion proxy on Android. This library only support publishing one hidden service at a time[74]. This means that only one single port can be forwarded by the onion proxy, and hence only one application-layer service can be exposed. It can be argued that an architecture should be independent of implementation details, but sometimes there are limitations related to the implementation of the system that affect the architecture. In this case, Swirlwave solves the problem by having a server side proxy between the onion proxy and the destination services. The server proxy receives traffic from the onion proxy, and then acts as a *reverse proxy*[75] that forwards traffic to the port that belongs to the destination application-layer service.

Traffic from the client is routed through the Tor network, which is made for anonymity. The consequence is that the identity of the client is unknown to the server-side. For Swirlwave, it is important to know that the client that connects is a known friend. Connection requests from unknown parties should be automatically rejected. Swirlwave must also know to which application-layer service the client wants to connect. This is solved in the architecture by the server proxy, as also mentioned in the last paragraph of section 4.3.7. When a client connects, an initial handshake is performed where the server proxy receives a message from the client containing, among other things, data encrypted with the client's private-key. The message is used for authenticating the client, and for determining to which application-layer service that the client wishes to connect. A more detailed description of this procedure is given later in the chapter.

[74] https://github.com/thaliproject/Tor_Onion_Proxy_Library/issues/5
[75] https://www.nginx.com/resources/glossary/reverse-proxy-server

4.4 Detailed Descriptions

An overview of the architecture has now been given, followed by an account of the reasoning behind the architectural choices. In this section, elements are described in greater detail.

4.4.1 Contact List

Every peer keeps its own contact list of friends. New contacts are added out-of-band. An entry in the contact list includes the following:

Table 1 – Contact List Record

Field Name	*Description*
Name	The human readable name of the friend
Peer ID	Every peer has an ID that is unique across all installations. The datatype is a 128-bit long universally unique identifier (UUID) [53].
Public-key	The public-key from the friend's asymmetric keys
Address	The friend's onion-address
Address Version	Each time a peer changes its address, it will increment the address version number. This number is given to friends accompanying the address.
Online Status	If the last attempt to reach the friend was unsuccessful, this status is set to offline, otherwise online.
Last Contact Time	The last time contact was made with the peer, which for instance can be used to sort the contact list from most to least recently seen friends.
Secondary Channel Address	In the current version of Swirlwave, this is the phone number used when sending SMS-messages to the peer.
Known Friends	This is a list of peer identifiers that the friend knows and have shared with the current peer. It is used to calculate a list of common friends.
Capabilities	A list of capabilities supported by this friend. Each entry in the list is a record with fields describing the capability.
Awaiting Answer from Fallback Protocol	This is used if the friend has used the fallback protocol when the current peer was offline. If the current peer comes online, and the friend is unreachable during address announcement–for instance due to changed address–then the peer must use the fallback protocol in the other direction.

Peer ID, Secondary Channel Address, and Public-key will not be changed. The Name-field can be changed manually on the device where the contact list is stored. The fields Online Status and Last Contact Time will be automatically updated by the local Swirlwave process. The rest of the fields can be changed based on information from the friend.

4.4.2 Establishing Connections

When establishing a connection to a Swirlwave server side proxy, the server proxy first sends a random number to the client. The client responds by sending a message that must be accepted by the server proxy before any payload can be transmitted. The message has three purposes:

1. It authenticates the client.
2. It specifies if the connection is for transmitting system messages or application-layer data.
3. It lets the proxy determine the destination service.

Table 2 - Connection Message

Fields	*Description*
Sender ID	The peer ID of the client
Random Number	A random number initially generated and sent by the server proxy
Message Type	Whether this is a system message or an application-layer connection
Destination	An identifier of a capability representing a service that the client wishes to consume. This will only be set for application-layer connections.
System Message	A system message that will be dispatched to a module that handles system messages. This will only be set for system message types.

4.4.2.1 Authentication

Tor hidden services provides end-to-end encryption to the system. The assumption is that Tor gives sufficient protection against eavesdropping and man-in-the-middle adversaries between devices. The missing part is authentication. Authentication is therefore included as part of the Swirlwave middleware.

In the connection message, the sender's peer ID is not encrypted. The rest of the message is encrypted with the sending peer's private-key. The server proxy looks up the peer ID in the contact list. If the peer is unknown, then the connection is rejected. If it is found in the contact list, the registered public-key is used to decrypt the message. If the message is successfully decrypted, then it is probable that the claimed identity of the client is correct. The decrypted part of the message contains the random number that initially was generated by the server proxy. If this number is equal to the one that was sent to the client, the connection is accepted.

One could argue that a connection message could be stolen and replayed by a man-in-the-middle, but as already mentioned, the Tor network is built with security in mind and is trusted to protect against eavesdropping. Messages could be stolen at the devices themselves. If someone manages to steal a message locally at the server side, then the encrypted random number will prevent the message from being accepted by other peers. The number in the encrypted part of the message would not match the random number generated by the other peers. A message could be stolen locally at the client side, by eavesdropping on the connection between the client proxy and the local onion proxy. Then the initial random number sent from the server will also prevent the message from being reused. If the message is stolen before it is sent to the local onion proxy, then it is likely that the adversary has extensive control over the device, and the security is already broken.

4.4.2.2 Message Types and Destinations

As shown in Table 2, the connection message has information about the message type, destination, and system message data.

If it is an application-layer connection, the server proxy will use the identifier in the destination field to match a local service endpoint. It will then set up a connection to this service, and relay all traffic between the connection from the client and the connection to the service.

If the message is a system message, then the content of the system message field will be dispatched to an internal module in the Swirlwave middleware that handles system messages.

4.4.3 Applications, Plugins, and Capabilities

Applications built on top of Swirlwave communicate over TCP. Clients connect to a listening server and communicate with it by using a protocol on top of TCP. To conduct meaningful communication with the server, the clients must use the same protocol as the server.

In Swirlwave, applications are said to have capabilities. One capability is that the application can be used as a server, another is that the application can be used as a client. An application can support both at the same time, but this is not a requirement; applications can be split in separate client and server parts.

The supported protocols are also capabilities. A client and server that know the same protocol can talk together. Swirlwave does not dictate which protocols are supported. Applications can use which ever protocol that is suitable.

This flexibility is made possible by identifiers. Applications register as plugins with Swirlwave along with their capabilities. The protocols supported by the application are represented as universally unique identifiers (UUID) [53]. More generally, they are identifiers of contracts or agreements that server and client must comply to in order to properly communicate. Swirlwave does not care about the details of this contract or agreement, but it simply uses the identifier to match clients and servers. In addition to the identifier, a human readable name is provided.

In the following example, assume that a Swirlwave user wants to send a message to friend. The user can select the friend from a list in the Swirlwave user interface. Swirlwave will then show a list of the communication types that are available. This is based on the registered identifiers found in the contact list entry for that friend. If the user has an application that can be used as a client, then Swirlwave knows about it by matching the identifiers of the locally installed applications with the identifier of the

friend's messaging service. Swirlwave can then open the messaging application on the client when the user taps on the name of the messaging service in the list.

The same principle can be used programmatically by applications by letting them query friends lists and examine exposed capabilities.

Custom information can sometimes be required for the clients and servers to function, for instance configuration settings that need to be set on a per peer basis. Because of this, an extra field is provided as part of the published capabilities. This field can contain arbitrary data.

4.4.4 Address Changes

When a smartphone changes from one network to another, for instance from Wi-Fi to cellular data, its access point is not the same as before. The IP address will most likely be different, and the route to the device will most certainly be different.

Swirlwave uses onion-addresses. The use of onion-addresses make it possible to connect to peers, even if their IP addresses are unreachable from the Internet. Nevertheless, routing is dependent on IP addresses under the hood, just as everything else on the Internet. It is possible to reuse an onion-address so that it resolves to a new access point, but there is a problem: Clients who have obtained a route to a hidden service will not update that route. Already running clients will continue to route to the old location, which is no longer valid. There is no support in the Tor control protocol for letting the client refresh the route to a hidden service.

Swirlwave's solution is to monitor network changes on the device, and registering a new hidden service when the smartphone connects to a new network. If a network and access point address is recognized from earlier, then the hidden service and onion-address from last time is reused.

4.4.4.1 Announcing Address Changes

Address changes must be announced to friends. A device that has been offline, or has changed its location, will contact its friends as soon as it is online again. The address is passed with a version number. This version number is increased every time a peer changes its address. The purpose of this version number is comparable to a Lamport [54] logical clock; it can be used to determine if one address is older than another when comparing registered addresses across peers.

If the peer has been offline for a while, it is not unlikely that some of its friends have changed addresses. The peer will not know the friends' new addresses, and will not be able to reach them. It then becomes the unreachable friends' responsibility to obtain the address it needs it. Alternatively, the peer will try to obtain the address when it needs to contact the friend later.

4.4.4.2 Obtaining Changed Addresses

If a connection is broken while being in use by an application-layer client, Swirlwave does not try to reconnect automatically. It is the responsibility of the application-layer to reconnect and resume transfers in the current version of Swirlwave.

Swirlwave does, however, have the responsibility to keep track of peer addresses.

Figure 8 is a sequence diagram that illustrates the scenario where Peer A no longer has a reachable address to Peer B. Assume that Peer A, Peer B, and Peer C are friends:

1. Peer A tries to open a new connection to Peer B, but fails after trying a couple of times. Peer B is marked as being offline in the contact list.
2. Peer A checks its contact list entry for Peer B. There it finds common friends, which in this case is Peer C. It asks Peer C for the address to Peer B. This is because Peer B may have given a new address to Peer C, but failed to give it to Peer A. The address is returned from Peer C together with the version number, encrypted with Peer C's private-key. If Peer A successfully decrypts the returned information, and discovers that the version number is higher than the one it already has, then it updates its contact list and proceeds to step 3. If not, it goes straight to step 4.
3. Peer A tries to open a connection to Peer B by using the address obtained from Peer C. If Peer B is reached, then it is marked as online in the contact list and the last contact time is updated. If it still is unreachable, Peer A proceeds to step 4.
4. Peer A uses an alternative channel to contact Peer B. Peer B can be either online or offline, but either way it will respond to Peer A as soon as possible with its current address. Peer A will then mark Peer B as online in its contact list, and update the last contact time.

Note that if Peer A and Peer B do not have any common friends, then step 2 and 3 would not apply.

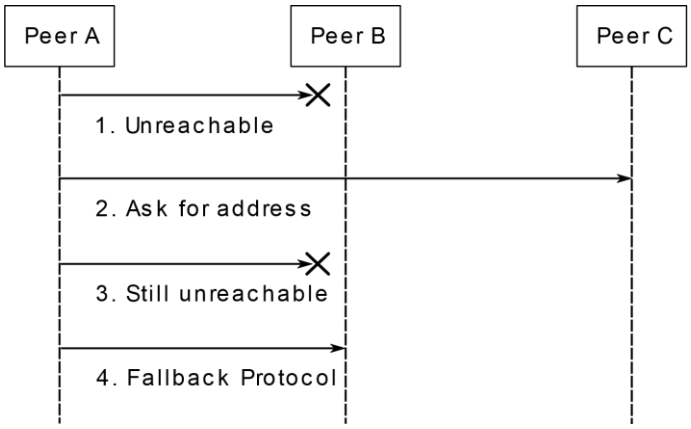


Figure 8 - Obtaining Changed Addresses

4.4.5 SMS Fallback Protocol

In this section, a protocol is described that obtains a peer's address by contacting it directly via an alternative channel. The alternative channel is SMS.

The peer that initiates the communication is called peer A. Peer A is connected to the Internet, and her services and apps are working property. There are no network problems. The peer to which A tries to connect is called B.

1. A has tried to connect to B without any success.
2. A looks up B' s phone number in the contact list.
3. A sends B an SMS with her address. The address is encrypted with A's private key. The message also contains a secret one-time code. This code has several purposes. It is a salt, preventing exactly equal encrypted messages to be sent more than once, and it is a combined message-ID and anti-forgery token sent back to A later.

4. B receives the SMS.
    a. B looks up the sender's phone number in his contact list, and by judging from the phone number, the message is from A.
    b. B decrypts the message with A's public key. Both the phone number and public key was originally transferred by using NFC, one time when they met face-to-face. The corresponding private key is only known by A. The combination of correct phone number and correct encryption key is therefore taken as a confirmation that the message really came from A.
    c. B updates his contact list with A's address sent by SMS, if the addresses differ.
    d. If a one-time code generated by B in conjunction with a previous message in the opposite direction exists, from B to A, it is removed. B connects to A's address.
5. B transfers his new address to A together with the one-time code that A originally send in the SMS. The message is encrypted with B's private key.
6. A decrypts the message by using B's public key that she keeps in her contact list. The secret one-time code that she originally sent is validated. The one-time code is deleted from storage.
7. A updates her contact list with B's new address.

If B fails to connect to A in step 5, then the protocol moves back to step 2, but with the roles of A and B reversed.

4.4.5.1 Connection Failure Reasons

The first step in the protocol states that A fails to connect to B. These are the cases supported by the protocol.

- If B simply has moved to another network, and thus his address has changed, then the SMS will be read at once. A will be notified with the correct address.
- If the reason that B cannot be reached is that he has turned off Swirlwave, then the SMS will be handled as soon as he starts Swirlwave again.
- If B cannot be reached because he does not have an Internet connection, then the SMS will be handled when he comes online again.
- If the reason that B cannot be reached is because the phone is off, then the SMS will still be delivered to B’s phone when it is turned on again. Swirlwave will read this SMS as soon as it is started again.

4.4.5.2 Cost of SMS

SMS can be associated with a cost per message, so the protocol is designed to minimize the number of SMS-messages that needs to be sent. It is also a point that the burden of the cost is put on the peer that initiates the communication. A is charged for the SMS sent to B in step 2, because she is the one trying to get in contact. If B cannot connect to A in step 5, then it is possible to let a configuration setting decide if B wants to spend money on sending an SMS to A. In that case, A must retry step 3 at a later point.

4.4.5.3 Advantages of the Protocol

The decision to use SMS may at first seem unorthodox, but the environment in which Swirlwave runs is different than for most other systems. It was designed with smartphones in mind, and SMS is available on nearly all smartphones. A smartphone does not change its phone number just because the

user moves between networks. There is no need to know the other phone's location other than the phone number. A phone number is a stable address.

The protocol does not require extra hardware, servers, or software acting as hubs, beacons, centralized or distributed directories. Remember that one of the main goals was to avoid communication via centralized clouds and servers.

The protocol works even when only two peers exist. It keeps the number of peers needed to locate the new address to a minimum. Gossiping, hand-offs, and other techniques [29] were considered, but they were all considered more complicated, less secure, less reliable, and they require the involvement of more than two participants.

It works even if the smartphones are far apart. They do not have to be on same local area network. They do not have to use radio-signals, Bluetooth, NFC, or similar alternative channels where the distance is a limitation. SMS works everywhere.

Essentially, the protocol uses SMS as a persistent queue. Persistent queues have advantages when used for communication between nodes in a distributed system, such that the communication is asynchronous, robust, and that sender and receiver does not need to be running at the same time.

When it comes to man-in-the-middle attacks, it is fairly secure, especially when combined with public-key encryption to avoid eves-dropping and message forgery. For instance, the combination of phone number and public key must be correct, and the anti-forgery token.

SMS was thus found to be a suitable communication channel as part of a fallback mechanism for locating mobile entities.

4.4.5.4 Why not use SMS all the time?

There are disadvantages to using SMS, rendering it unsuitable as a primary communications channel.

- It is slow compared to communicating over the Internet.
- The messages have a very limited length.
- Sending an SMS can cost more than sending the same amount of data over the Internet.

It can be noted that email was considered as an alternative channel, but was not included in this version of Swirlwave.

5 Implementation

5.1 Introduction

Swirlwave is implemented for smartphones running the Android operating system, version 4.4 or higher. Android provides an official application framework based on a Java language environment[76]. Swirlwave is therefore implemented in Java. A brief background on developing for Android is given in section 2.17.1.

The app has a front end graphical user interface and a background service. The background service is the work horse of the app. It has the responsibility for most of the features described in chapter 4.

To start the background service, the user first starts the app. The user is then presented with the app's main screen. From the main screen, the background service can be started by flicking a switch button to the right. The service will then start, and its status can be seen in the smartphone's notification bar. If an Internet connection is available, then the service will start the Tor onion proxy, followed by the Swirlwave client and server proxy. The service will continue to run even if the user interface is closed, and even when the screen is turned off or locked[77].

Messages from the background service will be shown in the notification bar. There the user can see the connection status of the service, and notifications about events. The user can click on the notifications to open the app's front end user interface again. The background service can be shut down by using the same switch button that started it.

From the main screen the user will see a contact list with all peers that they know. Detailed information about the peer will show up by clicking on it.

New peers can be added by NFC. This is done by opening the app on one phone, and then keeping the back sides of the phones at about 4cm or less.

Figure 9 gives a conceptual view of the modules and libraries relevant to the implementation.

[76] https://developer.android.com/guide/index.html

[77] The last paragraph of section 2.17.1.6 describes an issue concerning permissions in some Android-versions.

