A real-world study of mobile peer-to-peer networking

Mobile Peer-to-Peer-Netzwerke in der echten Welt

Master-Thesis von Jan Schejbal

July 2014

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Security Engineering

A real-world study of mobile peer-to-peer networking
Mobile Peer-to-Peer-Netzwerke in der echten Welt

Vorgelegte Master-Thesis von Jan Schejbal

1. Gutachten: Dr. Erik Tews
2. Gutachten: Prof. Dr. Stefan Katzenbeisser

Tag der Einreichung:

Abstract

Research in the field of mobile peer-to-peer communication mostly relies on simulations or artificial testbeds. This thesis presents an Android-based peer-to-peer network run in the real world on nearly 300 personal phones of volunteers. It demonstrates the feasibility of such networks and provides an insight into their practical operation and the challenges to be expected. A rich data set is provided for future research by extensively logging the operation of the network. Additionally, this thesis explores possible applications of decentralized mobile networks.

Keywords: *Android, Bluetooth, data collection, trace, delay-tolerant networking (DTN), MANET, peer-to-peer, store-and-forward, censorship resistance*

Errata, data set download and supporting data
A web site with errata, the generated data set, useful information about its use and a free electronic copy of this thesis will be maintained at:

`http://www.janschejbal.de/permanent/masterthesis`

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 2014-07-01 Jan Schejbal

Table of Contents

1 Introduction **1**

2 Related works **3**
2.1 Delay-tolerant, broadcast based communication systems . . . 3
2.1.1 Rangzen . . . 3
2.1.2 Serval/Rhizome/MeshMS . . . 3
2.2 Other DTN systems . . . 4
2.3 Data sets/mobility traces . . . 5
2.3.1 Lausanne Data Collection Campaign . . . 5
2.3.2 MIT Reality Mining Project . . . 6
2.3.3 DOME/DieselNet . . . 6
2.3.4 SIGCOMM2009 . . . 6
2.3.5 Other traces . . . 6
2.4 Issues with MANET simulations . . . 7

3 The need for data collection **9**
3.1 Combining the approaches . . . 10
3.2 Contributions . . . 10
3.2.1 Comparison with other data sets . . . 11

4 Design of the experiment **13**
4.1 Choice of platform . . . 13
4.2 Choice of wireless technology . . . 13
4.3 Design of the network . . . 15
4.4 Concept of the software . . . 16
4.4.1 Communication . . . 17
4.4.2 Logging . . . 17
4.4.3 Time synchronization . . . 19
4.5 Protocol . . . 20
4.5.1 Optimizations . . . 22
4.5.2 Security . . . 25
4.6 Implementation . . . 25
4.6.1 User interface . . . 25
4.6.2 Background architecture . . . 27
4.6.3 Permissions . . . 36
4.6.4 Randomness . . . 37
4.6.5 Wake locks . . . 37
4.6.6 Source code . . . 37
4.7 Log analysis . . . 38
4.7.1 Custom analysis . . . 38
4.7.2 Log database . . . 38
4.7.3 Visualization tool . . . 39
4.8 Recruitment of volunteers . . . 39
4.8.1 30th Chaos Communication Congress . . . 40
4.9 Privacy . . . 41

5 Challenges **43**
5.1 Lessons learned . . . 43
5.1.1 Unexpected behavior . . . 43
5.1.2 Logging . . . 43
5.1.3 Timing . . . 43
5.1.4 Consent and opt-out . . . 43
5.2 Operating system issues . . . 44
5.2.1 HTTP keepalive problems . . . 44
5.2.2 Service restart issues . . . 45
5.2.3 Interference from SE Android . . . 46
5.2.4 Crash reports . . . 46
5.3 Bluetooth bugs . . . 47
5.3.1 Pairing popups . . . 47
5.3.2 Visibility timeout . . . 47
5.3.3 Bluetooth name . . . 49
5.3.4 Inability to accept connections . . . 49
5.3.5 Reuse of message IDs, node ID transfers . . . 49
5.3.6 Corrupted transmission . . . 50
5.3.7 Total failure of Bluetooth . . . 50
5.3.8 Device reboots . . . 50
5.4 Privacy issues . . . 50
5.4.1 Failure to anonymize error logs . . . 50
5.4.2 Discovery of former participants . . . 51
5.4.3 Logging of device names . . . 53
5.5 Real-time timing information . . . 53
5.5.1 Deriving real-time timing from NTP data . . . 54
5.5.2 Improving timing using connections . . . 54
5.5.3 Estimating drift . . . 55

6 Observations **57**
6.1 Participating devices . . . 57
6.2 Node degrees . . . 59
6.3 Distribution of messages . . . 59
6.4 Throughput . . . 62
6.5 Timing . . . 63
6.6 Impact on regular use . . . 65
6.6.1 Battery/power consumption . . . 65
6.6.2 Memory usage . . . 65
6.6.3 Bluetooth . . . 66

7 The power of decentralized communication networks **67**
7.1 Dangers of censorship-resistant networks . . . 68
7.2 Existing solutions . . . 68
7.2.1 Current solutions . . . 69
7.2.2 Before the internet - the FidoNet . . . 69
7.2.3 Usenet . . . 70
7.3 Concept overview . . . 70
7.4 Feasibility . . . 72
7.4.1 Group size . . . 72
7.4.2 Message load . . . 74
7.4.3 Storage and bandwidth . . . 75

7.4.4 CPU . . . 76
7.5 Design decisions . . . 76
7.5.1 Signature algorithm . . . 76
7.5.2 Subscription management . . . 77
7.6 Potential applications . . . 79
7.6.1 Twitter-like microblogging (one-to-many communications) . . . 79
7.6.2 E-mail-like one-on-one communication with reduced network load . . . 80
7.6.3 Message boards and discussion groups (many-to-many communications) . . . 81
7.6.4 Social networks . . . 82
7.6.5 Multiple services . . . 82

8 Conclusions and outlook **83**
8.1 The data set . . . 83
8.1.1 Limitations . . . 84
8.2 Future research . . . 84
8.3 Outlook . . . 85

Bibliography **87**

Appendices **91**

A Log analysis example using SQL **93**

B Privacy policy/consent form **95**

List of Figures

4.1 Network concept . . . 15
4.2 Simplified chart of communication phases . . . 17
4.3 Excerpt from a log . . . 19
4.4 Message sequence chart of the protocol . . . 21
4.5 Two instances of the protocol run in parallel . . . 22
4.6 Screenshots of the application . . . 26
4.7 Software architecture overview . . . 28
4.8 Bluetooth core state chart . . . 30
4.9 Schema of the log database . . . 39
4.10 Visualization tool . . . 40
4.11 SQL query to select the 30C3 group . . . 41

5.1 Wireshark dump showing keepalive issue . . . 45
5.2 SELinux notification . . . 46
5.3 Stack trace from crash report . . . 46
5.4 Pairing popup on a participating device . . . 47
5.5 Screenshot of “Bluetooth connected” message . . . 48
5.6 Log from client showing a device unable to accept connections . . . 49

6.1 Number of reporting nodes over time . . . 57
6.2 Node presence over time . . . 58
6.3 Message spread - number of nodes reached . . . 60
6.4 Message spread - number of nodes reached within 24 hours . . . 61
6.5 Spread of a selected message . . . 61
6.6 Message propagation example during Congress . . . 62
6.7 Differences between NTP and system time . . . 64
6.8 Examples of clock drift patterns . . . 64
6.9 Bluetooth-spamming billboard . . . 66

7.1 Theoretical distribution of messages . . . 73
7.2 Using broadcast groups to limit network traffic . . . 81

List of Tables

4.1 CONFIG.java configuration options . . . 35

1 Introduction

The ability to communicate freely and without censorship is a critical part of any free society. Providing this ability empowers the victims of oppressive regimes to fight for their freedom. For this reason, oppressive governments fear free communication and fight it with extreme measures. Recent events have shown that even a full shutdown of internet access in an entire country has to be expected when an oppressive regime feels threatened by its people.

Large-scale natural disasters can at the same time create a life-critical need for communications and destroy the infrastructure required to fulfill this need.

Mobile peer-to-peer networks can provide communications even in such situations where regular infrastructure-based networks are unavailable. For this reason, they have become an increasingly popular research topic. However, most research uses either simulations or practical experiments on very small, artificial testbeds.