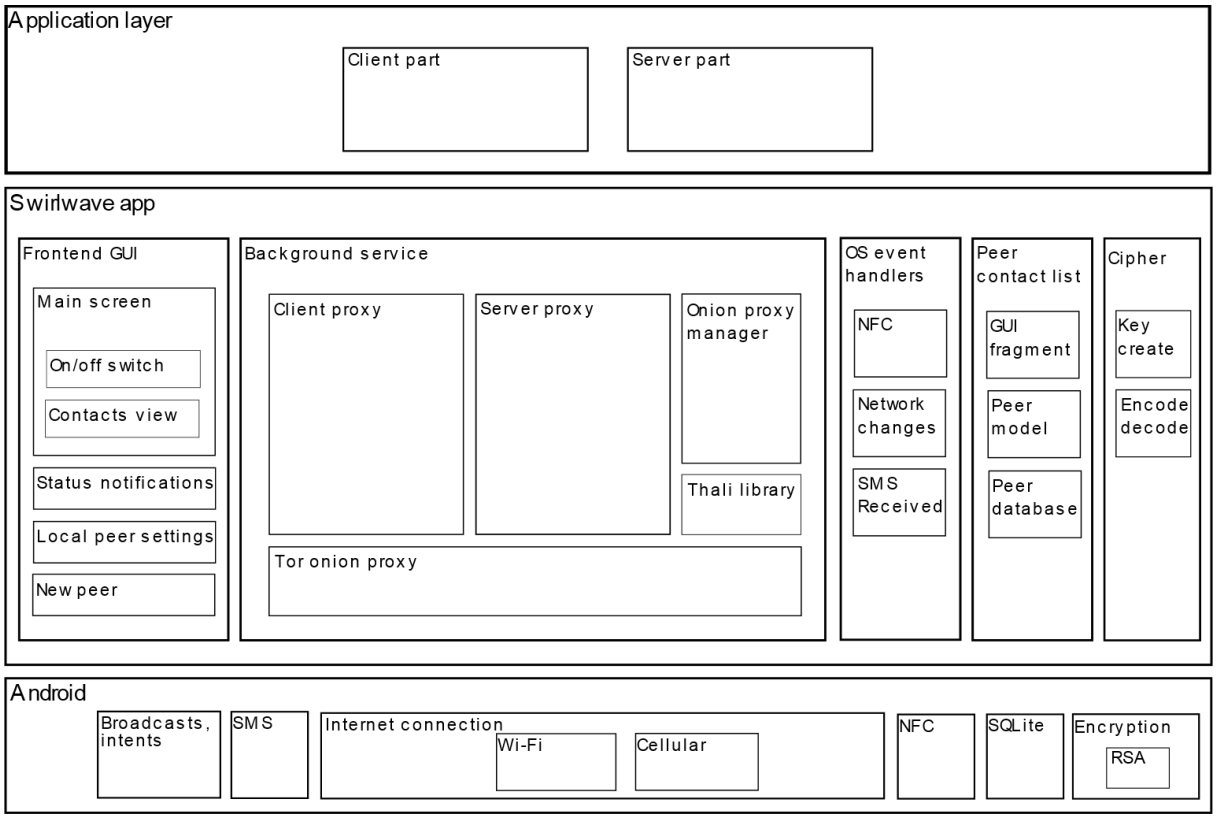


Figure 9 - A conceptual view of the implementation

5.2 Project Organization

Swirlwave was developed using *Android Studio*[78], the official IDE for Android. The project follows the canonical directory structure for Android projects. Git[79] was used as a source code version control system during development. The source code for the project will be made available as a public repository on *GitHub*[80] on the following address at a later time.

Class files are found under the folder *app/src/main/java*. The base package for the Swirlwave classes are *com.swirlwave.android*, and classes are generally organized by features, not layers. This is a style that has gained popularity in recent years. For instance, classes for managing the peer contact list is found under the package *com.swirlwave.android.peers*. The data access class *PeersDb*, the domain layer class *Peer*, the UI fragment class *PeersFragment*, and the adapter class *PeersFragmentAdapter*, are all in the same package. Nevertheless, some cross cutting features also match layers, such as general database access functionality which is found under *com.swirlwave.android.database*.

Android applications typically include resource files, such as layout definitions, localized strings, and other values in XML-format, icons, and graphics. These are in the folder *app/src/main/res*.

Swirlwave uses localized string resources. All text that is shown to the user is available in two languages: English, and Norwegian. If the user has a Norwegian version of Android, then the texts

[78] https://developer.android.com/studio/index.html
[79] https://git-scm.com
[80] https://github.com

will be in that language, else they will be in English. The language files are found in *app/src/main/res/values-nb/strings.xml* and *app/src/main/res/values/strings.xml*.

A fork of the Thali project on GitHub was made for Swirlwave. It is important to clone this repository and install it to a local *Maven*[81] repository. The Swirlwave projects uses *Gradle*[82] for building, but it imports the Thali-dependencies from the local Maven installation. The instructions can found in the file *README.md* in the project's base directory.

5.3 The Main Activity

Swirlwave's main screen, or *main activity*, is represented by the class *MainActivity*. It is declared as the app's main activity in the manifest file *AndroidManifest.xml*. There the intent filters for the activity are declared as well, such as an intent filter for NFC. The visual appearance is defined in a layout-XML file, *main_activity.xml*.

The main activity has a switch for turning the background service on and off.

The main activity also embeds a fragment, *PeersFragment*, that shows a list of friends retrieved from the contact list. The fragment has its own layout files: *peer_list.xml* and *peer_list_item.xml*. The fragment uses the class *PeerDb* to select data from the contact list database. The data is mapped to its list view by using the adapter class *PeerFragmentAdapter*.

The main activity has additional responsibilities:

When the app is started, it will check that all requested permissions have been accepted by the user. The needed permissions are part of the Android manifest, but the activity also checks that they have been approved, and it can ask for permissions at run time. The class *AppPermissions* was implemented for this purpose.

Another responsibility is to ensure that information about the local peer is registered. Upon starting the app for the first time, the main activity will open an activity that asks the user to fill in required information. This is implemented in the class *LocalSettingsActivity*, and the layout is defined in *activity_local_settings.xml*.

5.4 Local Settings

Each peer has a set of local settings that roughly corresponds to an entry for a friend in the contact list. One significant difference is that the local settings contain the private key for the installation, as well as the public key.

Local settings are managed by the Swirlwave-class *LocalSettings*, which uses *andoid.content.SharedPreferences* to store the app's settings. Some settings are created automatically, while others must be registered manually. Display name and phone number are the two settings that must be entered manually by the user through the user interface. Phone numbers cannot be reliably obtained programmatically, so unfortunately the user must enter the number. The format of the entered

[81] https://maven.apache.org
[82] https://gradle.org

number will be validated, and then rewritten to an international format by using the class *com.google.i18n.phonenumbers.PhoneNumberUtil* from a library created by Google.

The automatically generated values for the local settings include the Peer ID for the installation, and the asymmetric-keys. The Peer ID is generated by calling the method *randomUUID* of the *java.lang.UUID* class. The asymmetric keys are generated using the Swirlwave-class *AsymmetricEncryption*.

5.5 Cryptography

The Swirlwave-class *AsymmetricEncryption* implements the functionality needed to create asymmetric keys, and to encrypt and decrypt. The underlying classes used for cryptography are from the Android-implementation of the standard Java-libraries *java.security* and *javax.crypto*.

Swirlwave uses RSA encryption. The details can easily be changed, but the keys are currently 2048-bits, and transformation is set to *RSA/ECB/PKCS1Padding*.

5.6 Background Service

The app includes a background service. It is started from the main activity, but it continues to run when the user interface is closed. The service is implemented in the class *SwirlwaveService*, which inherits from *android.app.Service*. This class delegates much of its functionality to the class *SwirlwaveServiceHandler*, which inherits from *android.os.Handler*. A handler is a thread equipped with a message queue. Handlers can for instance be used for code that must be run on the UI thread.

The responsibility of the background service is to start and stop the onion proxy, the client proxy, and the server proxy. It also registers a broadcast receiver class, *SwirlwaveBroadcastReceiver*, for monitoring and handling network connectivity changes. This broadcast receiver will send an intent addressed to the background service with a custom action, *ACTION_CONNECTIVITY_CHANGE*.

The service sends the intent as a message to the *SwirlwaveServiceHandler*. The handler uses the Swirlwave-class *NetworkConnectivityState* to determine if the device is online or offline, and if it is connected to Wi-Fi or cellular data. It is also used for detecting if the location has changed since last connect, and other connection related features.

The service and service handler use instances of the classes *SwirlwaveOnionProxy*, *ServerProxy*, and *ClientProxy* to control the local onion proxy, the server proxy, and the client proxy.

5.7 Notifications and Toasts

To display the connections status of Swirlwave and other information to the user, the backend service uses status notifications. *Status notifications* in Android are shown in the status bar, which can be seen on the top part of the screen. By touching the top of the Android-screen and dragging the finger downwards, the notification messages can read. The class *SwirlwaveNotifications* was created to ease the use of notifications from the backend service.

When the background service is running, a permanent notification for Swirlwave will be shown. The message inside the notification will change, for instance according to the current connection status. If a user taps the notification, the main activity for Swirlwave will be opened.

In Android, a *toast* is a text message to the user. It is shown inside a dark rectangle that is faded in and out again. Since the implementation of Swirlwave made for this thesis is a prototype made for experimental purposes, toasts are used to give status messages about what happens internally in the app. One problem with this is that Swirlwave uses threads for performing tasks, but toasts can only be shown in the main thread, or more specifically the UI thread. Because of this, a class *Toaster* was created that posts a *Runnable* to the main thread that shows the toast.

5.8 Contact List

The contact list is represented by the classes in the package *com.swirlwave.peers*. General database functionality is also used, and this is implemented in the package *com.swirlwave.database*.

Android is shipped with the database engine *SQLite*[83]. The classes for using SQLite in Android is found in the package *android.database.sqlite*.

Swirlwave uses a single table, *peers*, for the contact list. The columns are equivalent to the fields described in section 4.2.5. with the addition of a primary key field, *_id*, that is an autoincremented integer. The class *PeersDb* takes care of the SQL-statements for creating the table, and for creating, updating, and selecting rows.

Peer is a *domain class*[84] representing a peer. It is implemented as a so called *plain old Java object* (POJO)[85] with getters and setters, and a couple of type conversions, to make it possible to instantiate objects in Java.

There are two more classes in the package. They are used for the user interface, and is described in section 5.3.

5.9 Swirlwave Onion Proxy Manager

The class *SwirlwaveOnionProxyManager* is used for managing the Tor onion proxy, and uses the class *OnionProxyManager* from the Thali-library. It has methods for starting and stopping the proxy, and can be used to get the current address and port of the onion proxy.

It is *SwirlwaveOnionProxyManager* that updates the current address in the local settings, and increments the address version number.

5.9.1 Reusing Onion-Addresses

When a smartphone changes network, Swirlwave will change the hidden service address (onion-address). The reason for this is explained in the fourth paragraph of section 3.2.

Swirlwave does this by using its class *NetworkConnectivityChange* to inspect details about the current network and create a file system friendly name based on these details. This name can be used as a directory name. The file system friendly name of the network is given as input to the *SwirlwaveOnionProxyManager* when starting the onion proxy.

---

[83] https://www.sqlite.org

[84] http://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/analysis/IdentifyingDomainClasses.htm

[85] https://www.martinfowler.com/bliki/POJO.html

The onion proxy will create the directory, if it does not already exist, and use it for storing files that it uses. If the directory is new, and thus empty, the end result will be that the onion proxy registers a new Tor hidden service. Because the Swirlwave-generated directory name is based on several details about the current network, a new Tor hidden service and onion-address will be given to networks that have not been connected to before by the smartphone.

On the other hand, if the network has been connected to before, a directory for it will exist. In that case, the already generated Tor hidden service and onion-address will be reused. This reduces the startup time for the onion proxy, since the hidden service does not need to be registered again.

5.10 Swirlwave Proxies

This section describes how the Swirlwave proxies were implemented.

The client and the server proxy have many similarities. They are both socket servers running in their own threads. The socket servers are implemented as single-threaded. Non-blocking sockets from the Java *NIO* [55] library is used to handle multiple sockets in the same thread. With traditional sockets a new thread must be opened for each incoming socket that connects. Also, one thread is needed per port that is listened to by a socket, and the client proxy listens to a range of port. For Swirlwave, this could result in many long running threads. Threads come with a cost. According to the official Android development web pages, one should not use more threads than needed[86]. Swirlwave will often start short-lived threads, but the total number of concurrent threads is kept lower by implementing the socket servers as single-threaded, which avoids having many long-running threads at the same time.

The proxies are started and stopped by the Swirlwave background service, but are not restarted as a result of connection changes. The listening server sockets will be kept open as long as the background service is running. This is to avoid a problem where old server sockets are not releasing their ports, so that new sockets are prevented from binding to the ports until the OS release them.

When an incoming connection is accepted by the proxy, it will establish an outbound connection. For the client proxy, the incoming connection will be from a local client, and the outbound connection will be to the onion proxy. For the server proxy, the incoming connection will be from the local onion proxy, and the outbound connection will be to the local application-layer server. The most basic functionality of the proxy is then to pass bytes between the incoming and outbound connections. Additionally, both proxies also manage the built-in authentication mechanisms, and Swirlwave's system messages.

5.11 Client Proxy

The client proxy listens to connections from local clients on a range of ports. It opens one port for each service for each friend. When a client connects, the port number will be used to find the correct onion-address and requested service on the remote peer. Relevant information will be retrieved from the contact list. The client proxy then connects to the locally running onion proxy.

To connect to an onion proxy, the SOCKS4a [49] protocol must be used. The client proxy implements the protocol by assembling and sending the correct header bytes to the onion proxy and then waiting

[86] https://developer.android.com/topic/performance/threads.html

for a response code. The header contains the friend's onion-address, and this is how the onion proxy knows which hidden service is requested. The socket to the onion proxy is what the client proxy uses when it later sends and receives data from the peer in the other end.

If the onion proxy returns a response code (0x5A) telling that it successfully connected to the remote hidden service, then the client proxy will wait for the server proxy to return a random, four-byte number. This number will later be returned to the server as part of the authentication message. After the number has been received, the client proxy creates an authentication message that will be sent to the server proxy on the other peer. It creates the message by instantiating an object of type *ConnectionMessage*. The class has properties corresponding to the fields described in Table 2. It also has a method for converting the message to an array of bytes. The connection message bytes will consist of a 128-bit UUID in plaintext, followed by the rest of the fields in the message encrypted with the client's private key. Methods for encrypting and decrypting bytes are implemented in the Swirlwave-class *AsymmetricEncryption*, which uses on the standard *java.security* and *javax.crypto* libraries for performing RSA-encryption. Before sending the actual message bytes to the server proxy, the length of the byte array is prepended, so that the server proxy knows how many bytes to read before reconstructing the message from the bytes. If the server proxy responds with a success code (0x10), then the client proxy will start reading and writing bytes between the incoming socket from the application-layer client and the outbound onion proxy socket.

If the onion proxy fails to connect to the remote hidden service, then the client proxy will mark the friend as being offline in the contact list. It will then try to request an updated address. As a last resort, it will send an SMS to the friend, asking it to get in touch and send an updated address. The client proxy will not try to establish new connections to the friend until it has responded. The friend will then be marked as online again.

5.12 Server Proxy

The server proxy also listens for incoming connections from the local host, but only on a single port. This is because the only client to the server proxy, is the onion proxy. When the onion proxy receives connections to the hidden service, it will connect to the server proxy on a specific port by using an ordinary TCP socket.

The first thing that the server proxy does when a connection is established by the onion proxy, is to generate a random number in the form of four bytes. For this it uses a static member variable of type *java.util.Random*. Afterwards it starts reading bytes from the socket as soon as they become available. These bytes are expected to be a connection message originating from a client proxy. Since all bytes may not arrive immediately, the bytes are stored in an instance of the class *ConnectionMessageSelectionKeyAttachement*, and the server proxy keeps track of how many bytes that have been read, and if all bytes of the message have been read. When all bytes have been received, the unencrypted UUID of the friend is used to retrieve the public key of the friend from the contact list. The encrypted part of the message will then be decrypted, and the random number in the message will be compared to the one originally sent to the client proxy. The message will be converted to an instance of *ConnectionMessage*.

The server proxy will then inspect the type of message. The types of messages are implemented as an enum, *MessageType*. If it is of type *ADDRESS_ANNOUNCEMENT*, the update address for the friend is extracted from the bytes in the *SystemMessage* field of the *ConnectionMessage*-instance, and given

to an instance of *FriendAddressUpdater*, which is started in its own thread. If the message type is *APPLICATION_LAYER_CONNECTION*, then the correct local port for the requested application-layer service will be resolved and a connection to it will be established by the server proxy. The server proxy will then start reading and writing between the incoming onion proxy socket and the outbound connection to the local application-layer service.

5.13 Announcing Address Changes

AddressChangeAnnouncer is a class used to connect to friends and give them the current onion-address of the peer. Like several other classes used by the client and server proxies to perform time consuming tasks, it implements *java.lang.Runnable*, so that it can be run in a separate thread. The reason for doing this is because the Swirwave proxies are single threaded, and the time spent on processing each step in a proxy should be limited. Some tasks are thus offloaded to worker threads.

The address announcer instance will then connect to friends and give them the current onion-address of the peer. As part of this, it authenticates to the remote server proxy in the same way as the client proxy does. It uses the class *ConnectionMessage* to create a message of type *ADDRESS_ANNOUNCEMENT*, so that the server proxy will know that this is a system message for updating a friend's address in the contact list.

The address announcer can either send an update to all friends, or a specific friend. When the client proxy starts, all friends will be updated. When a data SMS is received from a friend, as a result of a fallback address request, then this single friend will be updated at once, given that Swirlwave is online. Note that the class also checks if a friend has been marked as waiting for an answer from the peer as result of a fallback protocol.

On the server proxy end, another runnable class is used to update the contact list. This class is *FriendAddressUpdater*. This class updates the contact list database by using the class *PeersDb*. It also has code for validating that an address has the correct format, and for updating the online status of a friend.

5.14 SMS

When the client proxy discovers that it cannot connect to a friend, it will try to send a data SMS to the friend. In contrast to an ordinary text SMS, a data SMS will not be visible to the user. Instead it will be received directly by Swirlwave.

5.14.1 Sending

The class *SmsSender* was implemented for sending SMS in Swirlwave. To activate the SMS fallback protocol, a new instance of SmsSender is created and started in its own thread.

To send the data SMS, the method *sendDataMessage* is called on an instance of the Android-class *android.telephony.SmsManager*. When sending a data SMS, the data type of the message is a byte array, and a destination port is given as parameter. On the receiving smartphone, the port will be used by Android to invoke the correct application, which in this case is Swirlwave.