This thesis attempts to provide insight into the challenges posed by real-world limitations. A simple implementation of a store-and-forward broadcast network is run on real devices of volunteers. Unlike artificial test beds, this provides a realistic sample of different devices and software versions, similar to what would be encountered in a real peer-to-peer network.

Detailed data about the performance of this network is collected and made available for research. To maximize usefulness of the collected data, events from various layers are logged, from the spread of messages in the broadcast network down to Bluetooth discoveries, connection attempts, message transfers and failures. This means that the collected data can be used to build more realistic simulations of mobile P2P networks, even if the proposed network uses a completely different structure than the simulated one. At the same time, a realistic (yet simple) broadcast network is actually demonstrated in practice.

The generated logs are pseudonymized by mapping the already-pseudonymous device identifiers to new values. The pseudonymized data is used for analysis and finally published.

The main objective and contribution of this thesis is providing this comprehensive data set. Additionally, practical issues encountered during the experiment are documented for the benefit of others who want to perform practical mobile peer-to-peer research on the Android platform. This thesis does not aim to evaluate a specific network. For this reason, only very rough analysis is done on the collected data.

After briefly discussing related publications in chapter 2, the need for such a data set is explained in chapter 3. Section 3.2 of this chapter includes a detailed explanation of the contributions made in this thesis.

Chapter 4 explains the design of this experiment and the decisions that were taken, and provides a description of the generated log format. The challenges encountered during this data collection project are analyzed in chapter 5, which also provides “lessons learned” for data collection and mobile peer-to-peer efforts. Chapter 6 presents observations gained both from performing the experiment and from the resulting data, as well as the methods used to analyze the collected information.

Finally, this thesis provides a glance on the power and potential applications of mobile P2P networks in chapter 7, and concludes in chapter 8 with a summary and outlook as well as pointers for future research.

2 Related works

The original motivation for this thesis was to determine how well a mobile, delay-tolerant peer-to-peer network based on common devices can work, mainly for the purpose of providing censorship-resistant communications. Several publications follow this general approach, evaluate concepts important for such networks, and usually provide a simulation-based evaluation of their proposed network. After analysis of existing publications, the decision was made to focus on data collection.

2.1 Delay-tolerant, broadcast based communication systems

These publications propose and/or evaluate networks with concepts similar to the one that drove the design of this experiment. As such, they have a different focus than this thesis (design/evaluation of the network vs. data collection), but could benefit from the collected data. They show and explain the power of such networks and thus the importance of research in the field.

2.1.1 Rangzen

Rangzen [1] proposes the idea of using delay-tolerant networks with (semi-)broadcast distribution of messages to build a microblogging/communication system for censorship resistance. It provides good general insight into the advantages and challenges of the DTN/broadcast approach, and explains why creating an "Internet-like" network with on-line communication capabilities based on mesh routing is difficult.

It focuses on the important aspect of defending such a network against Denial-of-Service/overloading attacks while maintaining privacy. The evaluation is performed using a simulation based on synthetic data. Mobility traces are not used for several reasons. The authors assume that such traces would not properly represent the node population for their use cases. In addition to this, they need information about the social network of the node owners to evaluate their defensive mechanism, which is based on social trust.

Rangzen shows that delay-tolerant broadcast networks can provide an effective and valuable censorship-resistant communication tool, and proposes important core concepts. More importantly, it provides a secure way to limit the impact of malicious attacks against the network, while preserving anonymity. However, all evaluation is done only using simulations, based on synthetic data, thus ignoring many of the challenges such a system would face in the real world.

2.1.2 Serval/Rhizome/MeshMS

The Serval Project[1] provides practical, free solutions for infrastructure-free communication. Focus lies on providing affordable, mesh-based communications to developing countries or areas struck by disasters and connecting remote areas. The project includes development of specialized hardware like a specialized WiFi access point called the WiFi Mesh Extender. Communication is done via WiFi. The Android platform was chosen due to developer experience and low cost. [2]

[1] http://www.servalproject.org/

The main feature of Serval is mesh-based telephony, requiring a routed (non-delay-tolerant) mesh network. [2] However, the Rhizome/MeshMS component adds a delay-tolerant broadcasting component. [3] While this component is primarily meant to allow SMS-like text message delivery, a microblogging feature is mentioned as a possible extension. [3, Sec. 4.3]

Censorship resistance is mentioned as a side note, however it is explicitly stated that this is not an intended use case. [2, Sec. 1.3.1] This is reflected in the security concept: Functionality appears to be considered more important than security, and while some parts of the communication are secured, attacks aimed at disrupting the network itself do not seem to be considered.

Root access is required for Ad-hoc WiFi networking (but not for connecting to makeshift infrastructure like Mesh Extenders) [4]. Due to the undocumented low-level access used by the application to enable ad-hoc mode, the application has severe compatibility issues [5].

The Serval whitepaper [2] also points out the importance of mobile communications, which in turn shows the importance of research on mobile peer-to-peer networks that can provide them. The Serval project performs frequent field tests as described in on their website, their publications [2, 3] and the blog[2] of the project founder. Some of these field tests have shown the true power of delay-tolerant broadcast networks by transferring data across continents. The project focuses on highly practical solutions, which is reflected in the design of the field tests. In near ideal obstacle-free, low-interference environments (e.g. the Australian outback), phone-to-phone ranges of hundreds of meters were observed, and good results were seen even in challenging underground environments using a high device density. [2, Sec. 4] This demonstrates that such networks could be very useful for the targeted audience in developing countries, but does not provide much data about performance in densely populated cities. The tests documented in the above-mentioned publications were performed using large sets of identical devices. A larger-scale test mentioned in [3, Sec. 4.4] is unfortunately not described in detail, but [6] likely refers to this test and indicates that the test operated on 50 identical phones.

Having identical phones is realistic in some scenarios (e.g. dedicated network built for disaster response), but unrealistic in others (e.g. emergency communication between civilians immediately after a disaster).

[3] suggests several optimizations for broadcast networks like confirming the receipt of messages and stopping propagation once a message is confirmed, or limiting traffic according to geographic or topological location. Limitation of WiFi (802.11) broadcast transmissions to 1 Mbit are mentioned, with a suggested workaround of sending unicast packets and letting other nodes sniff them. Such suggestions can provide valuable insight for developers of other mobile peer-to-peer networks. Encryption of payload and addresses is described and designed into the protocol, but not implemented.

Serval is extremely interesting due to its practical approach, the existence of functional prototypes, and impressive results in field tests.

2.2 Other DTN systems

Countless other implementations of delay-tolerant networks exist, mostly for research purposes. Since this thesis focuses on data collection on mobile devices, only a few are given here as examples for the applications of DTNs.

The concept of DTNs was originally created to support deep-space communications and an interplanetary internet. [7] This is still the focus of most DTN research, and thus, most DTNs are based on the idea of smaller, closed systems with trusted nodes.

[2] `http://servalpaul.blogspot.de/`

RFC 4838 [8] describes a DTN architecture based on store-and-forward transmission of “bundles”. It provides a comprehensive definition of many properties of delay-tolerant networks and is a good starting point for anyone wanting an overview of the topic. Another overview is given in [7].

An implementation of this DTN architecture was successfully tested on actual spacecraft in the Deep Impact Networking Experiment (DINET) [9].

However, this thesis focuses on communication in a quickly fluctuating, open network between common consumer devices, not spacecraft. Since the constraints are vastly different in this scenario (e.g. less stringent bandwidth and energy limits, low delays on point-to-point links), the research made on DTNs for space use has only limited applicability for this scenario. More comparable networks are commonly found in the field of Mobile Ad Hoc Network (MANET) research, which has been a fervently studied topic in the last two decades.

This kind of research is likely to profit most from the data collected in this thesis, as the data collected in this experiment is designed to be universally useful, even for differently designed networks, as long as they target the same platform (mobile devices carried by their owners) or use similar technology (i.e. Bluetooth or WiFi).

A DTN similar to the one built in this experiment is DieselNet, described in more detail in section 2.3.3.

PRoPHET [10] describes a more efficient routing algorithm for this kind of DTN, taking a probabilistic approach based on previously seen connections. Bytewalla [11] implements a DTN on Android using PRoPHET.