The SmsSender class is also used in conjunction with the class *AddressChangeAnnouncer*, which will send an SMS if an unreachable friend has used the SMS fallback protocol and is currently waiting for an answer.

5.14.2 Receiving

Swirlwave implements a class *SmsReceiver* to receive and process data SMS. The class inherits from *android.content.BroadcastReceiver*. The class is registered as a receiver in the Android manifest that handles the action *android.intent.action.DATA_SMS_RECEIVED* for port number 6739. The number is just a randomly chosen integer used by Swirlwave.

Android will invoke the SmsReceiver when a data SMS with destination port 6739 is received, even if the app is not currently running.

The received message will contain the address of the sender. If the message is accepted, then the address is updated in the contact list. The friend is also marked as awaiting an answer. If Swirlwave is running and connected, then the friend will be answered immediately over the Internet with the current address. Otherwise it will receive the answer with the peer's address as soon as Swirlwave connects again. Note that the friend now can be offline or have changed its address. In that case, if it is marked as awaiting an answer, then an SMS will be sent to it.

5.15 Near Field Communication

Handling NFC consists of two parts: Receiving data, and sending data.

In the Android manifest, an intent filter is added to the main activity so that it will be called when it is discovered that NFC data is available for Swirlwave. This is done by filtering for the action android.nfc.action.NDEF_DISCOVERED with a custom Swirlwave mime-type of *application/vnd.com.swirlwave.android.beam*. When a contact information is received from a new friend via NFC, the main activity's *onResume* will be called. In this method, it is discovered that the method was called as a result of NFC by inspecting the received intent. The transferred data is then read. Since the data transferred over NFC is a text string, the peer information has been serialized as a *JSON*[87]-string. The library *Gson*[88] is used for serializing and deseralizing peer information. The new peer is added to the contact list by using the class *PeersDb*. If a peer already exists, it will be updated instead of added.

Sending contact information over NFC is a bit different. The main activity implements an interface that is part of the Android framework called *NfcAdapter.CreateNdefMessageCallback*. When data is ready to be transferred to the other smartphone via NFC, the implemented callback *createNdefMessage* will be called. In this method, Swirlwave first makes sure that Swirlwave is online, so that it has a valid onion-address. It then creates an instance of the *Peer*-class. The property values for the peer will be set to the corresponding values in *LocalSettings*. The *Peer*-instance is then serialized to JSON, and converted to a byte array. An *android.nfc.NdefRecord* is created with package name *com.swirlwave.android*, mime-type application/vnd.com.swirlwave.android.beam. The content is set to the bytes from the JSON-serialized *Peer* instance. The record is added to an *android.nfc.NdefMessage* that is returned. Android will then take care of transferring data via NFC to the other smartphone.

[87] http://www.json.org

[88] https://github.com/google/gson

6 Experiments

In this chapter, the implemented Swirlwave prototype is used for experiments. All experiments are implemented in an application-layer client and server that is used to automatically measure performance.

6.1 Background

Swirlwave is designed for communicating over the Internet in a peer-to-peer manner with smartphones. The experiments hence focus on measuring performance factors concerning networking.

What is considered good or substandard performance will greatly vary—it is subjective, and it depends on the type of network, the Internet subscription, the location, and it changes over time.

Because of this, experiments are conducted by measuring the performance with and without the system involved, or parts of it, and comparing the results against each other.

It is also important to remember that Swirlwave depends on the Internet as its surrounding environment. Many factors affecting network performance will vary in ways that are infeasible to control, predict, or analyze. As network packets travel through the Internet, they will be affected by varying amounts of traffic from other hosts, sometimes congestions or packet losses will occur, and the packets will be routed in unforeseeable ways.

To mitigate the effect of unpredictable Internet connections, an attempt is made to make the environments as similar as possible during the experiments. This is essential when making measurements with and without parts of the system that are going to be compared. Therefore, the 4G and Wi-Fi networks are the same for all experiments. The smartphones keep the same roles for all experiments. Experiments that collects measurements that are meant to be compared, are carried out at the same time. The experiments are also conducted multiple times to provide a basis for basic statistical calculations, such as median, and 95^{th} percentile.

The same two smartphones are used for all experiments:

- Huawei P9 Lite
    - Android Marshmallow 6.0
    - Connected to cellular data (4G)
    - Used as client
- Samsung Galaxy Note 4
    - Android Marshmallow 6.0.1
    - Connected to Wi-Fi
    - Used as server

An Internet subscription with a static, public IP is used in the experiments, and the wireless router is manually configured to forward from a specific port to one of the smartphones. This means that the smartphone can be contacted directly from the Internet, which for instance makes it possible to test the system with and without hidden services in a relatively similar environment. The other smartphone is connected to the Internet via cellular data. This is to ensure that the two smartphones do not connect directly inside of the local area network. All traffic will go through the Internet, but since the same

Internet connections for 4G and Wi-Fi are used for all experiments, and since the related experiments are carried out approximately at the same time, the results should be reasonable comparable.

6.2 Starting Onion Proxy

This experiment measures how long it takes from the onion proxy is started to the hidden service is registered and ready for use. The difference between starting a new hidden service and reusing an existing one is compared.

Table 3 - Onion Proxy Start-Up Times

	Median	*90th Percentile*	*Num. Trials*
New Hidden Service	18.080s	43.941s	10
Reused Hidden Service	8.401s	8.855s	10

6.3 Establishing Connections

The question in this experiment is: How long does it take to establish a connection between the client and the server in the following cases:

- Connecting via Swirlwave, including the time it takes to authenticate the client.
- Both client and server uses Orbot. Server side uses a manually registered hidden service.
- The connection is established directly over the Internet.

Table 4 - Time Establishing Connections

	Median	*95th Percentile*	*Num. Trials*
Connecting via Swirlwave	1.829s	3.557s	100
Hidden Service w/Orbot	1.384s	2.931s	100
Directly over Internet	1.827s	3.597s	100

6.4 Throughput

The *throughput* is a measure of how fast we can actually send data through a network [56]. If a connection has been established, and the client has been authenticated, how long does it take from the client starts reading the first byte to 12.5MB has been read under the given conditions?

- With Swirlwave
- Client and server is using Orbot. Tor hidden service is manually registered.
- Directly over the Internet to a smartphone with public IP, without using Tor

The throughput is calculated from the measurements.

Table 5 - Throughput

	Median	*95th Percentile*	*Num. Trials*
Swirlwave	2.500Mbps	3.230Mbps	100
Hidden Service w/Orbot	1.950Mbps	0.910Mbps	100
Directly over Internet	18.58Mbps	11.95Mbps	100

Note that the throughput is not measured at the sender, because it can be unclear if all bytes have been transferred, or if bytes are still buffered locally and not yet sent.

6.5 Transmission Time

The *transmission time* is measured from the first bit of a message is sent until the last bit of the message is received. For short messages, for instance one byte, the transmission time will be short and vice versa.

Transmission time = (Message Size) / Bandwidth [56]

To estimate a transmission time, the median throughput is used in the place of the bandwidth, and the message size is set to 8 bits (1 byte).

Table 6 - Transmission Times

	Transmission Time 1 Byte (8 bits)
Swirlwave	3.200×10^{-6}s (3.200µs)
Hidden Service w/Orbot	4.103×10^{-6}s (4.103µs)
Directly over Internet	4.306×10^{-7}s (0.4306µs)

6.6 Propagation Time

Propagation time is the time required for a bit to travel from the source to the destination [56]. The basis used for calculating propagation time is a formula for *latency (delay)*:

Latency = propagation time + transmission time + queuing time + processing time [56]

For the experiments, the *processing time* and *queuing time* is considered part of what is interesting to measure, so that the queuing and processing times caused by Swirlwave are included as part of the propagation time. An estimate of the transmission time is given in section 6.5.

The two smartphones do not have synchronized clocks. Consequently, it is not sensible to measure the transmission start time at the client and the transmission end time at the server.

Round-trip times are used instead of latencies. A round-trip time is measured by how long it takes from the client sends a byte until it receives a response byte from the server. This has the advantage that start and end times can be measured at the same smartphone. For the experiments, round-trip time is defined as:

Round-trip time = 2 × latency + processing delay

The extra processing delay is due to the time that passes from the byte is read by the server until it sends a response byte to the client.

Since the processing and queuing times affect all experiments approximately equally much, since the same client and server programs are used, and are thus ignored in the calculations. Also, the transmission times are negligible compared to round-trip times. This leaves the simplified calculation:

Propagation time = Round-trip time / 2

Table 7 - Round-Trip Times

	Median	*95th Percentile*	*Num. Trials*
Swirlwave	6.370×10^{-1}s	8.160×10^{-1}s	100
Hidden Service w/Orbot	6.390×10^{-1}s	1.554s	100
Directly over Internet	1.055×10^{-1}s	1.039s	100

Table 8 - Propagation Times

	Round-Trip	*Transmission Time*	*Propagation Time*
Swirlwave	6.370×10^{-1}s	3.200×10^{-6}s	3.185×10^{-1}s
Hidden Service w/Orbot	6.390×10^{-1}s	4.103×10^{-6}s	3.195×10^{-1}s
Directly over Internet	1.055×10^{-1}s	4.306×10^{-7}s	5.275×10^{-2}s

6.7 Discussion

This section discusses the results measured as part of the experiments.

The implementation of Swirlwave will reuse an already registered hidden service when reconnecting to a previously seen network location. As seen in Table 3, there is a difference in Onion proxy start-up times between registering a new hidden service and reconnecting to one that is already registered. Registering a new hidden service took about twice as long. The start-up time varied much more when registering new hidden services than for reusing. When comparing the 90th percentiles, the start-up time for registering a new hidden service was approximately five times as slow as reusing.

When it comes to establishing connections from a client to a server, the measured times are nearly identical for connecting via Swirlwave and directly via the Internet. This is a bit surprising, since the connections in case of Swirlwave must be made through the Tor overlay network. Additionally, the Swirlwave connection times include authenticating the client. Connecting via Tor by using Orbot, which does not include any authentication mechanisms, is faster that connecting directly via a plain Internet connection. This may suggest that the difference between connecting via Tor and via the Internet roughly equals the time Swirlwave uses to authenticate the client.

A difference was observed between connecting the first time after

The throughput was lower when routing via Tor than directly over the Internet. This was true for both Swirlwave and Orbot. The throughput for Swirlwave was in this case higher than Orbot, but Internet had higher throughput than both. Transmitting directly over the Internet without Tor was 7.4 times faster than Swirlwave, and 9.5 times faster than Orbot.

Propagation times were almost similar for Swirlwave and Orbot, about three tenths of a second, but the propagation times when transmitting directly over the Internet without Tor were approximately six times less.

The most prominent downside with Swirlwave exposed by the measurements, is the lower throughput because of Tor. The extra round trips and processing involved in authenticating the client does not seem to affect the performance much. Neither does the processing done by the Swirlwave proxies.

Unfortunately, some instabilities with Swirlwave were experienced during the experiments. The transferring of data would sometimes seem to stop for unknown reasons. This is probably due to some implementation issues in the prototype.

7 Future Work

During testing of the implemented Swirlwave prototype, some instabilities were experienced. In the future, these issues will be examined and corrected, and work will be started to create a production ready version of Swirlwave that includes all aspects from the architecture chapter. As part of implementing a complete version, it is expected that improvements will be made to the architecture as well. The release version of Swirlwave will be published on Google Play, and the source code will be made openly available on the Internet.

8 Conclusion

In this thesis, a common paradigm for smartphone development has been described, where apps are viewed mainly as clients to cloud backends. Two aspects of this were problematized: Privacy for users and using smartphones as nodes in distributed systems.

A solution has been suggested by showing how a middleware can be designed to enable wide area peer-to-peer communication for smartphones—and other devices that frequently change networks and lack publicly reachable IP addresses—without the need for clouds or application servers as middlemen for storing, processing, and sharing data.

A prototype was implemented as a proof-of-concept, and experiments were conducted that explored important properties of the system.

The system was given the name Swirlwave.

Works cited

1. Myers, J. *World Economic Forum: 4 charts that explain the decline of the PC*. 2016; Available from: https://www.weforum.org/agenda/2016/04/4-charts-that-explain-the-decline-of-the-pc/.
2. Asay, M. *AWS, Google, and Microsoft cement their cloud dominance*. 2017; Available from: http://www.infoworld.com/article/3165508/cloud-computing/aws-google-and-microsoft-cement-their-cloud-dominance.html.
3. Dingledine, R., N. Mathewson, and P. Syverson, *Tor: The second-generation onion router*. 2004, DTIC Document.
4. Lynch, C. *The Washington Post: Brazil's president condemns NSA spying*. 2013; Available from: https://www.washingtonpost.com/world/national-security/brazils-president-condemns-nsa-spying/2013/09/24/fe1f78ee-2525-11e3-b75d-5b7f66349852_story.html.
5. Schneier, B., *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. 2015: W. W. Norton & Company.
6. *Gartner Says Smartphone Sales Surpassed One Billion Units in 2014*. 2015; Available from: http://www.gartner.com/newsroom/id/2996817.
7. Smith, A. *U.S. Smartphone Use in 2015*. 2015; Available from: http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/.
8. Berners-Lee, T. *Three challenges for the web, according to its inventor*. 2017; Available from: http://webfoundation.org/2017/03/web-turns-28-letter/.
9. Horton, H. *Snapchat just reserved the rights to store and use all selfies taken with the device*. The Telegraph 2015; Available from: http://www.telegraph.co.uk/technology/news/11966036/Snapchat-just-reserved-the-rights-to-store-and-use-all-selfies-taken-with-the-device.html.
10. D'Onfro, J. *Snapchat now has nearly 100 million daily users*. Business Insider UK 2015; Available from: http://uk.businessinsider.com/snapchat-daily-active-users-2015-5?r=US&IR=T.
11. Vincent, J. *'The Snappening': Has Snapchat been hacked? What is Snapsaved? Your questions answered*. Independent 2014; Available from: http://www.independent.co.uk/life-style/gadgets-and-tech/the-snappening-has-snapchat-been-hacked-whats-snapsaved-your-questions-answered-9790658.html.
12. Johnson, A. *Almost 600 Accounts Breached in 'Celebgate' Nude Photo Hack, FBI Says*. NBC News 2015; Available from: http://www.nbcnews.com/pop-culture/pop-culture-news/almost-600-accounts-breached-celebgate-nude-photo-hack-fbi-says-n372641.
13. Johannessen, N., et al. *VG avslører: Politikere og toppbyråkrater rammet av hackerangrep*. 2017; Available from: http://www.vg.no/nyheter/innenriks/erna-solberg/vg-avsloerer-politikere-og-toppbyraakrater-rammet-av-hackerangrep/a/23920237/.
14. Fox-Brewster, T. *Google Just Discovered A Massive Web Leak... And You Might Want To Change All Your Passwords*. 2017; Available from: https://www.forbes.com/sites/thomasbrewster/2017/02/24/google-just-discovered-a-massive-web-leak-and-you-might-want-to-change-all-your-passwords/.
15. Fox-Brewster, T. *How Bad Was CloudBleed? 1.2 Million Leaks Bad*. 2017; Available from: https://www.forbes.com/sites/thomasbrewster/2017/03/01/cloudbleed-leak-massive-but-not-too-harmful/.
16. Authority, N.D.P., *The Great Data Race*. 2015.
17. Authority, N.D.P., *Personal data in exchange for free services: an unhappy partnership?* 2016.
18. *Microsoft Azure: What is cloud computing?* ; Available from: https://azure.microsoft.com/nb-no/overview/what-is-cloud-computing/.
19. Mell, P. and T. Grance, *The NIST definition of cloud computing*. 2011.
20. Fox, A., et al., *Above the clouds: A Berkeley view of cloud computing*. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 2009. **28**(13): p. 2009.

21. Metz, C. *Wired: How Facebook Moved 20 Billion Instagram Photos Without You Noticing*. 2014; Available from: https://www.wired.com/2014/06/facebook-instagram/.
22. Novet, J., *Venture Beat: Snapchat by the numbers: 161 million daily users in Q4 2016, users visit 18 times a day*. 2017.
23. Barrow, B. *Fortune: Snapchat Has Signed Another Mega Cloud Purchase, and It's Not From Google*. 2017; Available from: http://fortune.com/2017/02/09/snap-inc-signs-big-aws-deal/.
24. Weinberger, M. *Business Insider Nordic: Amazon, Google, and Microsoft might be going to war to win Uber's cloud business*. 2016; Available from: http://nordic.businessinsider.com/uber-outsourcing-infrastructure-to-public-cloud-2016-5.
25. Larus, J.R., *The cloud will change everything.* ACM SIGPLAN Notices, 2011. **46**(3): p. 1-2.
26. *Google Cloud Platform: The Mobile Cloud Era*. Available from: https://cloud.google.com/solutions/mobile/.
27. *Google Cloud Platform: Mobile App Backend Services*. Available from: https://cloud.google.com/solutions/mobile/mobile-app-backend-services.
28. Aiken, B., et al., *Rfc 2768: Network policy and services: A report of a workshop on middleware.* Internet Engineering Task Force, 2000: p. 1.
29. Tanenbaum, A.S. and M.v. Steen, *Distributed Systems: Pearson New International Edition: Principles and Paradigms*. 2/E ed. 2014: Pearson.
30. Rogers, M. and S. Bhatti, *How to disappear completely: A survey of private peer-to-peer networks.* networks, 2007. **13**: p. 14.
31. Bricklin, D. *Friend-to-Friend Networks*. 2000; Available from: http://www.bricklin.com/f2f.htm.
32. Popescu, B.C., B. Crispo, and A.S. Tanenbaum, *Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System*, in *Security Protocols: 12th International Workshop, Cambridge, UK, April 26-28, 2004. Revised Selected Papers*, B. Christianson, et al., Editors. 2006, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 213-220.
33. Li, S., L. Da Xu, and S. Zhao, *The internet of things: a survey.* Information Systems Frontiers, 2015. **17**(2): p. 243-259.
34. Postel, J., *RFC 791: Internet protocol.* 1981.
35. Deering, S. and R. Hinden, *RFC 2460: Internet Protocol*. 1998, Version.
36. Perkins, C., D. Johnson, and J. Arkko, *Mobility support in IPv6*. 2011.
37. Comer, D., *Internetworking with TCP/IP*. Sixth edition, Pearson new international edition. ed. Pearson custom library. 2014, Harlow: Pearson. volumes.
38. Hu, Z. *NAT traversal techniques and peer-to-peer applications*. in *HUT T-110.551 Seminar on Internetworking*. 2005. Citeseer.
39. Ford, B., P. Srisuresh, and D. Kegel. *Peer-to-Peer Communication Across Network Address Translators*. in *USENIX Annual Technical Conference, General Track*. 2005.
40. Lee, Y.-D., *SOCKS: A protocol for TCP proxy across firewalls.* NEC Systems, 2005: p. 1-3.
41. Bahl, P., et al. *Advancing the state of mobile cloud computing*. in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. 2012. ACM.
42. Goodrich, M.T. and R. Tamassia, *Introduction to computer security*. First edition, Pearson new international edition.