Finally, section 7.2 of this thesis describes additional historic networks, namely the FidoNet and the early Usenet, which successfully and efficiently distributed messages without persistent connections.

Internet-based broadcast networks are used in practice to disseminate information. For example, the Bitcoin protocol uses such a network to distribute transaction information and alerts to nodes. [12, 13]

2.3 Data sets/mobility traces

The publications presented in this section are similar to this thesis by providing data sets for the simulation and/or analysis of mobile networks.

2.3.1 Lausanne Data Collection Campaign

The Lausanne Data Collection Campaign ran by the Nokia Research Center Lausanne is a very comprehensive, long-term campaign. [14] Location data, Bluetooth visibility information, social interaction data (from call logs, contact lists etc.), usage patterns and more were collected. The collection was done using a homogenous set of Nokia-provided Nokia N95 devices. The device itself as well as monetary compensation for data usage fees served as an incentive. The chosen “viral” recruitment process led to nearly 200 heterogeneous, socially interconnected participants. [15]

This data set is one of the richest data sets available due to the number of participants, the duration of the collection campaign, and the comprehensive collection of data. Unfortunately, the privacy concerns arising from such a comprehensive data collection mean that access to the data set is restricted. No actual peer-to-peer communication was attempted between the nodes.

The paper [16] describing the campaign also includes some useful recommendations on limiting power consumption for location tracking.

2.3.2 MIT Reality Mining Project

The "Reality Mining" data set [17] contains traces of 100 students and faculty of the MIT, collected using a homogenous set of Nokia 6600 devices over a period of six months in 2004. It includes data about Bluetooth visibility, nearby cell towers, as well as user activity (phone usage, call logs). Due to the large number of participants and the long duration, this is also a very popular and valuable data set. However, again, no peer-to-peer communication between the nodes was attempted. Interestingly, issues with the Bluetooth stack were also observed by the authors. A pseudonymized version of the data set is publicly available.

2.3.3 DOME/DieselNet

The University of Massachusetts DOME testbed [18] was a long-term outdoor networking experiment consisting of different kinds of devices:

- Mobile nodes (DieselNet): computers with WiFi capabilities mounted on busses
- Semi-stationary nodes (Throwboxes): movable but usually stationary portable nodes
- A mesh of fixed WiFi access points

A large number of traces from this testbed are available from a public repository[3]. Unlike the examples above, where only visibility/contact information was collected, information about actual data transfers can be found in some of these. This makes it possible to model data transmission based on real data, instead of wireless propagation models of questionable accuracy.

As the nodes were mounted on busses or stationary locations instead of being carried around by participants, human interaction and movement is not reflected in this data set (except for encounters with user-carried devices that were not part of the testbed itself).

2.3.4 SIGCOMM2009

The SIGCOMM2009 data set[4] was collected using a homogenous set of Windows Mobile devices distributed to volunteers during the SIGCOMM conference in 2009. Unlike most other data sets, it includes actual data transmissions between the participating nodes, as well as other data like social media interaction, using a software described in [19].

This is the data set most similar to the data set collected in this thesis. A detailed comparison can be found in section 3.2.1.

2.3.5 Other traces

CRAWDAD ("A Community Resource for Archiving Wireless Data At Dartmouth")[5] is a comprehensive archive of mobile- and wireless-related data traces and tools, containing nearly 100 data sets as of April 2014. While traces containing Bluetooth and/or WiFi visibility/signal strength data are plentiful, traces containing actual node-to-node data transfers are rare. Notable exceptions are the SIGCOMM2009 and DieselNet data sets already mentioned above.

[3] http://traces.cs.umass.edu/
[4] http://crawdad.cs.dartmouth.edu/thlab/sigcomm2009/
[5] http://crawdad.cs.dartmouth.edu/

In addition to this, there are several data sets providing location traces, either for personal mobile devices or public transport vehicles. These can be used in simulations to calculate a simulated signal strength based on the distance between two nodes, with the severe limitations pointed out in section 3. One of the largest location data sets is the Microsoft Research GeoLife GPS trajectory data set [20], involving 182 users and spanning more than three years.

2.4 Issues with MANET simulations

The main reason why practical experiments are highly important are concerns about the inaccuracy of simulations.

"On the Accuracy of MANET Simulators" [21] describes that simulating even a simple scenario in different simulators can lead to "barely comparable" results. This casts serious doubts on the accuracy of simulation-based studies and demonstrates the importance of practical field tests.

"MANET Simulation Studies: The Incredibles" [22] analyzes several research papers on MANETs and comes to the conclusion that "less than 15% of the published MobiHoc papers are repeatable" and the authors suspect "that over 90% of the MobiHoc published simulation results may include bias". Major points noted were unavailability of source code or use of bad/unknown PRNGs - two issues that were carefully avoided during this experiment as described in sections 4.6.4 and 4.6.6.

3 The need for data collection

When researching mobile peer-to-peer systems, two main approaches are commonly used: Testbeds, and simulations. Each of these approaches has its strengths and limitations.

Testbeds using a set of identical devices miss many issues that may arise in the wild, when many different devices from different manufacturers and with different software versions attempt to communicate. At the same time, experiments on testbeds are usually limited by the number of available devices, as well as by the number of volunteers willing to carry such devices in addition to their own. Data collection is limited by practical constraints like privacy laws, battery lifetime, limitations of the hardware platform and the willingness of volunteers to participate. However, testbed experiments do show how such a network behaves in reality, with actual movement, interaction and signal strength patterns. People willing to volunteer in a testbed experiment usually do not represent the general population very well, leading to a selection bias. This may not be an issue if the expected user base is similarly biased, but needs to be considered when using testbeds for research.

Simulations make it possible to test new concepts easily and using a large number of nodes, however, they leave many practical aspects unconsidered. Simulations often generate artificial movement data, calculate the expected signal strengths based on the distance between nodes, then calculate the probability of the success of a data transfer based on the simulated signal quality. Each of these steps adds an additional deviation from reality, making it less likely that the simulation results would match real world performance. For example, physical proximity does not directly translate to signal quality. The type of devices involved, the way the devices are carried, and obstacles between them greatly influence the signal. Interference from other networks and reflection of the signal also impacts performance. Different devices react differently to weak signals, and device-specific issues can prevent certain devices to connect to certain others. Devices using the same technology for different purposes can occupy channels, and some kinds of interference may not be accurately represented by the signal-to-noise ratio reported by devices.

In addition to this, there are serious doubts about the reliability of simulated results in general, as is shown in section 2.4.

To make simulations more realistic, they can be based on actual location or mobile interaction traces. Location traces can replace artificial movement patterns in simulation, which adds realism but reduces flexibility. The issues caused by attempts to derive actual communication behavior from locations/distances still remain. A popular, more accurate way of using real data in simulations is to collect signal strength and/or signal-to-noise-ratio data, since signal-to-noise ratio and communication performance are related. However, as mentioned above, signal strength/SNR still cannot accurately predict the ability to communicate.

Like testbeds, location and interaction traces are often limited by privacy concerns, power requirements, willingness of volunteers to participate and time constrains stemming from this. The selection bias issue applies in a similar way.

Location traces using GPS quickly become useless in buildings. The range of common phone-to-phone communication channels (Bluetooth/WiFi) are severely limited inside buildings (depending on construction type). A distance of less than 15 meters can be sufficient to prevent communication. As buildings are important locations where people interact, this severely limits the usefulness of location traces for simulations.

Having a combination of location and interaction/visibility data can make it easier to understand and visualize interaction traces. However, collecting location traces comes with its own set of problems: Privacy concerns are much stronger with location data than with pure interaction data, and activating the GPS receiver in phones significantly affects battery lifetime.

The most realistic results can - by definition - be obtained by building and deploying a real network. Unfortunately, this does not only require significant effort, it also has other drawbacks. Depending on the type of the results collected, the results are of little use for any other purpose than analysis of a single, specific network. Is difficult to observe all relevant variables, and impossible to simply re-run the experiment to try different parameters. Additionally, practical problems that could be easily avoided in a homogenous testbed can reduce the usefulness of the collected data.

3.1 Combining the approaches

The experiment described in this thesis attempts to combine a real-world networking experiment under realistic and harsh conditions with a detailed data collection campaign designed to be as universally useful as possible.