Pearson new international edition. ed. Pearson custom library. 2014, Harlow, Essex, England: Pearson. ii, 514 pages.
43. Butterfield, A. and G.E. Ngondi, *A dictionary of computer science*. 2016, Oxford University Press,: Oxford. p. 1 online resource.
44. Daemen, J. and V. Rijmen, *AES proposal: Rijndael.* 1999.
45. ETSI, M., *Mobile-edge computing.* Introductory Technical White Paper, 2014.
46. Smith, C. *DMR: 108 Amazing Android Statistics (November 2016)*. 2016; Available from: http://expandedramblings.com/index.php/android-statistics/.
47. Callaham, J. *Andoid Central: Google says there are now 1.4 billion active Android devices worldwide*. 2015; Available from: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.
48. Meier, R., *Professional Android 4 Application Development*. 2012: Wrox. 864.

49. Lee, Y.D., *SOCKS 4A: A Simple Extension to SOCKS 4 Protocol.* dostopno na: www. openssh. org/txt/socks4a. protocol, 2012.
50. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, 1990: p. 1-84.
51. Bass, L., P. Clements, and R. Kazman, *Software architecture in practice*. 2nd ed. 2003, Boston, Mass. ; London: Addison-Wesley. xxii, 528 p.
52. Bradbury, D., *Unveiling the dark web.* Network Security, 2014. **2014**(4): p. 14-17.
53. Leach, P.J., M. Mealling, and R. Salz, *A universally unique identifier (uuid) urn namespace.* 2005.
54. Lamport, L., *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, 1978. **21**(7): p. 558-565.
55. Travis, G. *Getting started with new I/O (NIO)*. 2003; Available from: https://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html.
56. Forouzan, B.A., *Data communications and networking*. 5th ed. McGraw-Hill Forouzan networking series. 2013, New York, N.Y.: McGraw Hill. xxxviii, 1226 p.

Appendix A: Main Activity

```
package com.swirlwave.android;

import android.content.Intent;
import android.nfc.NdefMessage;
import android.nfc.NdefRecord;
import android.nfc.NfcAdapter;
import android.nfc.NfcEvent;
import android.os.Bundle;
import android.os.Parcelable;
import android.support.v4.app.ActivityCompat;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.CompoundButton;
import android.widget.Switch;
import android.widget.Toast;

import com.google.gson.Gson;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;
import com.swirlwave.android.peers.PeersFragment;
import com.swirlwave.android.permissions.AppPermissions;
import com.swirlwave.android.permissions.AppPermissionsResult;
import com.swirlwave.android.service.ActionNames;
import com.swirlwave.android.service.SwirlwaveService;
import com.swirlwave.android.settings.LocalSettings;
import com.swirlwave.android.tor.SwirlwaveOnionProxyManager;

import java.util.Date;

import static android.nfc.NdefRecord.createMime;

public class MainActivity extends AppCompatActivity implements CompoundButton.OnCheckedChangeListener,
ActivityCompat.OnRequestPermissionsResultCallback, NfcAdapter.CreateNdefMessageCallback {
    private AppPermissions mAppPermissions = new AppPermissions(this);
    private NfcAdapter mNfcAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Check if all app permissions are granted.
        // Note: If the result isn't success, then the user will be asked and the callback method
        // onRequestPermissionsResult will be called async.
        AppPermissionsResult permissionsResult = mAppPermissions.requestAllPermissions();
        if (permissionsResult == AppPermissionsResult.Success) {
            init(true);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        AppPermissionsResult appPermissionsResult =
                mAppPermissions.onRequestPermissionsResult(requestCode, permissions, grantResults);

        boolean wasSuccess = false;
        if (appPermissionsResult == AppPermissionsResult.Success) {
            wasSuccess = true;
        } else if(appPermissionsResult == AppPermissionsResult.Refused) {
            Log.e(getString(R.string.app_name), "The user has refused to grant needed permissions");
            Toast.makeText(this, R.string.required_permissions_refused, Toast.LENGTH_LONG).show();
        } else if (appPermissionsResult == AppPermissionsResult.Interrupted) {
            Log.e(getString(R.string.app_name), "Permission granting interaction was interrupted");
            Toast.makeText(this, R.string.permission_granting_interrupted, Toast.LENGTH_LONG).show();
        }

        init(wasSuccess);
    }

    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        Intent intent = new Intent(this, SwirlwaveService.class);
        String actionName = isChecked ? ActionNames.ACTION_INIT_SERVICE :
ActionNames.ACTION_SHUT_DOWN_SERVICE;
        intent.setAction(actionName);
        startService(intent);
    }

    @Override
    protected void onResume() {
        super.onResume();

        // Check to see that the Activity started due to an Android Beam
        if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
            Parcelable[] rawMsgs = getIntent().getParcelableArrayExtra(
```

```
                NfcAdapter.EXTRA_NDEF_MESSAGES);
            // only one message sent during the beam
            NdefMessage msg = (NdefMessage) rawMsgs[0];
            // record 0 contains the MIME type, record 1 is the AAR, if present
            String peerJson = new String(msg.getRecords()[0].getPayload());

            if (peerJson.equals("")) {
                Log.e(getString(R.string.app_name), "Transmitted peer info was empty");
                showToastOnUiThread(getString(R.string.nfc_empty_peer_info));
            } else {
                Peer peer = new Gson().fromJson(peerJson, Peer.class);

                Peer alreadyExistingPeer = PeersDb.selectByUuid(this, peer.getPeerId());
                if (alreadyExistingPeer == null) {
                    PeersDb.insert(this, peer);
                } else {
                    peer.setId(alreadyExistingPeer.getId());
                    PeersDb.update(this, peer);
                }

                refreshPeersFragment();
            }
        }
    }

    private void refreshPeersFragment() {
        PeersFragment peersFragment = (PeersFragment)
getFragmentManager().findFragmentById(R.id.peersFragment);
        peersFragment.refresh();
    }

    @Override
    protected void onNewIntent(Intent intent) {
        setIntent(intent);
    }

    private void init(boolean allPermissionsGranted) {
        Switch serviceSwitch = (Switch) findViewById(R.id.serviceSwitch);
        serviceSwitch.setEnabled(allPermissionsGranted);
        serviceSwitch.setChecked(SwirlwaveService.isRunning());
        serviceSwitch.setOnCheckedChangeListener(this);

        // Due to a bug in Android, this doesn't always work, and the user has to whitelist manually
        mAppPermissions.requestIgnoreBatteryOptimization(this);

        LocalSettings.ensureInstallationNameAndPhoneNumber(this);

        // Check for available NFC Adapter
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        if (mNfcAdapter == null) {
            Toast.makeText(this, R.string.nfc_not_available, Toast.LENGTH_LONG).show();
            finish();
            return;
        }

        // Register NFC callback
        mNfcAdapter.setNdefPushMessageCallback(this, this);
    }

    @Override
    public NdefMessage createNdefMessage(NfcEvent event) {
        String peerJson;
        String address = SwirlwaveOnionProxyManager.getAddress();

        if (address.equals("")) {
            showToastOnUiThread(getString(R.string.must_be_conntected_to_transfer_contact_info));
            peerJson = null;
        } else {
            try {
                LocalSettings localSettings = new LocalSettings(this);
                Peer localPeer = new Peer(
                        localSettings.getInstallationName(),
                        localSettings.getUuid(),
                        localSettings.getAsymmetricKeys().first,
                        localSettings.getAddress(),
                        Integer.parseInt(localSettings.getAddressVersion()),
                        true,
                        new Date(),
                        localSettings.getPhoneNumber(),
                        PeersDb.selectAllFriendUuids(this),
                        localSettings.getCapabilities(),
                        false
                        );
                peerJson = new Gson().toJson(localPeer);
            } catch (Exception e) {
                showToastOnUiThread(getString(R.string.something_went_wrong) + ": " +
e.getLocalizedMessage());
                peerJson = null;
```

```
            }
        }

        NdefMessage msg = null;

        if (peerJson != null) {
            msg = new NdefMessage(new NdefRecord[] {
                            createMime("application/vnd.com.swirlwave.android.beam", peerJson.getBytes()),
                            NdefRecord.createApplicationRecord("com.swirlwave.android") });
        }

        return msg;
    }


    private void showToastOnUiThread(String text) {
        runOnUiThread(new ShowToastRunnable(text));
    }

    private class ShowToastRunnable implements Runnable {
        private String mText;

        public ShowToastRunnable(String text) {
            mText = text;
        }

        @Override
        public void run() {
            Toast.makeText(getApplicationContext(), mText, Toast.LENGTH_LONG).show();
        }
    }
}
```

Appendix B: Server Proxy

```
package com.swirlwave.android.proxies.serverside;

import android.content.Context;
import android.support.annotation.NonNull;
import android.util.Log;

import com.swirlwave.android.R;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;
import com.swirlwave.android.proxies.ConnectionMessage;
import com.swirlwave.android.proxies.FriendOnlineStatusUpdater;
import com.swirlwave.android.proxies.SelectionKeyAttachment;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;
import java.util.Iterator;
import java.util.Random;
import java.util.Set;
import java.util.UUID;

public class ServerSideProxy implements Runnable {
    public static final int PORT = 9345;
    public static final int LOCAL_SERVER_PORT = 8088;
    private static ServerSocketChannel sServerSocketChannel;
    private static final Random mRnd = new Random();
    private Context mContext;
    private volatile boolean mRunning = true;
    private final ByteBuffer mBuffer = ByteBuffer.allocate(16384);
    private static Selector sSelector;

    public ServerSideProxy(Context context) {
        mContext = context;
    }

    @Override
    public void run() {
        try {
            Selector selector = getServerSocketChannel();

            while (mRunning) {
                int numChannelsReady = selector.select();

                if (numChannelsReady == 0) {
                    continue;
                }

                Set<SelectionKey> keys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = keys.iterator();

                while (iterator.hasNext()) {
                    SelectionKey selectionKey = iterator.next();

                    try {
                        if (selectionKey.isAcceptable()) {
                            SocketChannel clientSocketChannel = acceptIncomingSocket(selectionKey);
                            ConnectionMessageSelectionKeyAttachment connectionMessageSelectionKeyAttachment
= new ConnectionMessageSelectionKeyAttachment();

                            boolean ok = sendRandomNumber(clientSocketChannel,
connectionMessageSelectionKeyAttachment);

                            if (ok) {
                                clientSocketChannel.register(selector, SelectionKey.OP_READ,
connectionMessageSelectionKeyAttachment);
                            } else {
                                closeChannel(clientSocketChannel);
                            }
                        } else if (selectionKey.isConnectable()) {
                            SocketChannel localServerChannel = (SocketChannel) selectionKey.channel();
                            if (localServerChannel.finishConnect()) {
                                SelectionKey localServerSelectionKey = selectionKey;
                                localServerSelectionKey.interestOps(SelectionKey.OP_READ);

                                SocketChannel incomingClientChannel = (SocketChannel)
```

```
selectionKey.attachment();
                                SelectionKey incomingClientSelectionKey =
incomingClientChannel.register(selector, SelectionKey.OP_READ);

                                SelectionKeyAttachment localServerSelectionKeyAttachment = new
SelectionKeyAttachment(incomingClientChannel, incomingClientSelectionKey, true);
                                localServerSelectionKey.attach(localServerSelectionKeyAttachment);

                                SelectionKeyAttachment incomingClientSelectionKeyAttachment = new
SelectionKeyAttachment(localServerChannel, localServerSelectionKey, false);
                                incomingClientSelectionKey.attach(incomingClientSelectionKeyAttachment);
                            }
                        } else if (selectionKey.isReadable()) {
                            SocketChannel inChannel = null;
                            SocketChannel outChannel = null;

                            try {
                                inChannel = (SocketChannel) selectionKey.channel();

                                Object attachmentObject = selectionKey.attachment();

                                if (attachmentObject instanceof SelectionKeyAttachment) {
                                    SelectionKeyAttachment attachment = (SelectionKeyAttachment)
attachmentObject;
                                    outChannel = attachment.getSocketChannel();

                                    boolean ok = processInput(inChannel, outChannel, attachment);

                                    if (!ok) {
                                        closeSocketPairs(selectionKey);
                                    }
                                } else if (attachmentObject instanceof
ConnectionMessageSelectionKeyAttachment) {
                                    ConnectionMessageSelectionKeyAttachment
connectionMessageSelectionKeyAttachment = (ConnectionMessageSelectionKeyAttachment) attachmentObject;

                                    boolean ok = processSystemMessage(inChannel,
connectionMessageSelectionKeyAttachment);

                                    if (ok) {
                                        if (connectionMessageSelectionKeyAttachment.isCompleted()) {
                                            // Deregister reading (OP_READ) from client socket until local
server is connected
                                            selectionKey.attach(null);
                                            selectionKey.cancel();

                                            ConnectionMessage message =
sendSystemMessageResponse(inChannel, connectionMessageSelectionKeyAttachment);

                                            if (message != null) {
                                                switch (message.getMessageType()) {
                                                    case APPLICATION_LAYER_CONNECTION: {
                                                        SocketChannel localServerChannel =
connectLocalServer(selector);
                                                        localServerChannel.register(selector,
SelectionKey.OP_CONNECT, inChannel);
                                                        new Thread(new FriendOnlineStatusUpdater(mContext,
message.getSenderId(), true)).start();
                                                    }
                                                    break;

                                                    case ADDRESS_ANNOUNCEMENT: {
                                                        closeChannel(inChannel);
                                                        String address = new
String(message.getSystemMessage(), StandardCharsets.UTF_8);
                                                        new Thread(new FriendAddressUpdater(mContext,
message.getSenderId(), address)).start();
                                                    }
                                                    break;

                                                    default: {
                                                        closeChannel(inChannel);
                                                    }
                                                    break;
                                                }
                                            } else {
                                                closeChannel(inChannel);
                                            }
                                        }
                                    } else {
                                        closeSocketPairs(selectionKey);
                                    }
                                }
                            } catch (IOException ie) {
                                closeSocketPairs(selectionKey);
                            }
                        }
                    } catch (Exception e) {
```

```
                    Log.e(mContext.getString(R.string.service_name), e.toString());
                }

                iterator.remove();
            }

            keys.clear();
        }
    } catch (Exception e) {
        Log.e(mContext.getString(R.string.service_name), e.toString());
    }
}

@NonNull
private synchronized Selector getServerSocketChannel() throws IOException {
    if (sServerSocketChannel == null || !sServerSocketChannel.isOpen() ||
sServerSocketChannel.socket().isClosed() || !sServerSocketChannel.socket().isBound()) {
        sSelector = Selector.open();
        sServerSocketChannel = ServerSocketChannel.open();
        sServerSocketChannel.configureBlocking(false);
        ServerSocket serverSocket = sServerSocketChannel.socket();
        InetSocketAddress inetSocketAddress = new InetSocketAddress(PORT);
        serverSocket.bind(inetSocketAddress);
        sServerSocketChannel.register(sSelector, SelectionKey.OP_ACCEPT);
    }

    return sSelector;
}

private boolean sendRandomNumber(SocketChannel clientSocketChannel,
ConnectionMessageSelectionKeyAttachment connectionMessageSelectionKeyAttachment) {
    byte[] randomBytes = new byte[4];
    mRnd.nextBytes(randomBytes);
    ByteBuffer buffer = ByteBuffer.wrap(randomBytes);

    try {

connectionMessageSelectionKeyAttachment.setSentRandomNumber(ConnectionMessage.bytesToInt(randomBytes));

        while (buffer.hasRemaining()) {
            clientSocketChannel.write(buffer);
        }
    } catch (Exception e) {
        return false;
    }

    return true;
}

private boolean processSystemMessage(SocketChannel inChannel, ConnectionMessageSelectionKeyAttachment
systemMessageAttachment) throws IOException {
    boolean isOk;

    mBuffer.clear();
    int bytesRead = inChannel.read(mBuffer);
    mBuffer.flip();

    // A value of -1 means that the socket has been closed by the peer.
    if (bytesRead == -1) {
        isOk = false;
    } else if (mBuffer.limit() > 0) {
        isOk = true;

        int systemMessageBytesRead = systemMessageAttachment.getBytesRead();

        while (mBuffer.hasRemaining()) {
            byte nextByte = mBuffer.get();

            if (systemMessageAttachment.notCompleted()) {
                systemMessageBytesRead++;
                systemMessageAttachment.getByteArrayStream().write(nextByte);

                if (systemMessageBytesRead == 4) {
                    byte[] longBytes = systemMessageAttachment.getByteArrayStream().toByteArray();

                    ByteArrayInputStream byteInputStream = new ByteArrayInputStream(longBytes);
                    DataInputStream dataInputStream = new DataInputStream(byteInputStream);
                    int messageLength = dataInputStream.readInt();
                    dataInputStream.close();

                    systemMessageAttachment.setMessageLength(messageLength);
                    systemMessageAttachment.setByteArrayStream(new ByteArrayOutputStream());
                } else if (systemMessageBytesRead > 4 && systemMessageBytesRead - 4 ==
systemMessageAttachment.getMessageLength()) {
                    systemMessageAttachment.setCompletedStatus(true);
                }

                systemMessageAttachment.setBytesRead(systemMessageBytesRead);
```

```
                }
            }
        } else {
            isOk = true;
        }

        return isOk;
    }

    private ConnectionMessage sendSystemMessageResponse(SocketChannel clientSocketChannel,
ConnectionMessageSelectionKeyAttachment connectionMessageSelectionKeyAttachment) {
        ConnectionMessage message;
        byte responseCode = (byte)0x0b;

        try {
            byte[] bytes = connectionMessageSelectionKeyAttachment.getByteArrayStream().toByteArray();
            UUID senderId = ConnectionMessage.extractSenderId(bytes);
            Peer sender = PeersDb.selectByUuid(mContext, senderId);

            message = ConnectionMessage.fromByteArray(bytes, sender.getPublicKey());

            if (message.getRandomNumber() == connectionMessageSelectionKeyAttachment.getSentRandomNumber())
{
                responseCode = (byte)0x0a;
            } else {
                message = null;
            }
        } catch (Exception e) {
            return null;
        }

        try {
            ByteBuffer buffer = ByteBuffer.wrap(new byte[] {(byte)responseCode});
            while(buffer.hasRemaining()) {
                clientSocketChannel.write(buffer);
            }
        } catch (IOException ie) {
            return null;
        }

        return message;
    }

    private SocketChannel acceptIncomingSocket(SelectionKey selectionKey) throws IOException {
        ServerSocketChannel serverSocketChannel = (ServerSocketChannel) selectionKey.channel();
        Socket socket = serverSocketChannel.socket().accept();
        SocketChannel socketChannel = socket.getChannel();
        socketChannel.configureBlocking(false);
        return socketChannel;
    }

    private SocketChannel connectLocalServer(Selector selector) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_CONNECT);
        InetSocketAddress localServerAddress = new InetSocketAddress("127.0.0.1", LOCAL_SERVER_PORT);
        socketChannel.connect(localServerAddress);
        return socketChannel;
    }

    private void closeSocketPairs(SelectionKey selectionKey) {
        SocketChannel inChannel = (SocketChannel) selectionKey.channel();

        Object attachmentObject = selectionKey.attachment();
        if (attachmentObject instanceof  SelectionKeyAttachment) {
            SelectionKeyAttachment selectionKeyAttachment = (SelectionKeyAttachment) attachmentObject;
            SocketChannel outChannel = selectionKeyAttachment.getSocketChannel();
            selectionKeyAttachment.getSelectionKey().attach(null);
            selectionKeyAttachment.getSelectionKey().cancel();
            closeChannel(outChannel);
        }

        selectionKey.attach(null);
        selectionKey.cancel();
        closeChannel(inChannel);
    }

    private void closeChannel(SocketChannel socketChannel) {
        Socket socket = socketChannel.socket();
        try {
            // Closing the socket, even if channel will be closed.
            // Reason: To close a little bit sooner. The cancel on the channel will not close the socket
until the next select.
            socket.close();
        } catch (IOException ie) {
            Log.e(mContext.getString(R.string.service_name), "Error closing socket" + socket + ": " + ie);
        }
```

```
        try {
            socketChannel.close();
        } catch (IOException ie) {
            Log.e(mContext.getString(R.string.service_name), "Error closing channel: " + ie);
        }
    }

    private boolean processInput(SocketChannel inChannel, SocketChannel outChannel, SelectionKeyAttachment
attachment) throws IOException {
        boolean isOk;

        mBuffer.clear();
        int bytesRead = inChannel.read(mBuffer);
        mBuffer.flip();

        // A value of -1 means that the socket has been closed by the peer.
        if (bytesRead == -1) {
            isOk = false;
        } else if (mBuffer.limit() > 0) {
            if (attachment.isClientChannel()) {
                isOk = processDataFromLocalServer(mBuffer, outChannel, attachment);
            } else {
                isOk = processDataFromClient(mBuffer, outChannel, attachment);
            }
        } else {
            isOk = true;
        }

        return isOk;
    }

    private boolean processDataFromClient(ByteBuffer inBuffer, SocketChannel outChannel,
SelectionKeyAttachment attachment) throws IOException {
        while(inBuffer.hasRemaining()) {
            outChannel.write(inBuffer);
        }

        return true;
    }

    private boolean processDataFromLocalServer(ByteBuffer inBuffer, SocketChannel outChannel,
SelectionKeyAttachment attachment) throws IOException {
        while(inBuffer.hasRemaining()) {
            outChannel.write(inBuffer);
        }

        return true;
    }

    public void terminate() {
        mRunning = false;
    }
}
```

Appendix C: Client Proxy

```
package com.swirlwave.android.proxies.clientside;

import android.content.Context;
import android.util.Log;
import android.util.Pair;

import com.swirlwave.android.R;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;
import com.swirlwave.android.proxies.ConnectionMessage;
import com.swirlwave.android.proxies.FriendOnlineStatusUpdater;
import com.swirlwave.android.proxies.MessageType;
import com.swirlwave.android.proxies.SelectionKeyAttachment;
import com.swirlwave.android.settings.LocalSettings;
import com.swirlwave.android.sms.SmsSender;
import com.swirlwave.android.tor.SwirlwaveOnionProxyManager;

import java.io.IOException;
import java.net.BindException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.UUID;

public class ClientSideProxy implements Runnable {
    public static final int START_PORT = 9346;
    private Context mContext;
    private LocalSettings mLocalSettings;
    private static List<ServerSocketChannel> sServerSocketChannels = new ArrayList<>();
    private volatile boolean mRunning = true;
    private final ByteBuffer mBuffer = ByteBuffer.allocate(16384);
    private String mPublicKeyString, mPrivateKeyString;
    private static Selector sSelector;

    public ClientSideProxy(Context context) throws Exception {
        mContext = context;
        mLocalSettings = new LocalSettings(mContext);
        Pair<String, String> keys = mLocalSettings.getAsymmetricKeys();
        mPublicKeyString = keys.first;
        mPrivateKeyString = keys.second;
    }

    @Override
    public void run() {
        try {
            Selector selector = bindPorts();

            while (mRunning) {
                int numChannelsReady = selector.select();

                if (numChannelsReady == 0) {
                    continue;
                }

                Set<SelectionKey> keys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = keys.iterator();

                while (iterator.hasNext()) {
                    SelectionKey selectionKey = iterator.next();

                    try {
                        if (selectionKey.isAcceptable()) {
                            ServerSocketChannel serverSocketChannel = (ServerSocketChannel)
selectionKey.channel();
                            int incomingPort =  serverSocketChannel.socket().getLocalPort();

                            SocketChannel clientSocketChannel = acceptIncomingSocket(selectionKey);
                            SocketChannel onionProxyChannel = connectOnionProxy(selector);

                            Pair<Integer, SocketChannel> attachment = new Pair<>(incomingPort,
clientSocketChannel);

                            onionProxyChannel.register(selector, SelectionKey.OP_CONNECT, attachment);
                        } else if (selectionKey.isConnectable()) {
                            SocketChannel onionProxyChannel = (SocketChannel) selectionKey.channel();
                            if (onionProxyChannel.finishConnect()) {
```

```
                                SelectionKey onionProxySelectionKey = selectionKey;
                                onionProxySelectionKey.interestOps(SelectionKey.OP_READ);

                                Pair<Integer, SocketChannel> attachment = (Pair<Integer, SocketChannel>)
onionProxySelectionKey.attachment();
                                onionProxySelectionKey.attach(null);
                                int clientPort = attachment.first;
                                SocketChannel incomingClientChannel = attachment.second;

                                // TODO: Implement resolving of friends more properly
                                Peer friend = resolveFriend(clientPort);

                                if (friend.getOnlineStatus()) {
                                    String onionAddress = friend.getAddress();
                                    if (onionAddress != null) {
                                        // TODO: Implement with real value
                                        UUID destination = resolveDestination(clientPort);

                                        onionProxyChannel.socket().setSoTimeout(10000);
                                        boolean ok = performSocks4aConnectionRequest(onionProxyChannel,
onionAddress);

                                        if (ok) {
                                            OnionProxySelectionKeyAttachment
onionProxySelectionKeyAttachment = new OnionProxySelectionKeyAttachment(incomingClientChannel);

onionProxySelectionKeyAttachment.setMode(ClientProxyMode.AWAITING_ONIONPROXY_RESULT);
                                            onionProxySelectionKeyAttachment.setDestination(destination);
                                            onionProxySelectionKeyAttachment.setFriend(friend);

onionProxySelectionKey.attach(onionProxySelectionKeyAttachment);
                                        } else {
                                            incomingClientChannel.close();
                                        }
                                    } else {
                                        incomingClientChannel.close();
                                    }
                                } else {
                                    incomingClientChannel.close();
                                }
                            }
                        } else if (selectionKey.isReadable()) {
                            try {
                                boolean ok = false;

                                SocketChannel inChannel = (SocketChannel) selectionKey.channel();
                                SelectionKeyAttachment attachment = (SelectionKeyAttachment)
selectionKey.attachment();

                                if (attachment == null) {
                                } else if (attachment.acceptingPayload()) {
                                    SocketChannel outChannel = attachment.getSocketChannel();
                                    ok = processInput(inChannel, outChannel, attachment);
                                } else if (attachment instanceof OnionProxySelectionKeyAttachment) {
                                    OnionProxySelectionKeyAttachment onionProxySelectionKeyAttachment =
(OnionProxySelectionKeyAttachment) attachment;
                                    ClientProxyMode mode = onionProxySelectionKeyAttachment.getMode();
                                    if (mode == ClientProxyMode.AWAITING_ONIONPROXY_RESULT) {
                                        ok = readSocks4aConnectionResponse(inChannel,
onionProxySelectionKeyAttachment);
                                    } else if (mode == ClientProxyMode.AWAITING_SERVERPROXY_RANDOM_NUMBER)
{
                                        ok = readRandomNumber(inChannel, onionProxySelectionKeyAttachment);
                                    } else if (mode ==
ClientProxyMode.AWAITING_SERVERPROXY_AUTHENTICATION_RESULT) {
                                        ok = readServerProxyAuthenticationResult(inChannel,
onionProxySelectionKeyAttachment);

                                        // Start to read payload from client
                                        if (onionProxySelectionKeyAttachment.getMode() ==
ClientProxyMode.ACCEPTING_PAYLOAD) {
                                            SocketChannel clientChannel =
onionProxySelectionKeyAttachment.getSocketChannel();
                                            SelectionKey clientSelectionKey =
clientChannel.register(selector, SelectionKey.OP_READ);
                                            SelectionKeyAttachment clientSelectionKeyAttachment = new
SelectionKeyAttachment(inChannel, selectionKey, false);
                                            clientSelectionKey.attach(clientSelectionKeyAttachment);
                                        }
                                    }
                                }

                                if (!ok) {
                                    closeSocketPairs(selectionKey);
                                }
                            } catch (IOException ie) {
                                closeSocketPairs(selectionKey);
                            }
```

```
                    }
                } catch (Exception e) {
                    Log.e(mContext.getString(R.string.service_name), e.toString());
                }

                iterator.remove();
            }

            keys.clear();
        }
    } catch (Exception e) {
        Log.e(mContext.getString(R.string.service_name), e.toString());
    }
}

private boolean performSocks4aConnectionRequest(SocketChannel onionProxyChannel, String onionAddress) {
    short remoteHiddenServicePort = (short)SwirlwaveOnionProxyManager.HIDDEN_SERVICE_PORT;

    mBuffer.clear();
    mBuffer.put((byte)0x04);
    mBuffer.put((byte)0x01);
    mBuffer.putShort(remoteHiddenServicePort);
    mBuffer.putInt(0x01);
    mBuffer.put((byte)0x00);
    mBuffer.put(onionAddress.getBytes());
    mBuffer.put((byte)0x00);
    mBuffer.flip();

    try {
        while (mBuffer.hasRemaining()) {
            onionProxyChannel.write(mBuffer);
        }
    } catch (IOException ie) {
        return false;
    }

    return true;
}

private boolean readSocks4aConnectionResponse(SocketChannel onionProxyChannel,
OnionProxySelectionKeyAttachment onionProxySelectionKeyAttachment) throws IOException {
    boolean isOk;

    mBuffer.clear();
    int bytesRead = onionProxyChannel.read(mBuffer);
    mBuffer.flip();

    // A value of -1 means that the socket has been closed by the peer.
    if (bytesRead == -1) {
        isOk = false;
    } else if (mBuffer.limit() > 0) {
        int alreadyReceived = onionProxySelectionKeyAttachment.getOnionProxyResultBytesReceived();
        byte[] byteArray = onionProxySelectionKeyAttachment.getOnionProxyResult();

        isOk = true;
        while (mBuffer.hasRemaining()) {
            byte nextByte = mBuffer.get();

            if (alreadyReceived < byteArray.length) {
                byteArray[alreadyReceived] = nextByte;
                alreadyReceived++;
                onionProxySelectionKeyAttachment.setOnionProxyResultBytesReceived(alreadyReceived);
            }
        }

        if (isOk && alreadyReceived == byteArray.length) {
            Peer friend = onionProxySelectionKeyAttachment.getFriend();
            byte[] resultBytes = onionProxySelectionKeyAttachment.getOnionProxyResult();
            if (resultBytes[0] == (byte)0x00 && resultBytes[1] == (byte)0x5a) {

onionProxySelectionKeyAttachment.setMode(ClientProxyMode.AWAITING_SERVERPROXY_RANDOM_NUMBER);
                if (!friend.getOnlineStatus()) {
                    new Thread(new FriendOnlineStatusUpdater(mContext, friend.getPeerId(),
true)).start();
                }
            } else {
                onionProxySelectionKeyAttachment.setMode(ClientProxyMode.INVALID_ONIONPROXY_RESULT);

                // Refresh friend and check online status in case someone already changed it
                friend = PeersDb.selectByUuid(mContext, friend.getPeerId());

                if (friend.getOnlineStatus()) {
                    new Thread(new FriendOnlineStatusUpdater(mContext, friend.getPeerId(),
false)).start();
                    new Thread(new SmsSender(mContext, friend.getSecondaryChannelAddress(),
SwirlwaveOnionProxyManager.getAddress())).start();
                }
```

```
                    isOk = false;
                }
            }
        } else {
            isOk = true;
        }

        return isOk;
    }

    private boolean readRandomNumber(SocketChannel onionProxyChannel, OnionProxySelectionKeyAttachment
onionProxySelectionKeyAttachment) throws IOException {
        boolean isOk;

        mBuffer.clear();
        int bytesRead = onionProxyChannel.read(mBuffer);
        mBuffer.flip();

        // A value of -1 means that the socket has been closed by the peer.
        if (bytesRead == -1) {
            isOk = false;
        } else if (mBuffer.limit() > 0) {
            int alreadyReceived = onionProxySelectionKeyAttachment.getServerProxyRandomBytesReceived();
            byte[] byteArray = onionProxySelectionKeyAttachment.getServerProxyRandomBytes();

            isOk = true;
            while (mBuffer.hasRemaining()) {
                byte nextByte = mBuffer.get();

                if (alreadyReceived < byteArray.length) {
                    byteArray[alreadyReceived] = nextByte;
                    alreadyReceived++;
                    onionProxySelectionKeyAttachment.setServerProxyRandomBytesReceived(alreadyReceived);
                }
            }

            if (isOk && alreadyReceived == byteArray.length) {
                isOk = performServerProxyAuthenticationRequest(onionProxyChannel,
onionProxySelectionKeyAttachment);
            }
        } else {
            isOk = true;
        }

        return isOk;
    }

    private boolean performServerProxyAuthenticationRequest(SocketChannel onionProxyChannel,
OnionProxySelectionKeyAttachment onionProxySelectionKeyAttachment) {
        byte[] randomBytesFromServer = onionProxySelectionKeyAttachment.getServerProxyRandomBytes();
        UUID destination = onionProxySelectionKeyAttachment.getDestination();

        try {
            byte[] bytes = generateConnectionMessage(randomBytesFromServer, destination);

            if (bytes == null) {
                return false;
            }

            mBuffer.clear();
            mBuffer.putInt(bytes.length);
            mBuffer.put(bytes);
            mBuffer.flip();

            try {
                while (mBuffer.hasRemaining()) {
                    onionProxyChannel.write(mBuffer);
                }
            } catch (IOException ie) {
                return false;
            }


onionProxySelectionKeyAttachment.setMode(ClientProxyMode.AWAITING_SERVERPROXY_AUTHENTICATION_RESULT);

            return true;
        } catch (Exception e) {
            return false;
        }
    }

    private byte[] generateConnectionMessage(byte[] randomBytesFromServer, UUID destination) throws
Exception {
        ConnectionMessage message = new ConnectionMessage();
        message.setSenderId(mLocalSettings.getUuid());
        message.setRandomNumber(randomBytesFromServer);
        message.setMessageType(MessageType.APPLICATION_LAYER_CONNECTION);
        message.setDestination(destination);
```

```
        message.setSystemMessage(new byte[] { (byte)0x0 });
        return message.toByteArray(mPrivateKeyString);
    }

    private boolean readServerProxyAuthenticationResult(SocketChannel onionProxyChannel,
OnionProxySelectionKeyAttachment onionProxySelectionKeyAttachment) throws IOException {
        boolean isOk = true;

        ByteBuffer buffer = ByteBuffer.allocate(1);

        int numRead = onionProxyChannel.read(buffer);
        buffer.flip();

        if (numRead == -1) {
            isOk = false;
        } else if (numRead > 0) {
            if (buffer.get() == (byte)0x0a) {
                onionProxySelectionKeyAttachment.setMode(ClientProxyMode.ACCEPTING_PAYLOAD);
                isOk = true;
            } else {
                onionProxySelectionKeyAttachment.setMode(ClientProxyMode.REFUSED_BY_SERVERPROXY);
                isOk = false;
            }
        }

        return isOk;
    }


    private Peer resolveFriend(int port) {
        // TODO: Really resolve the friend's onion address from its port

        List<UUID> friendUuids = PeersDb.selectAllFriendUuids(mContext);
        int friendIndex = port - START_PORT;

        if (friendIndex >= 0 && friendIndex < friendUuids.size()) {
            return PeersDb.selectByUuid(mContext, friendUuids.get(friendIndex));
        } else {
            return null;
        }
    }

    private UUID resolveDestination(int clientPort) {
        // TODO: Resolve capability from port, reverse of binding ports
        return UUID.fromString("60b9abad-0638-468d-92aa-f0bf83a10ab6");
    }

    private synchronized Selector bindPorts() throws IOException {
        if (sServerSocketChannels.size() == 0) {
            sSelector = Selector.open();

            // TODO: Bind to one port for each friend
            for (int i = 0; i < 10; i++) {
                ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
                serverSocketChannel.configureBlocking(false);
                ServerSocket serverSocket = serverSocketChannel.socket();
                InetSocketAddress inetSocketAddress = new InetSocketAddress(START_PORT + i);
                serverSocket.bind(inetSocketAddress);
                serverSocketChannel.register(sSelector, SelectionKey.OP_ACCEPT);

                sServerSocketChannels.add(serverSocketChannel);
            }
        } else {
            for (int i = 0; i < 10; i++) {
                ServerSocketChannel serverSocketChannel = sServerSocketChannels.get(i);

                if (serverSocketChannel == null || !serverSocketChannel.isOpen() ||
serverSocketChannel.socket().isClosed() || !serverSocketChannel.socket().isBound()) {
                    serverSocketChannel = ServerSocketChannel.open();
                    serverSocketChannel.configureBlocking(false);
                    ServerSocket serverSocket = serverSocketChannel.socket();
                    InetSocketAddress inetSocketAddress = new InetSocketAddress(START_PORT + i);
                    serverSocket.bind(inetSocketAddress);
                    serverSocketChannel.register(sSelector, SelectionKey.OP_ACCEPT);
                    sServerSocketChannels.set(i, serverSocketChannel);
                }
            }
        }

        return sSelector;
    }

    private SocketChannel acceptIncomingSocket(SelectionKey selectionKey) throws IOException {
        ServerSocketChannel serverSocketChannel = (ServerSocketChannel) selectionKey.channel();
        Socket socket = serverSocketChannel.socket().accept();
        SocketChannel socketChannel = socket.getChannel();
        socketChannel.configureBlocking(false);
        return socketChannel;
```

```
    }

    private SocketChannel connectOnionProxy(Selector selector) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_CONNECT);
        int socksPort = SwirlwaveOnionProxyManager.getsSocksPort();
        InetSocketAddress localOnionProxyAddress = new InetSocketAddress("127.0.0.1", socksPort);
        socketChannel.connect(localOnionProxyAddress);
        return socketChannel;
    }

    private void closeSocketPairs(SelectionKey selectionKey) {
        SocketChannel inChannel = (SocketChannel) selectionKey.channel();
        closeChannel(inChannel);

        Object attachmentObject = selectionKey.attachment();

        if (attachmentObject instanceof SelectionKeyAttachment) {
            SelectionKeyAttachment selectionKeyAttachment = (SelectionKeyAttachment) attachmentObject;
            selectionKey.attach(null);
            selectionKey.cancel();

            SocketChannel outChannel = selectionKeyAttachment.getSocketChannel();
            SelectionKey outChannelSelectionKey = selectionKeyAttachment.getSelectionKey();

            if (outChannelSelectionKey != null) {
                outChannelSelectionKey.attach(null);
                outChannelSelectionKey.cancel();
            }

            if (outChannel != null) {
                closeChannel(outChannel);
            }
        } else if (attachmentObject != null) {
            selectionKey.attach(null);
            selectionKey.cancel();

            try {
                Pair<Integer, SocketChannel> pair = (Pair<Integer, SocketChannel>) attachmentObject;
                closeChannel(pair.second);
            } catch (ClassCastException cce) {

            }
        }
    }

    private void closeChannel(SocketChannel socketChannel) {
        Socket socket = socketChannel.socket();
        try {
            // Closing the socket, even if channel will be closed.
            // Reason: To close a little bit sooner. The cancel on the channel will not close the socket
until the next select.
            socket.close();
        } catch (IOException ie) {
            Log.e(mContext.getString(R.string.service_name), "Error closing socket" + socket + ": " + ie);
        }

        try {
            socketChannel.close();
        } catch (IOException ie) {
            Log.e(mContext.getString(R.string.service_name), "Error closing channel: " + ie);
        }
    }

    private boolean processInput(SocketChannel inChannel, SocketChannel outChannel, SelectionKeyAttachment
attachment) throws IOException {
        boolean isOk;

        mBuffer.clear();
        int bytesRead = inChannel.read(mBuffer);
        mBuffer.flip();

        // A value of -1 means that the socket has been closed by the peer.
        if (bytesRead == -1) {
            isOk = false;
        } else if (mBuffer.limit() > 0) {
            if (attachment.isClientChannel()) {
                isOk = processDataFromOnionProxy(mBuffer, outChannel, attachment);
            } else {
                isOk = processDataFromClient(mBuffer, outChannel, attachment);
            }
        } else {
            isOk = true;
        }

        return isOk;
    }
```

```
    private boolean processDataFromClient(ByteBuffer inBuffer, SocketChannel outChannel,
SelectionKeyAttachment attachment) throws IOException {
        while(inBuffer.hasRemaining()) {
            outChannel.write(inBuffer);
        }

        return true;
    }

    private boolean processDataFromOnionProxy(ByteBuffer inBuffer, SocketChannel outChannel,
SelectionKeyAttachment attachment) throws IOException {
        while(inBuffer.hasRemaining()) {
            outChannel.write(inBuffer);
        }

        return true;
    }

    public void terminate() {
        mRunning = false;
    }
}
```