A full broadcast network is run on the actual, physical devices of the participants during this experiment. This can be used later to verify simulations and derive information about practical systems, including potential issues to be expected. The source code for all used versions of the data collection application is provided together with the results. This ensures that other researchers can verify the results and discover (and possibly correct for) possible mistakes.

Unlike other traces, which only perform discovery, logging visibility and signal strength/SNR of other devices, full connections are established in this experiment and data is exchanged along these connections. This produces a highly realistic data set that can either be used directly for simulation, or used to derive more realistic ways of modeling communication abilities based on other, less detailed traces. For example, visibility and signal quality data is collected as well, making it possible to reduce the data set to a regular Bluetooth visibility trace.

This approach also ensures that software issues or incompatibilities between devices can be discovered, which is necessary to be able to accurately estimate the possible performance of mobile peer-to-peer networks. For example, the assumption of a node density of 100 nodes per km^2 is useless without the information that 40% of these nodes will be unable to receive incoming connections due to software issues.

Several aspects of the experimental network were intentionally not optimized to provide a versatile data set. Details of this network are discussed in section 4.3.

Such a combined approach does not come without a price: Since respecting the privacy of volunteers and keeping battery usage at acceptable levels is critical to get as many participants as possible, a decision not to collect location data in this experiment was made.

3.2 Contributions

The main contribution of this thesis is the resulting data set. It provides data collected by 293 nodes over a time span of several months in two kinds of environments using heterogeneous devices (see section 6.1 for more detailed information about the involved devices). For the largest part of the experiment, the population consisted mostly of students of the Technical University of Darmstadt, creating a low-density network with rare interactions, observed over a term of months. During this time, nearly 100 nodes

contributed data. Towards the end of the experiment, a large number of volunteers was recruited during the 30th Chaos Communication Congress[1] (30C3). These nodes operated in the very dense environment of the congress center, with a high likelihood of radio interference. Nearly 200 nodes joined during this time, and some nodes accompanied their owners from Darmstadt to the 30C3. Additional nodes, which did not upload logs, were observed.

As described above, the log data provides very detailed information, ranging from the state of devices to Bluetooth discoveries to individual packets transferred. From this data set, subsets can be derived as required, making the data useful for a wide range of applications. As a real network was created, the data demonstrates the performance of a simple broadcast-based mobile peer-to-peer network in the real world. The collected data is kept verifiable by providing not only nearly raw logs, but also the source code that generated them.

A large number of technical issues was observed during this experiment, and they are documented in chapter 5. Thus, this thesis provides an important primer for anyone attempting to deploy real peer-to-peer mobile networks on a heterogeneous set of Android devices. Affected versions are documented to the extent they are known, workarounds are provided where possible.

Overall, this thesis gives current and detailed insight into the reality of mobile P2P networking. Finally, some thoughts on the possibilities of mobile P2P networking for censorship resistance and emergency communications are provided in chapter 7.

3.2.1 Comparison with other data sets

The data collection experiment done during this master thesis cannot compete with the scale of the LDCC. Yet, the heterogeneous set of participating Android devices provides a more realistic depiction of real-world environments. While the limited scope of the data collection (e.g. no location data being collected) is an obvious disadvantage, it limits the privacy implications of the data set and makes it possible to distribute the pseudonymized data set freely, unlike the LDCC data.

Similar differences exist between the data from this experiment and the Reality Mining data set. The DieselNet data set does contain wireless communication data, but is based on nodes with very different properties: Busses and people have very different movement and interaction patterns, and the capabilities (e.g. range) of phones vs. specialized WiFi boxes vary significantly.

The SIGCOMM2009 data set is the most comparable one, but lacks the long-term component and shares one important property with the other data sets: All these data sets have been collected using homogenous sets of devices, often of types that are rarely seen in the real world today. The data described in this thesis was collected using the very same Android phones people were carrying during the experiment. Furthermore, Android is a wide-spread, somewhat standardized platform that is unlikely to disappear in the near future.

Data sets with heterogeneous devices are rare, and most data sets only collect signal strength/visibility information instead of actually transferring data between the devices. As is shown in this thesis (see e.g. section 5.3), visibility does not necessarily mean that data can be transferred - *especially* in heterogeneous networks. Thus, the combination of these two properties makes this data set an excellent foundation for simulations.

[1] https://events.ccc.de/congress/2013/wiki/Main_Page

4 Design of the experiment

To create the experimental network and collect the resulting data, an Android app communicating with other nodes via Bluetooth was created and deployed on the phones of volunteers. This section describes the decisions made while designing the experiment.

4.1 Choice of platform

Smartphones provide all resources required for mobile peer-to-peer networking. They are highly portable, general purpose computers with built-in wireless communication capabilities, power supply and touchscreens for user interaction.

Since many people already own a smartphone, using them as a platform avoids the need to purchase and distribute special hardware. This makes them a highly attractive platform for both experimental and productive mobile peer-to-peer networks. For these reasons, the decision was made to use smartphones as a platform for this project.

Using custom hardware is not feasible for a project of this scale, but might be attractive for domain-specific mobile peer-to-peer applications, e.g. for communication between emergency responders. Customized hardware can be tailored to the specific needs, e.g. by including better radio capabilities with longer range, and can provide a stable, uniform platform. For peer-to-peer applications aimed at the general public, using (already existing) smartphones will likely remain the most economic and practical approach.

Custom hardware usually is severely outdated by the time it is ready for deployment, and even more when it reaches the end of its operational life. By using smartphones as a platform, it is reasonably easy to upgrade the underlying hardware to take advantage of new developments like the introduction of WiFi Direct - only small changes to the software are required, and the new hardware is either already present or can be easily purchased.

Creating compatible software for different mobile platforms would, while theoretically possible, exceed the scope of this thesis. Therefore, the decision was made to limit the experiment to a single platform. The Android platform was chosen since it is by far the most common smartphone platform in Germany [23, 24]. Additional advantages are previous development experience and the openness of the platform, making deployment of custom software easy.

A disadvantage of the Android platform is the large diversity both in hardware and in software. In the experiment, 92 different models from 21 manufacturers participated, with 14 different Android versions. (Details about these numbers can be found in section 6.1). Many of the different hardware and software versions have specific properties or issues, often causing problems that could be avoided by using a single, known-good configuration. Some software versions had to be specifically excluded due to bugs that made them unusable for the purpose of this experiment. The issues are described in detail in section 5.2.

4.2 Choice of wireless technology

Suitable candidates for the wireless technology were WiFi and Bluetooth. Both are supported by most Android phones. WiFi provides higher throughput and less energy consumption for large data transfers. While it is often claimed that Bluetooth is more efficient for small, sporadic data transfers, this is

disputed [25]. The cost of frequent discovery/connection establishment has to be considered, and new developments in wireless chipset design are likely to quickly invalidate any results on this topic.

For WiFi, three operation modes are possible: Ad-hoc connections between devices, an “infrastructure” mode where one device acts as an access point and another as a client, and the WiFi Direct abstraction layer specifically designed for peer-to-peer connections.

Ad-hoc connections are not possible on the Android operating system without significant modifications, which would require root access. Requiring a rooted phone would drastically reduce the number of compatible devices and thus of potential participants.

Putting a device into a mode where it emulates an access point is only possible through the use of tethering (or root access). Using tethering would not only require manual configuration, it would also give third parties access to the phone’s mobile internet connection, potentially generating costs for the participants.

WiFi Direct is only accessible on phones with Android version 4.0 or newer, and only if the hardware platform supports it. At design time, the version requirement seemed like a significant disadvantage. However, it turned out that Bluetooth on Android versions below 2.3.5 is unusable for the intended purpose, making the version requirement less significant. The hardware requirement still poses a significant limitation.

The most important argument against the use of WiFi is the fact that any of these modes, including WiFi Direct, monopolizes the WiFi radio on many devices. The WiFi alliance explicitly states[1] that “Simultaneous connection to a Wi-Fi Direct-certified group and an infrastructure network is an optional feature”„ and most tested devices were unable to use WiFi Direct and an infrastructure network at the same time. This would prevent usage of regular WiFi for internet access for the duration of the experiment - a restriction that would be unacceptable to most participants. However, WiFi would have the advantage of offering true broadcast, reducing the required transmission energy if multiple devices are in range.

Modern devices running current versions of the Android operating system (e.g. Nexus 4, Nexus 5 and second-generation Nexus 7 devices running 4.4.x) appear to be able to use infrastructure-based Wi-Fi while communicating to other devices using WiFi Direct. Unfortunately, these devices were not sufficiently wide-spread during the experiment.

Bluetooth, on the other hand, supports multiple simultaneous connections even on older devices and is used less frequently than WiFi in practice, thus reducing the impact on normal use of the phone. For these reasons, Bluetooth was selected as the wireless technology for this experiment.

For future practical implementations of mobile peer-to-peer networks, WiFi Direct should be strongly considered, especially it is expected that large amounts of bandwidth will be required and simultaneous connections to regular WiFi networks are supported by modern devices (or unnecessary in the desired scenario).

WiFi is also better supported by laptop computers, which could serve as especially powerful nodes in such a network. Using WiFi, existing access points could be used to boost range and connect many mobile devices at the same time, and possibly provide connectivity over longer distances using still working parts of a wired network.

Tests with a small number of current devices showed that the practical ranges of Bluetooth and WiFi are comparable when connecting two mobile devices to each other. Both operate in the same frequency band, making observations about Bluetooth visibility transferable to WiFi.

[1] `http://www.wi-fi.org/discover-wi-fi/wi-fi-direct`

4.3 Design of the network

The main goal in the design of the peer-to-peer network was to keep it as simple as possible to provide universally-usable data, while still creating a practical, working broadcast network that could be used for real applications. Thus, the simplified network ran by the application is derived from a concept meant for actual, productive use. That concept is described in detail in section 7.3.

Messages are organized into *channels* with one channel per node. The channel contains all messages created by the corresponding node. The channel ID is identical with the node ID. Each message is uniquely identified within the channel by a sequential number. The channel (or node) ID together with this message ID uniquely identify a message in the network.

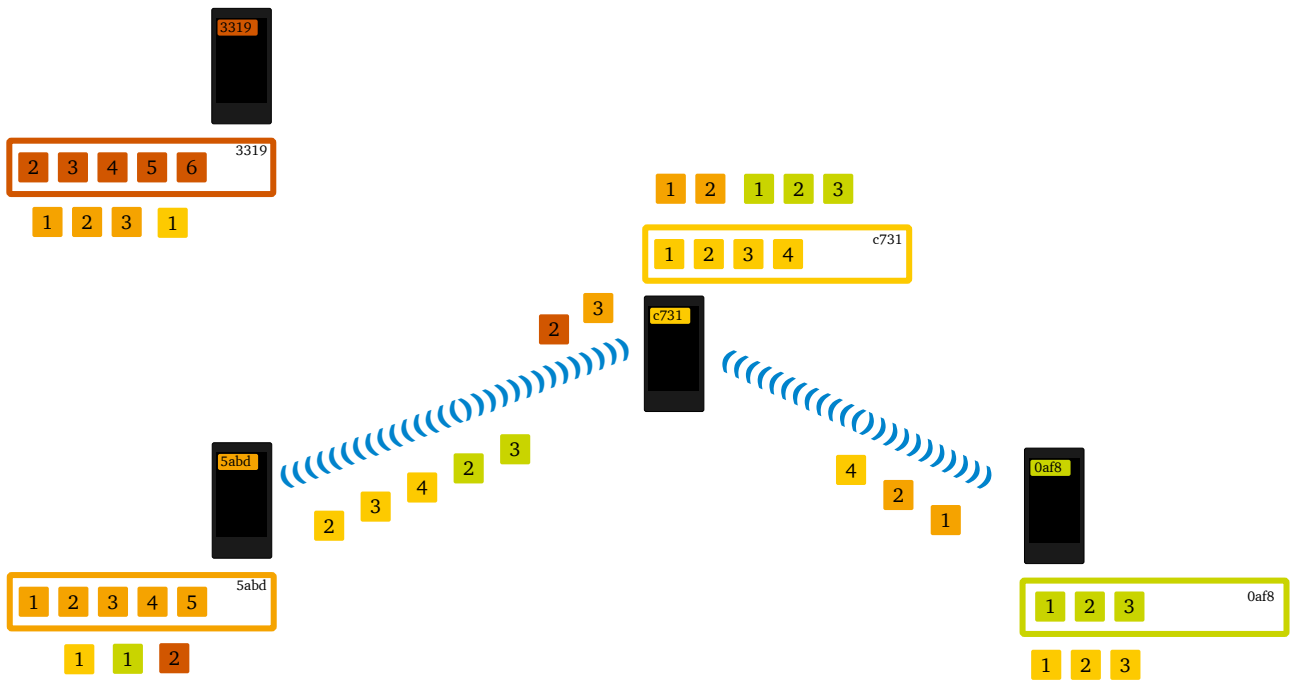

Figure 4.1.: Network concept: Each node has a channel, and provides all messages to other connected nodes that do not have them. Thus, messages are distributed across the network even if the source node is offline.

Channel/node IDs are 8-byte binary values, while message IDs are 4-byte integers. Only the most current 512 messages of the channel are considered relevant. Older messages get automatically discarded, i.e. if a node receives a message with ID 2511, it discards all messages with IDs smaller than 2000. Digital signatures are not used in the experiment.

8 bytes (64 bits) are sufficient to make it extremely unlikely that two of the randomly generated IDs used in the experiment collide. Even assuming 1000 participants, the probability of a collision would be negligible: Using the square approximation $n^2/2m$ of the birthday problem gives a collision probability of $1000^2/\left(2\cdot 2^{64}\right) \approx 3\cdot 10^{-14} \approx 0.00000000000\%$. For real networks with a larger number of users, 2^{64} IDs would be more than sufficient assuming controlled assignment, but insufficient if the IDs were assigned randomly - especially considering that an attacker could attempt to create collisions intentionally.

4-byte integers allow the creation of 2^{31} messages if they are treated as signed and only positive IDs are allowed. Assuming a node would create one message per second, this is sufficient for 68 years. Thus, 4-byte message IDs are perfectly sufficient even for real networks.

This allows to represent a channel state by storing the channel ID, a 512-bit bitmap, and an offset identifying which messages are represented by the channel bitmap. For example, if the offset is 1000, the 0th bit of the bitmap represents message 1000, and the 511th bit of the bitmap represents message 1511 of said channel. Using this, nodes can efficiently store and communicate which messages they currently have, using 76 bytes per channel (8 for the channel ID, 4 for the offset, and 64 for the bitmap), or ~15 KiB for a full state containing information about 200 channels.

There are more efficient set reconciliation algorithms than simply exchanging full channel states. For example, [26] describes one such algorithm, gives potential tradeoffs and pitfalls, and compares it with several other algorithms. Ultimately, the main reason why such a simple system was chosen was simplicity. Due to the reasonably small number of channels expected in this experiment, the benefits of potentially reducing the bandwidth costs were greatly outweighed by the risks of introducing additional complexity.

For the purposes of the experiment, the messages are generated by each node at a fixed rate while the application is running.

During a handshake, the channel states are exchanged and used to determine which messages a node is missing. The node then requests the missing messages and receives them from the other node. Every message has a fixed payload length of 1024 bytes. This is sufficient for example for a short text message and a digital signature and covers common use cases (see section 7.4.2).

Several optimizations have been made: Channels which have not received any new messages for seven days are not advertised to avoid cluttering the network with old messages of dead nodes. Nodes remember when they last contacted another node and what state it had at that point, allowing to transmit only information that changed when connecting again to the same node. To reduce storage requirements, only the information which messages a node has are actually stored. The message payload is replaced with random data, generated on-the-fly when a message is sent and discarded by the recipient. This is described in detail in section 4.5.1.

Further optimizations were avoided to keep the data more useful for research into other kinds of networks. For an actual peer-to-peer network, it would be highly recommended to implement additional optimizations. Possible optimizations could include:

- Refusing incoming connections during discovery
- Optimizing node selection (e.g. prioritizing nodes based on the last-seen timestamp)
- Sending push notifications when a new message is received
- Avoiding the needless repetition of connections when no state was changed
- Using multiple connections and building temporary low-latency P2P networks
- Using broadcast communications instead of emulating them via multiple point-to-point connections
- Using existing WiFi access points

4.4 Concept of the software

A software application deployed on each participating node implements the network, collects the data, and reports them to a central server. This section gives an overview of the building blocks of this application and design decisions that were made during development. Details of the implementation can be found in section 4.6.