Appendix D: Address Change Announcer

```
package com.swirlwave.android.proxies.clientside;

import android.content.Context;
import android.util.Log;
import android.util.Pair;

import com.msopentech.thali.toronionproxy.Utilities;
import com.swirlwave.android.R;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;
import com.swirlwave.android.proxies.ConnectionMessage;
import com.swirlwave.android.proxies.MessageType;
import com.swirlwave.android.settings.LocalSettings;
import com.swirlwave.android.sms.SmsSender;
import com.swirlwave.android.toast.Toaster;
import com.swirlwave.android.tor.SwirlwaveOnionProxyManager;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class AddressChangeAnnouncer implements Runnable {
    private final Context mContext;
    private final LocalSettings mLocalSettings;
    private final String mPrivateKeyString;
    private final UUID mFriendId;
    private long mDelay;

    public AddressChangeAnnouncer(Context context, long delay, UUID friendId) throws Exception {
        mContext = context;
        mLocalSettings = new LocalSettings(mContext);
        Pair<String, String> keys = mLocalSettings.getAsymmetricKeys();
        mPrivateKeyString = keys.second;
        mDelay = delay;
        mFriendId = friendId;
   }

    @Override
    public void run() {
        try {
            Thread.sleep(mDelay);
            announceAddress(mFriendId);
        } catch (Exception e) {
            Toaster.show(mContext, mContext.getString(R.string.error_while_announcing_address_to_friends));
            Log.e(mContext.getString(R.string.service_name), "Error announcing address to friends: " +
e.toString());
        }
    }

    /**
     * This method uses plain sockets without NIO. Friends are announced sequentialially.
     * Maybe rewrite this if neccessary. Runs in its own thread anyway.
     */
    private void announceAddress(UUID friendId) {
        int onionProxyPort = SwirlwaveOnionProxyManager.getsSocksPort();

        // Get current onion-address
        String currentAddress = SwirlwaveOnionProxyManager.getAddress();

        if (currentAddress == null || "".equals(currentAddress)) {
            Toaster.show(mContext,
mContext.getString(R.string.will_not_announce_address_to_friend_because_it_is_empty));
            Log.i(mContext.getString(R.string.service_name), "Will not announce address to friends because
address is empty!");
            return;
        }

        List<Peer> friends;
        if (friendId == null) {
            friends = PeersDb.selectAllPeers(mContext);
        } else {
            friends = new ArrayList<>();
            friends.add(PeersDb.selectByUuid(mContext, friendId));
        }

        Toaster.show(mContext, mContext.getString(R.string.announcing_address_to_friends));

        // Send message to friends
        for (Peer friend : friends) {
            String friendAddress = friend.getAddress();
```

```
            if (friendAddress == null || "".equals(friendAddress)) {
                Toaster.show(mContext, friend.getName() + " " +
mContext.getString(R.string.has_an_empty_address));
                Log.e(mContext.getString(R.string.service_name), friend.getName() + " has an empty
address.");
                continue;
            }

            try (Socket socket = Utilities.socks4aSocketConnection(friendAddress,
SwirlwaveOnionProxyManager.HIDDEN_SERVICE_PORT, "127.0.0.1", onionProxyPort)) {
                // Read random bytes from server
                DataInputStream dataInputStream = new DataInputStream(socket.getInputStream());
                int randomNumber = dataInputStream.readInt();

                // Create address change message
                byte[] message = generateAddressAnnouncementMessage(randomNumber, currentAddress);

                // Send address change message
                DataOutputStream dataOutputStream = new DataOutputStream(socket.getOutputStream());
                dataOutputStream.writeInt(message.length);
                dataOutputStream.write(message);

                // Read response code
                byte responseCode = (byte)0x0a;
//                byte responseCode = dataInputStream.readByte();
                // Got contact. If the friend was waiting for an answer, the it has been given now.
                if (friend.isAwaitingAnswerFromFallbackProtocol()) {
                    PeersDb.updateAwaitingAnswerFromFallbackProtocol(mContext, friend, false);
                }

                if (responseCode == (byte)0x0a) {
                    Toaster.show(mContext, friend.getName() + " " +
mContext.getString(R.string.accepted_address_announcement));
                    Log.i(mContext.getString(R.string.service_name), friend.getName() + " accepted address
announcement.");

                    if (!friend.getOnlineStatus()) {
                        PeersDb.updateOnlineStatus(mContext, friend, true);
                        Toaster.show(mContext, friend.getName() + " " +
mContext.getString(R.string.is_online));
                        Log.i(mContext.getString(R.string.service_name), friend.getName() + " is now
registered as online.");
                    }
                } else {
                    Toaster.show(mContext,
mContext.getString(R.string.friend_rejected_address_announcement) + " " + friend.getName());
                    Log.i(mContext.getString(R.string.service_name), "Friend rejected address announcement
message " + friend.getName());
                }
            } catch (Exception e) {
                Toaster.show(mContext, mContext.getString(R.string.failed_announcing_address_to) + " " +
friend.getName());
                Log.i(mContext.getString(R.string.service_name), "Address could not be announced to friend,
" + friend.getName() + ": " + e.toString());

                if (friend.isAwaitingAnswerFromFallbackProtocol()) {
                    try {
                        Toaster.show(mContext, friend.getName() + " " +
mContext.getString(R.string.is_awaiting_answer_from_fallback_protocol));
                        Log.i(mContext.getString(R.string.service_name), friend.getName() + " is awaiting
answer from fallback protocol.");

                        PeersDb.updateAwaitingAnswerFromFallbackProtocol(mContext, friend, false);

                        new Thread(new SmsSender(mContext, friend.getSecondaryChannelAddress(),
SwirlwaveOnionProxyManager.getAddress())).start();
                    } catch (Exception e2) {
                        Toaster.show(mContext,
mContext.getString(R.string.failure_answering_fallback_protocol_during_address_announcement_to) + " " +
friend.getName());
                        Log.i(mContext.getString(R.string.service_name), "Failure answering fallback
protocol to friend during address announcement to " + friend.getName() + ": " + e2.toString());
                    }
                }
            }
        }
    }

    private byte[] generateAddressAnnouncementMessage(int randomBytesFromServer, String address) throws
Exception {
        ConnectionMessage message = new ConnectionMessage();
        message.setSenderId(mLocalSettings.getUuid());
        message.setRandomNumber(randomBytesFromServer);
        message.setMessageType(MessageType.ADDRESS_ANNOUNCEMENT);
        message.setDestination(UUID.randomUUID());
        message.setSystemMessage(address.getBytes(StandardCharsets.UTF_8));
        return message.toByteArray(mPrivateKeyString);
```

```
    }
}
```

Appendix E: Friend Address Updater

```
package com.swirlwave.android.proxies.serverside;

import android.content.Context;
import android.util.Log;

import com.swirlwave.android.R;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;

import java.util.UUID;

public class FriendAddressUpdater implements Runnable {
    private Context mContext;
    private UUID mFriendUuid;
    private String mAddress;

    public FriendAddressUpdater(Context context, UUID friendUuid, String address) {
        mContext = context;
        mFriendUuid = friendUuid;
        mAddress = address;
    }

    @Override
    public void run() {
        try {
            mAddress = validateAddressFormat(mAddress);

            if (mAddress == null) {
                Log.e(mContext.getString(R.string.service_name), "Invalid address format received from
friend");
                return;
            }

            Peer friend = PeersDb.selectByUuid(mContext, mFriendUuid);

            if (!friend.getOnlineStatus()) {
                friend.setOnlineStatus(mContext, true);
                friend.setAddress(mContext, mAddress);
                PeersDb.update(mContext, friend);
            } else if (!friend.getAddress().equals(mAddress)) {
                friend.setAddress(mContext, mAddress);
                PeersDb.update(mContext, friend);
            }
        } catch (Exception e) {
            Log.e(mContext.getString(R.string.service_name), "Couldn't update friend " + e.toString());
        }
    }

    public static String validateAddressFormat(String address) {
        if (address == null)
            return null;

        address = address.trim();

        if (!address.endsWith(".onion"))
            return null;

        if (address.length() < 7)
            return null;

        return address;
    }
}
```

Appendix F: SMS Receiver

```
package com.swirlwave.android.sms;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.SmsMessage;
import android.util.Log;

import com.swirlwave.android.R;
import com.swirlwave.android.peers.Peer;
import com.swirlwave.android.peers.PeersDb;
import com.swirlwave.android.proxies.clientside.AddressChangeAnnouncer;
import com.swirlwave.android.proxies.serverside.FriendAddressUpdater;
import com.swirlwave.android.service.SwirlwaveService;
import com.swirlwave.android.toast.Toaster;
import com.swirlwave.android.tor.SwirlwaveOnionProxyManager;

public class SmsReceiver extends BroadcastReceiver {
    public SmsReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        try {
            // Get the data (SMS data) bound to intent
            Bundle bundle = intent.getExtras();

            SmsMessage[] msgs = null;

            if (bundle != null) {
                // Retrieve the Binary SMS data
                Object[] pdus = (Object[]) bundle.get("pdus");
                msgs = new SmsMessage[pdus.length];

                // For every SMS message received (although multipart is not supported with binary)
                for (int i = 0; i < msgs.length; i++) {
                    byte[] data = null;

                    msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

                    Peer friend = PeersDb.selectByPhoneNumber(context, msgs[i].getOriginatingAddress());
                    if (friend != null) {
                        // Return the User Data section minus the
                        // User Data Header (UDH) (if there is any UDH at all)
                        data = msgs[i].getUserData();

                        String str = "";
                        for (int index = 0; index < data.length; index++) {
                            str += Character.toString((char) data[index]);
                        }
                        str += ".onion";

                        str = FriendAddressUpdater.validateAddressFormat(str);

                        if (str != null) {
                            friend.setAddress(context, str);
                            friend.setOnlineStatus(context, true);
                            friend.setAwaitingAnswerFromFallbackProtocol(context, true);
                            PeersDb.update(context, friend);

                            if (SwirlwaveService.isRunning() &&
!SwirlwaveOnionProxyManager.getAddress().equals("")) {
                                // Send message back to friend if the Swirlwave service is currently
running.
                                // Don't have to wait long, assuming that service is already ready.
                                try {
                                    new Thread(new AddressChangeAnnouncer(context, 100,
friend.getPeerId())).start();
                                } catch (Exception e) {
                                    Log.e(context.getString(R.string.app_name), "Couldn't send address as
part of SMS fallback protocol: " + e.toString());
                                }
                            }
                        }
                    }
                }
            }
        } catch (Exception e) {
            Log.e(context.getString(R.string.app_name), "Error while processing received SMS: " +
e.toString());
        }
    }
}
```

Appendix G: SMS Sender

```
package com.swirlwave.android.sms;

import android.content.Context;
import android.telephony.SmsManager;

import com.swirlwave.android.R;
import com.swirlwave.android.toast.Toaster;

public class SmsSender implements Runnable {
    private static final short SMS_PORT = 6739;

    private Context mContext;
    private String mPhone;
    private String mAddress;

    public SmsSender(Context context, String friendPhone, String address) {
        mContext = context;
        mPhone = friendPhone;
        mAddress = address;
    }

    @Override
    public void run() {
        sendSms();
    }

    public void sendSms() {
        try {
            Toaster.show(mContext, mContext.getString(R.string.sending_sms_to) + " " + mPhone);
            String messageText = mAddress.replace(".onion", "");
            SmsManager smsManager = SmsManager.getDefault();
            smsManager.sendDataMessage(mPhone, null, SMS_PORT, messageText.getBytes(), null, null);
        } catch (Exception e) {
            Toaster.show(mContext, mContext.getString(R.string.error_sending_sms) + ": " + e.getMessage());
        }
    }
}
```