4.4.1 Communication

While data collection is running, the application continuously accepts connections.

The outbound communication is separated into three phases: Discovery, communication, and waiting. During the discovery phase, a Bluetooth discovery is run and participating devices are logged as they are discovered. Non-participant devices are logged anonymously, i.e. only the fact that such a device was discovered is logged. (See section 5.4.2 for an issue that causes former participants to be logged accidentally). Signal strength is logged if available. Bluetooth addresses are never logged, except for beacons. Bluetooth "beacons" are specially named devices, to be placed at strategic locations where participants are expected to meet. They offer themselves for discovery and are logged to provide location context and make logs easier to interpret, but perform no other function. Ultimately, these were not deployed in the field since some participants voiced privacy concerns and finding suitable locations for deployment was difficult. All beacon log entries are results of testing and should be ignored.

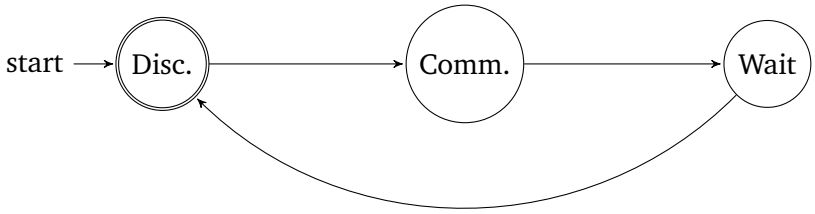

Figure 4.2.: Simplified chart of communication phases

After discovery is finished, the application attempts to connect to the discovered nodes in random order, until either a connection to each node has been attempted or the time limit for the communication phase is exceeded. If a connection is successful, the application waits until it finishes before the next connection is attempted. To avoid duplicate transfers of messages, nodes for which a current incoming connection exists are skipped. Since this is just an optimization and not critical for correctness, this is not enforced strictly; unlucky timing may still allow a concurrent connection.

Following the communication phase, a wait phase occurs during which leftover connections from the communication phase may finish, but no new connections are created. All outbound connections are killed at the end of the wait phase. This stops hanging connections from blocking the entire application and prevents resource conflicts between communication and discovery.

As soon as a connection is established, the protocol is run over the connection as explained in 4.5. Every step in the connection process is logged in detail.

4.4.2 Logging

All events are written into a central, tab-separated plaintext log file. The log file is written in gzip-compressed format to the application data folder on internal storage. This folder is not accessible to other applications to protect the privacy of the volunteers.

Each log line contains two time stamps in microseconds - the system clock time, and a relative timestamp based on the `elapsedRealtime()` timer (see section 4.4.3). This is followed by a tag indicating the primary type or source of the log message, optionally followed by additional fields containing e.g. a subtype or values.

Each node generates a random node ID, which serves as a pseudonymous identifier for the device. Other identifiers of participating devices, like MAC addresses or IMEIs are not logged for privacy reasons. (MAC addresses of beacons are logged to detect possible low-effort attempts at creating fake beacons). Important information about the system (e.g. device model, software version) is included in every log.

A new log is opened after a certain time or when a certain amount of log data is exceeded. Logs are uploaded over a secure connection to a central log collection server if WiFi is available.

Logs are highly verbose and focus on logging individual events to make them useful for a wide range of research. They include both information directly related to device-to-device communication and information that can be used to debug and verify the correct operation of the application. The following types of events and data are logged:

- state transitions representing the communication phases
- discovered devices
- connections to other devices (successful and failed)
- sent and received protocol messages (includes sent and received broadcast messages)
- creation of broadcast messages
- loading/saving of broadcast states and the states themselves
- NTP time pulls (both successful and failed)

- starting/stopping the data collection
- changes of the Bluetooth configuration
- creation/destruction and state changes of application components
- Android shutdown events
- wake locks
- kill switch state changes
- errors
- some method invocations

- device and application information (application version, build date, whether the device is on the main or experimental network, free disk space, node ID, information about the hardware platform and Android version)
- a unique identifier for each instance of the application (changes on application restart)
- whether the device is affected by the Bluetooth discoverability issue described in section 5.3.2.
- battery level

A detailed list accompanies the data set and the software release.

init	out		
remote name	P2P-4d7e836f72d7469a		
Received	HELLO	4d7e836f72d7469a	3eff9455
Sending	STATE	1	
Received	STATE	0	
Sending	REQUEST	0	
Received	REQUEST	1	
Sending	PAYLOAD	00fcd028d80e18af	134
Sending	ENDOFDATA		
Received	ENDOFDATA	-2083795262	15938515
Sending	FINISHED		
Received	FINISHED		
done			
closing			

Figure 4.3.: Excerpt from a log: Omitting the timestamps, tag, and connection ID, these are the log entries generated during a typical connection.

4.4.3 Time synchronization

System clocks cannot be used as a reliable time source for a number of reasons. On some devices, the time is set solely by the user, causing clock drift to accumulate and making incorrectly set clocks likely.

Others use automatic time synchronization using NTP or GPS. In theory, this should provide accurate time. However, Android bug reports indicate [2] that some devices set time using GPS, incorrectly calculating the offset between UTC and GPS time, leading to a constant ~15 second offset.

Clock stability strongly varies between devices. To test clock stability, automatic time synchronization was disabled on a Galaxy Nexus running stock Android 4.2.2 and a Nexus S running CyanogenMod 10.1-20130411 (which is also based on Android 4.2.2). The offset between the system clock and the time obtained via NTP (using 0.de.pool.ntp.org) was measured repeatedly over a period of 15 hours. Since the samples were taken over WiFi with the screen powered on and the device unlocked, the NTP round-trip time varied between 20 and 34 ms for the samples that were considered, giving good accuracy. The system clock on the Galaxy Nexus device lost approximately 50 ms during a period of 24 hours. The Nexus S, however, gained approximately 1.1 seconds during that time period.

Additionally, the Nexus S device (which did not have a SIM card and thus may have been unable to retrieve the current time from a mobile network via NITZ) had an initial offset of 5.6 seconds at the start of the experiment, while the Galaxy Nexus had an offset of only 0.2. Reports on the Android bug tracker [3] indicate drift rates of uncorrected system clocks on the order of several seconds per day.

Android provides three clocks to measure time [27, SystemClock]:

- The regular real-time clock accessible via `System.currentTimeMillis()`. This will provide time as an absolute value, but will be affected by clock adjustments. This is referred to as “system clock time” or “wallclock” time.
- A relative time counter, `SystemClock.elapsedRealtime()`, indicating the real time passed since startup. This is unaffected by time adjustments, but only provides relative timing.

[2] Android Issue 5485 - `http://code.google.com/p/android/issues/detail?id=5485`
[3] Android Issue 18681 - `http://code.google.com/p/android/issues/detail?id=18681`

- A relative time counter, `SystemClock.uptimeMillis()`, measuring only time during which the CPU was active. Time spent in standby does not count towards this counter.

Both the value of the real-time clock and the elapsedRealtime counter are used to timestamp logged data. During the above-mentioned experiment, the difference between elapsedRealtime and the system clock did not change, indicating that both are fed from the same, drifting timer.

The `uptimeMillis` timer is mostly useless for the purpose of this experiment and never used. The word "uptime", e.g. seen in the SQL dataset, refers to either the `elapsedRealtime` timer value or a validity period of this timer identified by an uptime ID (see section 4.6.2.2).

For this reason, the system clock cannot be used to reliably determine absolute time values with precision to the second. Furthermore, elapsedRealtime also drifts significantly and needs to be synchronized frequently (e.g. via NTP) to obtain accurate timings.

To obtain useful real-time references, the application occasionally performs (S)NTP lookups if WiFi is available and logs the response. This allows to correct clock drift during analysis of the logs. Details of the implementation are further explained in 4.6.2.14. Data about system clock differences and drift can be found in section 6.5.

Due to the decision to use NTP only over WiFi for latency and data usage reasons and the system clock drift observed on some devices, sub-second accuracy for log synchronization cannot be reliably achieved. This needs to be considered when analyzing the logs.

4.5 Protocol

The Bluetooth connection between two devices provides a TCP-like bidirectional reliable streaming transport. Over this transport, messages are sent using a simple type-length-value (TLV) format. Details about this lower layer of protocol handling can be found in section 4.6.2.12.

It is important to distinguish between *protocol* messages, i.e. TLV bundles sent over Bluetooth, and *broadcast* messages (the messages being distributed using the broadcast network). In this section, the word "message" used by itself refers to protocol messages. In other parts of this thesis, unless explicitly noted, the word "message" refers to broadcast messages.

One instance of the protocol consists of node A initiating the protocol by sending a hello message, node B responding with a list of available broadcast messages, node A requesting the broadcast messages it wants, node B providing these broadcast messages, and the connection being finalized. Thus, running one instance will only allow node A to download missing broadcast messages from node B, but not the other way around. For this reason, two instances of the protocol are run in parallel over every connection. This makes it irrelevant which node initiated the connection on the Bluetooth layer, since each node will start one instance of the protocol. Figure 4.4 shows a single instance of the protocol.

The protocol begins with node A sending `MESSAGE_HELLO` (represented by the `HelloMessage` class). This message contains the node ID, the minimum application version this node is willing to talk to, the application version used by the node, the identifier it uses for the underlying connection, and two values used to optimize message synchronization (described in section 4.5.1.1). The node ID is matched against the Bluetooth device name. The version information can be used to exclude severely outdated or buggy nodes. Each node logs the local connection identifier when logging information relevant to the connection. The remote connection identifier from the hello message is included in the logs, making it possible to find the corresponding log entries in the log generated by the remote node. It should be noted that the connection IDs cannot be guaranteed to be perfectly unique, though it is unlikely that the same

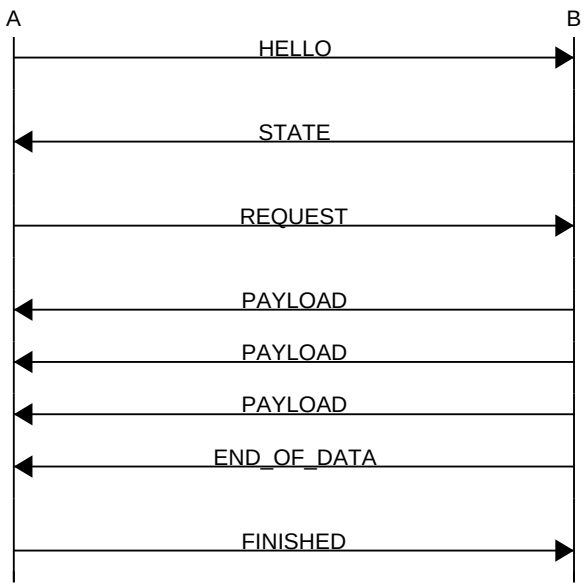

Figure 4.4.: Message sequence chart of the protocol

node will reuse the same ID in practice. (A small number of connection IDs indeed appear more than once in the collected data set).

In response to the hello message, node B will generate a list of the available broadcast messages (using the local broadcast state), and send a `MESSAGE_STATE` message with this list to node A. This list is also called the *offer*. Two optimizations are used to reduce the size of this list, see section 4.5.1 for details. This message is of the type represented by the `ChannelMessage` class and consists of a list of serialized channel states (see section 4.6.2.10 for details about channel states). This message is sent even if it is empty, since it is required to trigger the next step.

Node A uses the offer from node B to determine which broadcast messages should be retrieved (i.e. the ones that are neither present on node A nor too old). Using the same `ChannelMessage` format, it sends the `MESSAGE_REQUEST` message, requesting these messages from node B. This message is sent even if it is empty, since it is required to trigger the next step.

Node B sends the requested broadcast messages, each wrapped in a `MESSAGE_PAYLOAD` protocol message (represented by the `PayloadMessage` class). Since only metadata about the broadcast messages is stored to reduce storage requirements, the payload protocol message contains the channel ID and message ID identifying the message, followed by 1024 bytes of random data. If no messages have been requested, no payload messages are sent.

After the last payload message (or immediately if none are sent), node B sends a `MESSAGE_END_OF_DATA` message to inform the other node that no more messages will follow. This message is empty aside from information used for optimization of the state exchange (see section 4.5.1) and signifies the end of the protocol.

However, since two protocol instances are run over the same Bluetooth connection, and since (unlike with TCP) closing any input/output stream immediately tears down the connection, even destroying already sent data (see section 4.6.2.12), the connection can only be closed once *both* protocol instances are completely finished, i.e. *both* end-of-data messages have reached their destination. For this reason, the receipt of the end-of-data message is confirmed by sending an empty message (`MESSAGE_FINISHED`, represented by class `FinishedMessage`). As soon as one node has received both the end-of-data message and the finished message from the other node, the connection is torn down. (This is sufficient: The

node has received the end-of-data message sent to him, and knows from the finished message that the end-of-data message sent to the other node has reached it).

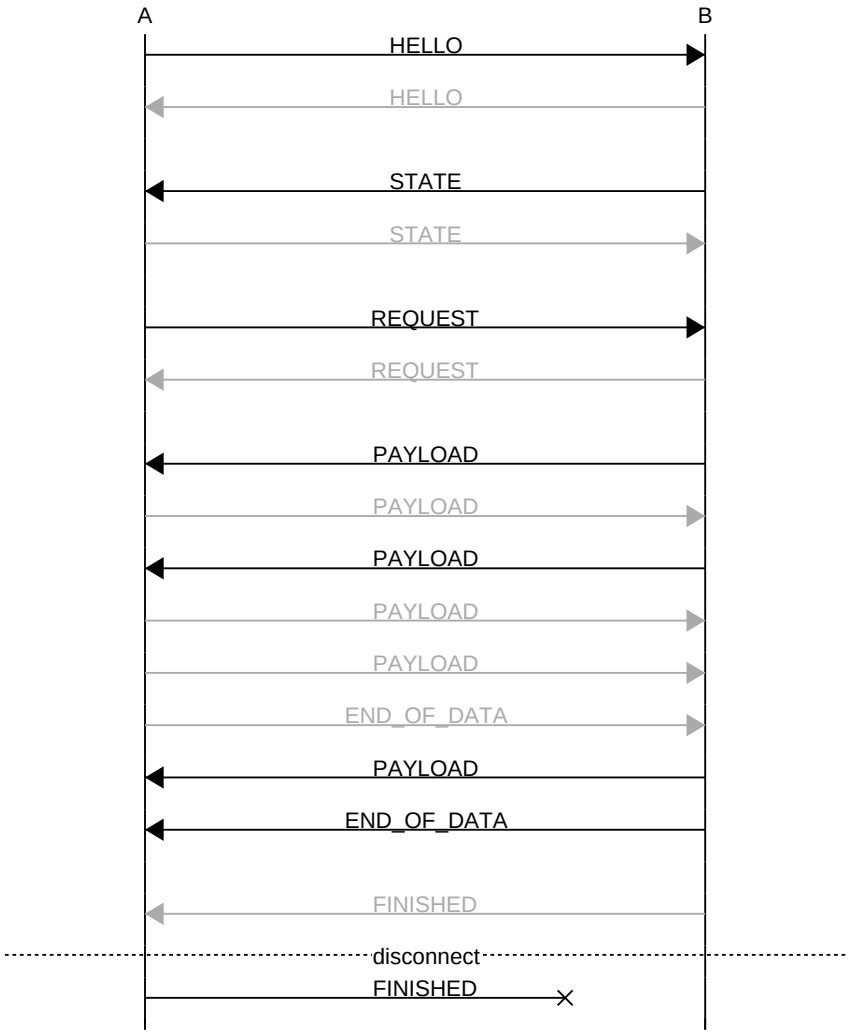

Figure 4.5.: Two instances of the protocol run in parallel

Figure 4.5 shows a full connection with both instances of the protocol (one instance is shown in black, while the other is shown in grey). While the order of messages within one instance is fixed, messages from the two instances can be arbitrarily interleaved. In theory, one instance of the protocol can finish before the other even gets to the state-exchange phase. In this example, this can be seen during the transfer of the payload messages. As soon as the first "finished" message arrives, the connection is torn down. Thus, the second "finished" message usually gets lost. Since this message is expressly designed for this purpose and does not contain any data, this is not a problem.