Appendix H: Connection Message

```
package com.swirlwave.android.proxies;

import com.swirlwave.android.crypto.AsymmetricEncryption;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.UUID;

public class ConnectionMessage {
    private UUID mSenderId;
    private int mRandomNumber;
    private MessageType mMessageType;
    private UUID mDestination;
    private byte[] mSystemMessage;

    public UUID getSenderId() {
        return mSenderId;
    }

    public void setSenderId(UUID senderId) {
        this.mSenderId = senderId;
    }

    public int getRandomNumber() {
        return mRandomNumber;
    }

    public void setRandomNumber(int randomNumber) {
        this.mRandomNumber = randomNumber;
    }

    public void setRandomNumber(byte[] intBytes) throws Exception {
        mRandomNumber = bytesToInt(intBytes);
    }

    public static int bytesToInt(byte[] intBytes) throws Exception {
        try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(intBytes);
DataInputStream dataInputStream = new DataInputStream(byteArrayInputStream)) {
            return dataInputStream.readInt();
        }
    }

    public MessageType getMessageType() {
        return mMessageType;
    }

    public void setMessageType(MessageType messageType) {
        this.mMessageType = messageType;
    }

    public UUID getDestination() {
        return mDestination;
    }

    public void setDestination(UUID destination) {
        this.mDestination = destination;
    }

    public byte[] getSystemMessage() {
        return mSystemMessage;
    }

    public void setSystemMessage(byte[] systemMessage) {
        this.mSystemMessage = systemMessage;
    }

    public static UUID extractSenderId(byte[] bytes) throws Exception {
        try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(bytes); DataInputStream
dataInputStream = new DataInputStream(byteArrayInputStream)) {
            return new UUID(dataInputStream.readLong(), dataInputStream.readLong());
        }
    }

    public static ConnectionMessage fromByteArray(byte[] bytes, String publicKeyString) throws Exception {
        try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(bytes); DataInputStream
dataInputStream = new DataInputStream(byteArrayInputStream)) {
            ConnectionMessage message = new ConnectionMessage();
            message.setSenderId(new UUID(dataInputStream.readLong(), dataInputStream.readLong()));

            byte[] encryptedBytes = new byte[byteArrayInputStream.available()];
            byteArrayInputStream.read(encryptedBytes, 0, byteArrayInputStream.available());
            byte[] contentBytes = AsymmetricEncryption.decryptBytes(encryptedBytes, publicKeyString,
```

```
false);

            processContentBytes(contentBytes, message);

            return message;
        }
    }

    public byte[] toByteArray(String encryptionKeyString) throws Exception {
        try (ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(); DataOutputStream
dataOutputStream = new DataOutputStream(byteArrayOutputStream)) {
            dataOutputStream.writeLong(mSenderId.getMostSignificantBits());
            dataOutputStream.writeLong(mSenderId.getLeastSignificantBits());
            byte[] contentBytes = getContentBytes();
            byte[] encryptedBytes = AsymmetricEncryption.encryptBytes(contentBytes, encryptionKeyString,
false);
            dataOutputStream.write(encryptedBytes);
            return byteArrayOutputStream.toByteArray();
        }
    }

    private byte[] getContentBytes() throws IOException {
        try (ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(); DataOutputStream
dataOutputStream = new DataOutputStream(byteArrayOutputStream)) {
            dataOutputStream.writeInt(mRandomNumber);
            dataOutputStream.write((byte)mMessageType.ordinal());
            dataOutputStream.writeLong(mDestination.getMostSignificantBits());
            dataOutputStream.writeLong(mDestination.getLeastSignificantBits());
            if (mSystemMessage != null) {
                dataOutputStream.write(mSystemMessage);
            }

            return byteArrayOutputStream.toByteArray();
        }
    }

    private static void processContentBytes(byte[] bytes, ConnectionMessage message) throws Exception {
        try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(bytes); DataInputStream
dataInputStream = new DataInputStream(byteArrayInputStream)) {
            message.setRandomNumber(dataInputStream.readInt());
            message.setMessageType(MessageType.values()[(int)dataInputStream.readByte()]);
            message.setDestination(new UUID(dataInputStream.readLong(), dataInputStream.readLong()));

            if (byteArrayInputStream.available() > 0) {
                byte[] systemMessageBytes = new byte[byteArrayInputStream.available()];
                byteArrayInputStream.read(systemMessageBytes, 0, byteArrayInputStream.available());
                message.setSystemMessage(systemMessageBytes);
            }
        }
    }
}
```

Appendix I: Peer

```
package com.swirlwave.android.peers;

import android.content.Context;

import com.swirlwave.android.R;
import com.swirlwave.android.toast.Toaster;

import java.util.Date;
import java.util.List;
import java.util.UUID;

public class Peer {
    public static final long DEFAULT_ID = -1;

    private long mId;
    private String mName;
    private UUID mPeerId;
    private String mPublicKey;
    private String mAddress;
    private int mAddressVersion;
    private boolean mOnlineStatus;
    private Date mLastContactTime;
    private String mSecondaryChannelAddress;
    private List<UUID> mKnownFriends;
    private String mCapabilities;
    private boolean mAwaitingAnswerFromFallbackProtocol;

    public Peer() {
        mId = DEFAULT_ID;
    }

    public Peer(long id) {
        mId = id;
    }

    public Peer(long id, String name, UUID uuid, String publicKey, String address, int addressVersion,
boolean onlineStatus, Date lastContactTime, String secondaryChannelAddress, List<UUID> knownFriends, String
capabilities, boolean awaitingAnswerFromFallbackProtocol) {
        mId = id;
        mName = name;
        mPeerId = uuid;
        mPublicKey = publicKey;
        mAddress = address;
        mAddressVersion = addressVersion;
        mOnlineStatus = onlineStatus;
        mLastContactTime = lastContactTime;
        mSecondaryChannelAddress = secondaryChannelAddress;
        mKnownFriends = knownFriends;
        mCapabilities = capabilities;
        mAwaitingAnswerFromFallbackProtocol = awaitingAnswerFromFallbackProtocol;
    }

    public Peer(String name, UUID uuid, String publicKey, String address, int addressVersion, boolean
onlineStatus, Date lastContactTime, String secondaryChannelAddress, List<UUID> knownFriends, String
capabilities, boolean awaitingAnswerFromFallbackProtocol) {
        this(DEFAULT_ID, name, uuid, publicKey, address, addressVersion, onlineStatus, lastContactTime,
secondaryChannelAddress, knownFriends, capabilities, awaitingAnswerFromFallbackProtocol);
    }

    public long getId() {
        return mId;
    }

    public void setId(long id) {
        mId = id;
    }

    public UUID getPeerId() {
        return mPeerId;
    }

    public void setPeerId(UUID id) {
        mPeerId = id;
    }

    public String getName() {
        return mName;
    }

    public void setName(String name) {
        mName = name;
    }

    public String getPublicKey() {
        return mPublicKey;
```

```
}

public void setPublicKey(String publicKey) {
    mPublicKey = publicKey;
}

public String getAddress() {
    return mAddress;
}

public void setAddress(String address) {
    mAddress = address;
}

public void setAddress(Context context, String address) {
    if (mAddress != null && !mAddress.equals(address)) {
        StringBuilder sb = new StringBuilder();
        sb.append(getName());
        sb.append(" ");
        sb.append(context.getString(R.string.has_changed_address));

        Toaster.show(context, sb.toString());
    }

    mAddress = address;
}

public int getAddressVersion() {
    return mAddressVersion;
}

public void setAddressVersion(int addressVersion) {
    mAddressVersion = addressVersion;
}

public boolean getOnlineStatus() {
    return mOnlineStatus;
}

public void setOnlineStatus(boolean onlineStatus) {
    mOnlineStatus = onlineStatus;
}

public void setOnlineStatus(Context context, boolean onlineStatus) {
    mOnlineStatus = onlineStatus;

    StringBuilder sb = new StringBuilder();
    sb.append(getName());
    sb.append(" ");
    if (onlineStatus)
        sb.append(context.getString(R.string.is_online));
    else
        sb.append(context.getString(R.string.is_offline));

    Toaster.show(context, sb.toString());
}

public Date getLastContactTime() {
    return mLastContactTime;
}

public void setLastContactTime(Date lastContactTime) {
    mLastContactTime = lastContactTime;
}

public String getSecondaryChannelAddress() {
    return mSecondaryChannelAddress;
}

public void setSecondaryChannelAddress(String secondaryChannelAddress) {
    mSecondaryChannelAddress = secondaryChannelAddress;
}

public List<UUID> getKnownFriends() {
    return mKnownFriends;
}

public void setKnownFriends(List<UUID> knownFriends) {
    this.mKnownFriends = knownFriends;
}

public String getCapabilities() {
    return mCapabilities;
}

public void setCapabilities(String capabilities) {
    this.mCapabilities = capabilities;
}
```

```
    public boolean isAwaitingAnswerFromFallbackProtocol() {
        return mAwaitingAnswerFromFallbackProtocol;
    }

    public void setAwaitingAnswerFromFallbackProtocol(boolean awaitingAnswerFromFallbackProtocol) {
        mAwaitingAnswerFromFallbackProtocol = awaitingAnswerFromFallbackProtocol;
    }

    public void setAwaitingAnswerFromFallbackProtocol(Context context, boolean
awaitingAnswerFromFallbackProtocol) {
        mAwaitingAnswerFromFallbackProtocol = awaitingAnswerFromFallbackProtocol;

        StringBuilder sb = new StringBuilder();
        sb.append(getName());
        sb.append(" ");
        if (awaitingAnswerFromFallbackProtocol)
            sb.append(context.getString(R.string.is_now_awaiting_answer_for_fallback_protocol));
        else
            sb.append(context.getString(R.string.is_no_longer_awaiting_answer_for_fallback_protocol));

        Toaster.show(context, sb.toString());
    }
}
```

Appendix J: Peers DB

```
package com.swirlwave.android.peers;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.support.annotation.NonNull;
import android.util.Log;

import com.swirlwave.android.R;
import com.swirlwave.android.database.DatabaseOpenHelper;

import org.apache.commons.lang3.StringUtils;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

public class PeersDb {
    public static final String TABLE_NAME = "peers";

    public static final String ID_COLUMN = "_id";
    public static final String NAME_COLUMN = "name";
    public static final String PEER_ID_COLUMN = "peer_id";
    public static final String PUBLIC_KEY_COLUMN = "public_key";
    public static final String ADDRESS_COLUMN = "address";
    public static final String ADDRESS_VERSION_COLUMN = "address_version";
    public static final String ONLINE_STATUS_COLUMN = "online_status";
    public static final String LAST_CONTACT_TIME = "last_contact_time";
    public static final String SECONDARY_CHANNEL_ADDRESS_COLUMN = "secondary_channel_address";
    public static final String KNOWN_FRIENDS_COLUMN = "known_friends";
    public static final String CAPABILITIES_COLUMN = "capabilities";
    public static final String AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN =
"awaiting_answer_from_fallback_protocol";

    public static final String CREATE_TABLE = "create table " +
            TABLE_NAME + " (" +
            ID_COLUMN + " integer primary key autoincrement, " +
            NAME_COLUMN + " text not null, " +
            PEER_ID_COLUMN + " text not null, " +
            PUBLIC_KEY_COLUMN + " text not null, " +
            ADDRESS_COLUMN + " text not null, " +
            ADDRESS_VERSION_COLUMN + " text not null, " +
            ONLINE_STATUS_COLUMN + " text not null, " +
            LAST_CONTACT_TIME + " text not null, " +
            SECONDARY_CHANNEL_ADDRESS_COLUMN + " text not null, " +
            KNOWN_FRIENDS_COLUMN + " text not null, " +
            CAPABILITIES_COLUMN + " text not null, " +
            AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN + " text not null);";

    public static final String SELECT_ALL = "select " +
            ID_COLUMN + "," +
            NAME_COLUMN + ", " +
            PEER_ID_COLUMN + ", " +
            PUBLIC_KEY_COLUMN + ", " +
            ADDRESS_COLUMN + ", " +
            ADDRESS_VERSION_COLUMN + ", " +
            ONLINE_STATUS_COLUMN + ", " +
            LAST_CONTACT_TIME + ", " +
            SECONDARY_CHANNEL_ADDRESS_COLUMN + ", " +
            KNOWN_FRIENDS_COLUMN + ", " +
            CAPABILITIES_COLUMN + ", " +
            AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN + " " +
            "from " +
            TABLE_NAME;

    public static final String SELECT_WHERE_PEER_ID = SELECT_ALL +
            " where " +
            PEER_ID_COLUMN +
            " = ?";

    private static final String SELECT_WHERE_SECONDARY_CHANNEL_ADDRESS = SELECT_ALL +
            " where " +
            SECONDARY_CHANNEL_ADDRESS_COLUMN +
            " = ?";

    public static final String SELECT_ALL_ADDRESSES = "select " + ADDRESS_COLUMN + " from " + TABLE_NAME;

    private static final String SELECT_ALL_PEER_IDS = "select " + PEER_ID_COLUMN + " from " + TABLE_NAME;

    private static final String ONLINE = "Online";
    private static final String OFFLINE = "Offline";
    private static final String YES = "Yes";
    private static final String NO = "No";
```

```
    public static long insert(Context context, Peer peer) {
        ContentValues values = new ContentValues();
        values.put(NAME_COLUMN, peer.getName());
        values.put(PEER_ID_COLUMN, peer.getPeerId().toString());
        values.put(PUBLIC_KEY_COLUMN, peer.getPublicKey());
        values.put(ADDRESS_COLUMN, peer.getAddress());
        values.put(ADDRESS_VERSION_COLUMN, peer.getAddressVersion());
        values.put(ONLINE_STATUS_COLUMN, peer.getOnlineStatus() ? ONLINE : OFFLINE);
        values.put(LAST_CONTACT_TIME, Long.toString(peer.getLastContactTime().getTime()));
        values.put(SECONDARY_CHANNEL_ADDRESS_COLUMN, peer.getSecondaryChannelAddress());
        values.put(KNOWN_FRIENDS_COLUMN, toString(peer.getKnownFriends()));
        values.put(CAPABILITIES_COLUMN, peer.getCapabilities());
        values.put(AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN,
peer.isAwaitingAnswerFromFallbackProtocol() ? YES : NO);

        long id = DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase().insert(TABLE_NAME, null, values);

        peer.setId(id);

//        context.getContentResolver().notifyChange()

        return id;
    }

    public static int update(Context context, Peer peer) {
        ContentValues values = new ContentValues();
        values.put(ID_COLUMN, peer.getId());
        values.put(NAME_COLUMN, peer.getName());
        values.put(PEER_ID_COLUMN, peer.getPeerId().toString());
        values.put(PUBLIC_KEY_COLUMN, peer.getPublicKey());
        values.put(ADDRESS_COLUMN, peer.getAddress());
        values.put(ADDRESS_VERSION_COLUMN, peer.getAddressVersion());
        values.put(ONLINE_STATUS_COLUMN, peer.getOnlineStatus() ? ONLINE : OFFLINE);
        values.put(LAST_CONTACT_TIME, Long.toString(peer.getLastContactTime().getTime()));
        values.put(SECONDARY_CHANNEL_ADDRESS_COLUMN, peer.getSecondaryChannelAddress());
        values.put(KNOWN_FRIENDS_COLUMN, toString(peer.getKnownFriends()));
        values.put(CAPABILITIES_COLUMN, peer.getCapabilities());
        values.put(AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN,
peer.isAwaitingAnswerFromFallbackProtocol() ? YES : NO);

        String peerIdString = new Long(peer.getId()).toString();

        return DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .update(TABLE_NAME, values, ID_COLUMN + " = ?", new String[] { peerIdString });
    }

    public static void updateOnlineStatus(Context context, Peer peer, boolean online) {
        String peerIdString = new Long(peer.getId()).toString();
        ContentValues values = new ContentValues();
        values.put(ONLINE_STATUS_COLUMN, online ? ONLINE : OFFLINE);

        DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .update(TABLE_NAME, values, ID_COLUMN + " = ?", new String[] { peerIdString });

        peer.setOnlineStatus(context, online);
    }

    public static void updateAwaitingAnswerFromFallbackProtocol(Context context, Peer peer, boolean
awaitingFallbackProtocol) {
        String peerIdString = new Long(peer.getId()).toString();
        ContentValues values = new ContentValues();
        values.put(AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN, awaitingFallbackProtocol ? YES : NO);

        DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .update(TABLE_NAME, values, ID_COLUMN + " = ?", new String[] { peerIdString });

        peer.setAwaitingAnswerFromFallbackProtocol(context, awaitingFallbackProtocol);
    }

    public static Cursor selectAll(Context context) {
        return DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .rawQuery(SELECT_ALL, null);
    }

    public static List<String> selectAllFriendAddresses(Context context) {
        List<String> addresses = new ArrayList<>();

        try (Cursor cursor =
DatabaseOpenHelper.getInstance(context).getWritableDatabase().rawQuery(SELECT_ALL_ADDRESSES, null)) {
            while (cursor.moveToNext()) {
                addresses.add(cursor.getString(cursor.getColumnIndex(ADDRESS_COLUMN)));
            }
        } catch (Exception e) {
```

```
            Log.e(context.getString(R.string.app_name), "Error selecting friend addresses: " +
e.getMessage());
        }

        return addresses;
    }

    public static Peer selectByUuid(Context context, UUID uuid) {
        Peer peer = null;

        try (Cursor cursor = DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .rawQuery(SELECT_WHERE_PEER_ID, new String[] { uuid.toString() })) {

            if (cursor.moveToFirst()) {
                peer = getPeer(cursor);
            }
        } catch (Exception e) {
            Log.e(context.getString(R.string.app_name), "Error selecting peer by uuid: " + e.getMessage());
        }

        return peer;
    }

    public static Peer selectByPhoneNumber(Context context, String phoneNumber) {
        Peer peer = null;

        try (Cursor cursor = DatabaseOpenHelper.getInstance(context)
                .getWritableDatabase()
                .rawQuery(SELECT_WHERE_SECONDARY_CHANNEL_ADDRESS, new String[] { phoneNumber })) {

            if (cursor.moveToFirst()) {
                peer = getPeer(cursor);
            }
        } catch (Exception e) {
            Log.e(context.getString(R.string.app_name), "Error selecting peer by phone number: " +
e.getMessage());
        }

        return peer;
    }

    public static List<UUID> selectAllFriendUuids(Context context) {
        List<UUID> uuids = new ArrayList<>();

        try (Cursor cursor =
DatabaseOpenHelper.getInstance(context).getWritableDatabase().rawQuery(SELECT_ALL_PEER_IDS, null)) {
            while (cursor.moveToNext()) {
                uuids.add(UUID.fromString(cursor.getString(cursor.getColumnIndex(PEER_ID_COLUMN))));
            }
        } catch (Exception e) {
            Log.e(context.getString(R.string.app_name), "Error selecting friend uuids: " + e.getMessage());
        }

        return uuids;
    }

    public static List<Peer> selectAllPeers(Context context) {
        List<Peer> peers = new ArrayList<>();

        try (Cursor cursor = selectAll(context)) {
            while (cursor.moveToNext()) {
                peers.add(getPeer(cursor));
            }
        } catch (Exception e) {
            Log.e(context.getString(R.string.app_name), "Error selecting all peers: " + e.getMessage());
        }

        return peers;
    }

    private static String toString(List<UUID> knownFriends) {
        if (knownFriends == null || knownFriends.size() == 0)
            return "";

        return StringUtils.join(knownFriends, ',');
    }

    @NonNull
    private static Peer getPeer(Cursor cursor) {
        return new Peer(
                cursor.getLong(cursor.getColumnIndex(ID_COLUMN)),
                cursor.getString(cursor.getColumnIndex(NAME_COLUMN)),
                UUID.fromString(cursor.getString(cursor.getColumnIndex(PEER_ID_COLUMN))),
                cursor.getString(cursor.getColumnIndex(PUBLIC_KEY_COLUMN)),
                cursor.getString(cursor.getColumnIndex(ADDRESS_COLUMN)),
                Integer.parseInt(cursor.getString(cursor.getColumnIndex(ADDRESS_VERSION_COLUMN))),
                cursor.getString(cursor.getColumnIndex(ONLINE_STATUS_COLUMN)).equals(ONLINE),
```

```
                new Date(Long.parseLong(cursor.getString(cursor.getColumnIndex(LAST_CONTACT_TIME)))),
                cursor.getString(cursor.getColumnIndex(SECONDARY_CHANNEL_ADDRESS_COLUMN)),
                parseKnownFriends(cursor.getString(cursor.getColumnIndex(KNOWN_FRIENDS_COLUMN))),
                cursor.getString(cursor.getColumnIndex(CAPABILITIES_COLUMN)),

cursor.getString(cursor.getColumnIndex(AWAITING_ANSWER_FROM_FALLBACK_PROTOCOL_COLUMN)).equals(YES)
        );
    }

    private static List<UUID> parseKnownFriends(String knownFriendsString) {
        List<UUID> uuids = new ArrayList<>();

        if (knownFriendsString == null || knownFriendsString.equals(""))
            return uuids;

        for (String uuidString : StringUtils.split(knownFriendsString, ',')) {
            uuids.add(UUID.fromString(uuidString));
        }

        return uuids;
    }
}
```

Appendix K: Local Settings

```
package com.swirlwave.android.settings;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.util.Log;
import android.util.Pair;
import android.widget.Toast;

import com.swirlwave.android.R;
import com.swirlwave.android.crypto.AsymmetricEncryption;
import com.swirlwave.android.toast.Toaster;

import java.util.UUID;

/**
 * Information about the installation's settings, such as phone number, uuid, keys
 */
public class LocalSettings {
    private static final String APP_PREFS = "SwirlwavePreferences";
    private static final String APP_PREFS_INITIALIZED = "PreferencesInitialized";
    private static final String APP_ID_MOST_SIGNIFICANT_BITS = "IdMostSignificantBits";
    private static final String APP_ID_LEAST_SIGNIFICANT_BITS = "IdLeastSignificant";
    private static final String APP_PRIVATE_KEY = "Private";
    private static final String APP_PUBLIC_KEY = "Public";
    private static final String APP_PHONE_NUMBER = "PhoneNumber";
    private static final String APP_INSTALLATION_NAME = "InstallationName";
    private static final String APP_ADDRESS = "Address";
    private static final String APP_ADDRESS_VERSION = "AddressVersion";

    private SharedPreferences mSharedPreferences;
    private UUID mUuid;
    private Pair<String, String> mAsymmetricKeys;
    private String mPhoneNumber;
    private String mInstallationName;

    public LocalSettings(Context context) throws Exception {
        mSharedPreferences = context.getSharedPreferences(APP_PREFS, Activity.MODE_PRIVATE);
        ensurePreferencesExist();
    }

    public void ensurePreferencesExist() throws Exception {
        if (!mSharedPreferences.getBoolean(APP_PREFS_INITIALIZED, false)) {
            SharedPreferences.Editor editor = mSharedPreferences.edit();

            UUID uuid = UUID.randomUUID();
            editor.putLong(APP_ID_MOST_SIGNIFICANT_BITS, uuid.getMostSignificantBits());
            editor.putLong(APP_ID_LEAST_SIGNIFICANT_BITS, uuid.getLeastSignificantBits());

            Pair<String, String> keys = AsymmetricEncryption.generateKeys();
            editor.putString(APP_PUBLIC_KEY, keys.first);
            editor.putString(APP_PRIVATE_KEY, keys.second);

            editor.putString(APP_PHONE_NUMBER, "");
            editor.putString(APP_INSTALLATION_NAME, "");

            editor.putString(APP_ADDRESS, "");
            editor.putString(APP_ADDRESS_VERSION, "0");

            editor.putBoolean(APP_PREFS_INITIALIZED, true);
            editor.apply();

            mUuid = uuid;
            mAsymmetricKeys = keys;
            mPhoneNumber = "";
        }
    }

    public UUID getUuid() {
        if (mUuid == null) {
            mUuid = new UUID(
                mSharedPreferences.getLong(APP_ID_MOST_SIGNIFICANT_BITS, 0L),
                mSharedPreferences.getLong(APP_ID_LEAST_SIGNIFICANT_BITS, 0L));
        }

        return mUuid;
    }

    public Pair<String, String> getAsymmetricKeys() {
        if (mAsymmetricKeys == null) {
            mAsymmetricKeys = new Pair<>(
                    mSharedPreferences.getString(APP_PUBLIC_KEY, ""),
                    mSharedPreferences.getString(APP_PRIVATE_KEY, "")
            );
```

```
        }

        return mAsymmetricKeys;
    }

    public String getPhoneNumber() {
        if (mPhoneNumber == null) {
            mPhoneNumber = mSharedPreferences.getString(APP_PHONE_NUMBER, "");
        }

        return mPhoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        SharedPreferences.Editor editor = mSharedPreferences.edit();
        editor.putString(APP_PHONE_NUMBER, phoneNumber);
        editor.apply();
        mPhoneNumber = phoneNumber;
    }

    public String getInstallationName() {
        if (mInstallationName == null) {
            mInstallationName = mSharedPreferences.getString(APP_INSTALLATION_NAME, "");
        }

        return mInstallationName;
    }

    public void setInstallationName(String installationName) {
        SharedPreferences.Editor editor = mSharedPreferences.edit();
        editor.putString(APP_INSTALLATION_NAME, installationName);
        editor.apply();
        mInstallationName = installationName;
    }

    public String getAddress() {
        return mSharedPreferences.getString(APP_ADDRESS, "");
    }

    public void setAddress(String address) {
        SharedPreferences.Editor editor = mSharedPreferences.edit();
        editor.putString(APP_ADDRESS, address);
        editor.apply();
    }

    public String getAddressVersion() {
        return mSharedPreferences.getString(APP_ADDRESS_VERSION, "0");
    }

    public void setAddressVersion(String addressVersion) {
        SharedPreferences.Editor editor = mSharedPreferences.edit();
        editor.putString(APP_ADDRESS_VERSION, addressVersion);
        editor.apply();
    }

    public String getCapabilities() {
        // TODO: Implement capabilites
        return "";
    }

    public static void ensureInstallationNameAndPhoneNumber(Context context) {
        LocalSettings localSettings;

        try {
            localSettings = new LocalSettings(context);
        } catch (Exception e) {
            localSettings = null;
            Log.e(context.getString(R.string.app_name), "Error opening local settings: " + e.getMessage());
            Toaster.show(context, context.getString(R.string.local_settings_unavailable));
        }

        if (localSettings != null) {
            if ("".equals(localSettings.getPhoneNumber()) ||
"".equals(localSettings.getInstallationName())) {
                localSettings.openLocalSettingsActivity(context);
            }
        }
    }

    public void openLocalSettingsActivity(Context context) {
        Intent intent = new Intent(context, LocalSettingsActivity.class);
        context.startActivity(intent);
    }
}
```

Appendix L: Network Connectivity State

```
package com.swirlwave.android.service;

import android.content.Context;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.net.wifi.WifiInfo;
import android.net.wifi.WifiManager;

import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Enumeration;
import java.util.Locale;

class NetworkConnectivityState {
    private static final int FILE_FRIENDLY_NAME_MAX_LENGTH = 50;
    private static final String FILE_FRIENDLY_NAME_ALLOWED_CHARS = "[^\\dA-Za-z]";
    private Context mContext;
    private boolean mConnected;
    private boolean mWifi; // Maybe this should be "int mNetworkType" instead
    private String mNetworkName = "";
    private String mIp = "";
    private String mFileFriendlyName = "";
    private String mFileFriendlyNamePrevious = "";

    public NetworkConnectivityState(Context context) {
        mContext = context;
    }

    public static String generateFileFriendlyLocationName(String network, String address) {
        String name = String.format("%s%s", address, network)
                .replaceAll(FILE_FRIENDLY_NAME_ALLOWED_CHARS, "")
                .toLowerCase();

        if (name.length() > FILE_FRIENDLY_NAME_MAX_LENGTH)
            name = name.substring(0, FILE_FRIENDLY_NAME_MAX_LENGTH);

        return name;
    }

    public boolean refresh() {
        boolean wasChanged = false;

        mFileFriendlyNamePrevious = mFileFriendlyName;

        ConnectivityManager cm = (ConnectivityManager) (mContext.getSystemService(
                Context.CONNECTIVITY_SERVICE));
        NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

        if (mConnected != hasInternetConnection(activeNetwork)) {
            mConnected = !mConnected;
            wasChanged = true;
        }

        if (mConnected) {
            if (mWifi != isWifi(activeNetwork)) {
                mWifi = !mWifi;
                wasChanged = true;
            }

            String freshIp = getFreshIp(mWifi);
            if (!mIp.equals(freshIp)) {
                mIp = freshIp;
                wasChanged = true;
            }

            String freshNetworkName = getFreshNetworkName(activeNetwork);
            if (!mNetworkName.equals(freshNetworkName)) {
                mNetworkName = freshNetworkName;
                wasChanged = true;
            }
        }

        if (wasChanged)
            mFileFriendlyName = generateFileFriendlyLocationName(mNetworkName, mIp);

        return wasChanged;
    }

    /**
     * After a refresh, this can be checked to see if the location has changed since last time.
     * @return True if the last call to refresh() caused the location name to change
     */
    public boolean locationHasChanged() {
        return !mFileFriendlyName.equals(mFileFriendlyNamePrevious);
```

```
    }

    public boolean isConnected() {
        return mConnected;
    }

    public String getFileFriendlyLocationName() {
        return mFileFriendlyName;
    }

    private boolean hasInternetConnection(NetworkInfo activeNetwork) {
        return activeNetwork != null && activeNetwork.isConnectedOrConnecting();
    }

    private String getFreshNetworkName(NetworkInfo activeNetwork) {
        String extraInfo = activeNetwork.getExtraInfo();
        return extraInfo == null ? "" : extraInfo;
    }

    private boolean isWifi(NetworkInfo activeNetwork) {
        return activeNetwork.getType() == ConnectivityManager.TYPE_WIFI;
    }

    private String getFreshIp(boolean isWifi) {
        if (isWifi)
            return getWifiIp();
        else
            return getMobileIp();
    }

    private String getWifiIp() {
        try {
            WifiManager wifiManager = (WifiManager) mContext.getSystemService(mContext.WIFI_SERVICE);
            WifiInfo wifiInfo = wifiManager.getConnectionInfo();
            int ipAddress = wifiInfo.getIpAddress();
            return String.format(Locale.getDefault(), "%d.%d.%d.%d",
                    (ipAddress & 0xff), (ipAddress >> 8 & 0xff),
                    (ipAddress >> 16 & 0xff), (ipAddress >> 24 & 0xff));
        } catch (Exception ex) {
            return "";
        }
    }

    private String getMobileIp() {
        try {
            for (Enumeration<NetworkInterface> en = NetworkInterface
                    .getNetworkInterfaces(); en.hasMoreElements(); ) {
                NetworkInterface intf = en.nextElement();
                for (Enumeration<InetAddress> enumIpAddr = intf
                        .getInetAddresses(); enumIpAddr.hasMoreElements(); ) {
                    InetAddress inetAddress = enumIpAddr.nextElement();
                    if (!inetAddress.isLoopbackAddress()) {
                        return inetAddress.getHostAddress().toString();
                    }
                }
            }
        } catch (SocketException ex) {
            return "";
        }
        return "";
    }
}
```

Appendix M: Asymmetric Encryption

```
package com.swirlwave.android.crypto;

import android.util.Base64;
import android.util.Pair;

import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

import javax.crypto.Cipher;

public class AsymmetricEncryption {
    private KeyPair mKeys;
    private Cipher mEncryptCipher, mDecryptCipher;

    public AsymmetricEncryption(String publicKeyString, String privateKeyString) throws Exception {
        mKeys = new KeyPair(
                decodePublicKey(publicKeyString),
                decodePrivateKey(privateKeyString));

        mEncryptCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        mEncryptCipher.init(Cipher.ENCRYPT_MODE, mKeys.getPublic());

        mDecryptCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        mDecryptCipher.init(Cipher.DECRYPT_MODE, mKeys.getPrivate());
    }

    private static PublicKey decodePublicKey(String publicKeyString) throws Exception {
        byte[] bytes = stringDecode(publicKeyString);
        return KeyFactory.getInstance("RSA").generatePublic(new X509EncodedKeySpec(bytes));
    }

    private static PrivateKey decodePrivateKey(String privateKeyString) throws Exception {
        byte[] bytes = stringDecode(privateKeyString);
        return KeyFactory.getInstance("RSA").generatePrivate(new PKCS8EncodedKeySpec(bytes));
    }

    public static byte[] decryptBytes(byte[] encryptedBytes, String keyString, boolean isPrivateKey) throws
Exception {
        Key key;
        if (isPrivateKey) key = decodePrivateKey(keyString);
        else key = decodePublicKey(keyString);

        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        cipher.init(Cipher.DECRYPT_MODE, key);
        return cipher.doFinal(encryptedBytes);
    }

    public static byte[] encryptBytes(byte[] cleartextBytes, String keyString, boolean isPublicKey) throws
Exception {
        Key key;
        if (isPublicKey) key = decodePublicKey(keyString);
        else key = decodePrivateKey(keyString);

        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        return cipher.doFinal(cleartextBytes);
    }

    private static String stringEncode(Key key) {
        return Base64.encodeToString(key.getEncoded(), Base64.NO_WRAP);
    }

    private static byte[] stringDecode(String keyAsString) {
        return Base64.decode(keyAsString, Base64.NO_WRAP);
    }

    public static Pair<String, String> generateKeys() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048);
        KeyPair keyPair = kpg.genKeyPair();
        return new Pair<>(
                stringEncode(keyPair.getPublic()),
                stringEncode(keyPair.getPrivate())
        );
    }

    public byte[] encrypt(byte[] bytes) throws Exception {
        return mEncryptCipher.doFinal(bytes);
    }
```

```
    public byte[] decrypt(byte[] bytes) throws Exception {
        return mDecryptCipher.doFinal(bytes);
    }
}
```

Appendix N: Swirlwave Onion Proxy Manager

```
package com.swirlwave.android.tor;

import android.content.Context;

import com.msopentech.thali.android.toronionproxy.AndroidOnionProxyManager;
import com.msopentech.thali.toronionproxy.OnionProxyManager;
import com.swirlwave.android.proxies.serverside.ServerSideProxy;
import com.swirlwave.android.settings.LocalSettings;
import com.swirlwave.android.toast.Toaster;

import java.util.Random;

public class SwirlwaveOnionProxyManager {
    public static final int HIDDEN_SERVICE_PORT = 9344;
    private String mFileStorageLocationPrefix = "tor_files_";
    private OnionProxyManager mOnionProxyManager;
    private static String sOnionAddress = "";
    private static int sSocksPort = -1;
    private Context mContext;

    public SwirlwaveOnionProxyManager(Context context) {
        mContext = context;
    }

    public static String getAddress() {
        return sOnionAddress;
    }

    public static int getsSocksPort() {
        return sSocksPort;
    }

    public void start(String fileFriendlyNetworkName) throws Exception {
        stop();

        mOnionProxyManager = new AndroidOnionProxyManager(
                mContext,
                mFileStorageLocationPrefix + fileFriendlyNetworkName);

        long onionProxyStartupFinished = 0;
        long onionProxyStartTime = System.currentTimeMillis();
        if (mOnionProxyManager.startWithRepeat(240, 5)) {
            sSocksPort = mOnionProxyManager.getIPv4LocalHostSocksPort();
            sOnionAddress = mOnionProxyManager.publishHiddenService(
                    HIDDEN_SERVICE_PORT, ServerSideProxy.PORT);

            onionProxyStartupFinished = System.currentTimeMillis();

            LocalSettings localSettings = new LocalSettings(mContext);
            if (!localSettings.getAddress().equals(sOnionAddress)) {
                localSettings.setAddress(sOnionAddress);
                Integer newVersion = Integer.parseInt(localSettings.getAddressVersion()) + 1;
                localSettings.setAddressVersion(newVersion.toString());
            }
        }

        Toaster.show(mContext, String.format("Started Onion Proxy in %s ms.", onionProxyStartupFinished -
onionProxyStartTime));
    }

    public void stop() throws Exception {
        sOnionAddress = "";
        sSocksPort = -1;

        if (mOnionProxyManager != null)
            mOnionProxyManager.stop();
    }
}
```