4.5.1 Optimizations

Several optimizations reduce the amount of data transferred and stored.

4.5.1.1 Avoiding unnecessary state transfer

Without optimizations, the entire state would be transferred with every connection. Since two nodes placed next to each other for extended periods of time will communicate very frequently, this would

cause large amounts of unnecessary data transfer and thus battery drain. At the same time, large parts of the state stay constant, making it unnecessary to exchange the full state every time.

A naive approach would be to save the last state sent to every node and only send changes in subsequent exchanges. This approach would have been feasible for a small network like the one in this experiment: Even 200 nodes would only require about 15 KB of state, so even storing a copy for each node would only require 3 MB of memory. However, this approach clearly does not scale: The size of the state grows linearly with the number of nodes in the network, and a copy of the state would need to be saved for every remote node contacted. As the number of nodes in contact with each node will typically grow as the network grows, this results in nearly quadratic storage requirements.

For this reason, a different approach is used. A monotonically increasing timer is used to tag each local channel state with a timestamp that gets updated whenever the state of that channel changes. Since this timer is reset when the device is rebooted, this timer is coupled with a random per-instance value - the "instance ID". This ID is lost and regenerated when the application restarts (thus guaranteeing that a reboot, which resets the timer, will also reset the ID).

At the beginning of a connection, before the offer is calculated, the offering node stores the current local timestamp in a temporary variable. It then calculates and sends the offer, waits for the requesting node to request messages, and provides the requested messages. In the end-of-data message, it includes the instance ID and timestamp. The requesting node now saves these two values. The offering node does not have to remember any node-specific information.

During the next connection, the requesting node includes these values in the hello message. The offering node can now calculate a shortened offer: First, it checks if the instance ID matches. If not, the timestamp is worthless and a full state is sent. Otherwise, only the channel states that changed since the given timestamp are offered. Channel states that remained unchanged were already sent to the requesting node during the last offer; had the requesting node wanted any of the messages in them, it would have requested them at that time.

Since the timestamp is calculated at the beginning of the connection, changes made during the connection (and thus possibly after the state was sent) will not be considered included and will be sent during the next connection. The new timestamp is only stored by the requesting node once the connection successfully completes - if the connection fails, the old timestamp (if one exists) is kept. This way, some unnecessary state may be sent, but changes are never lost.

Additionally, the burden of storing the optimization data is now on the requesting node, not the sending node. This prevents denial-of-service attacks that would have been possible in the naive approach by connecting to a node under multiple fake identities and making him store unnecessary states. Since only a constant (and small) amount of additional data is stored per channel (the channel change timestamp) and per connecting node (instance ID and timestamp), only a linear amount of storage is required.

4.5.1.2 Discarding stale channels

Any node that joins the network creates a channel. As soon as one of its messages gets out on the network, sooner or later, it will reach almost all other nodes, and become a permanent part of the network. This increases the size of the saved broadcast state and the size of offers, and thus affects performance. Without countermeasures, the broadcast state would permanently grow until the network becomes unusable. This can become an especially large problem if users delete the application data and restart the application, creating a new node with a new channel. At the same time, very old messages will likely not be of much interest anymore.

For this reason, channels *expire*. There are two kinds of expiration: soft expiration, and hard expiration. The expiration is measured from the date when the local node last received a message from this channel, not from the date where the last message was sent.

A channel that has reached the soft expiration threshold is no longer offered to other nodes. Nevertheless, the channel is still stored in the local broadcast state. If another node provides a message not yet known to the node, the channel is updated. This resets the "last changed" timestamp, making the channel active again. If the other node only offers messages that are already known, the node can see this (since the channel state is still saved) and no messages will be transferred. This preserves bandwidth on the Bluetooth links, but does not alleviate the problem with growing broadcast states that need to be locally saved. (In a real network, these would include the messages themselves, thus creating a significant burden).

To solve the storage problem, a second threshold for hard expiration exists. If a channel reaches this threshold, the node completely deletes it from the local broadcast state. However, if this node now meets another node where the channel has not yet reached the *soft* expiry threshold, the remote node will offer the channel. Since the node no longer has any information about the channel, it treats it as a fresh channel, requests all messages, and stores the channel in the local broadcast state again - with a fresh "last changed" timestamp. This can make hard expiry dangerous, as can be seen in the following example.

Assuming a soft expiry timeout of 7 days and a hard expiry timeout of 15 days, the following can occur:

- at t= 0d node X joins the network, sends a message to node A, and drops out forever
- at t= 6d node B joins the network and receives the message from A
- at t= 7d the channel X reaches the soft expiry threshold on node A, and A stops propagating the messages from the channel
- at t=10d B meets node C and gives C the message
- at t=13d the channel reaches soft expiry on node B, B stops propagating the messages
- at t=15d the channel reaches *hard* expiry on node A. A purges the channel from its state.
- at t=16d C meets A. Since A does not know anything about channel X, C transmits the entire channel X to node A!
- at t=17d the channel reaches soft expiry on node C.
- at t=21d the channel reaches hard expiry on node node B. B purges the channel from its state.
- at t=22d A meets B and transmits the entire channel again

This can lead to an endless circle where messages are passed on after being purged. For this reason, hard expiry is practically disabled in the experiment by setting a very high hard expiration threshold.

In a real network, it would be preferable to expire messages after a certain time has passed since they have been sent. This would require messages to be timestamped and timestamps to be transmitted in the clear. To make traffic analysis harder, timestamps could be limited to the date, without information about the exact time. Due to the lack of a central authority, the time information for timestamping would come from unreliable system clocks. By refusing to accept messages with a date in the future and not accepting/distributing messages older than a certain amount of days, problems arising from false timestamps could be limited. A similar approach (expiry based on sender-specified date, refusing old and future-dated messages) is followed in the Usenet.

4.5.2 Security

The protocol parser is designed to withstand invalid input; protocol violations cause the connection to be closed without influencing the rest of the program. Since the application is written in Java, a managed-memory language, it is nearly impossible that errors in the parser implementation could lead to code execution on the devices. A further security layer, the compartmentalization of apps in the Android operating system, protects participants from issues should the application be caused to misbehave.

Neither the protocol nor the implementation of the experimental network are designed to withstand attacks targeted at sabotaging the experiment. Since the attacker has nothing to gain from such an attack, the probability of such attacks was deemed too low to justify the added overhead of additional security measures. The introduction of cryptography and the required authorization structures would not only require significant additional development effort, but would also cause additional battery drain and error-prone complexity. Thus, security is limited to preventing accidental problems, protecting the participants and their devices, but not aimed at protecting the experiment against all possible attacks.

While this may be acceptable for a small-scale research project, it is absolutely not for any real, productive network. Any real network would need significant additional security measures like signatures in broadcast messages, disconnecting nodes that send protocol messages out of order or provide broadcast messages with false signatures, rate limiting, etc. While the ideas from the protocol may be reused after careful thought, the protocol itself and the protocol-specific code should not be.

One oversight discovered after the end of the experiment is the lack of rate limiting. An attacker could use this to send large amounts of messages, filling the storage of the victim with the resulting logs. This could cause both the data collection application and other apps on the affected device to crash. No attempts to exploit this were observed.

4.6 Implementation

The application consists of two main components, the background service and the user interface.

4.6.1 User interface

The user interface has a simple main Activity (`MainActivity`) which displays a menu and some information. An additional Activity is used to request user consent.

Until user consent is given, data collection cannot be started and any attempt to access the application brings up the consent Activity. The consent Activity displays the privacy policy and requires the user to type “einverstanden” (*agreed*) to confirm it. Once it has been confirmed, a timestamp and the input are saved to a consent file, indicating that consent was given, and the main Activity is launched. A special Extra indicates that consent was just obtained, allowing the main Activity to immediately start the data collection. Early beta versions required the user to explicitly activate data collection after giving consent. Many users did not do this until verbally reminded. This demonstrates how small mistakes in UI design can threaten the success of such field tests.

The main menu shows buttons to start/stop data collection, reconfigure the Bluetooth settings if required (e.g. when Bluetooth is disabled or the name is not set correctly), manually trigger an upload, or visit the web site of the experiment for further information. An additional button is used to display messages from the server (see section 4.6.2.13 for details). Clicking this button opens the URL associated with the

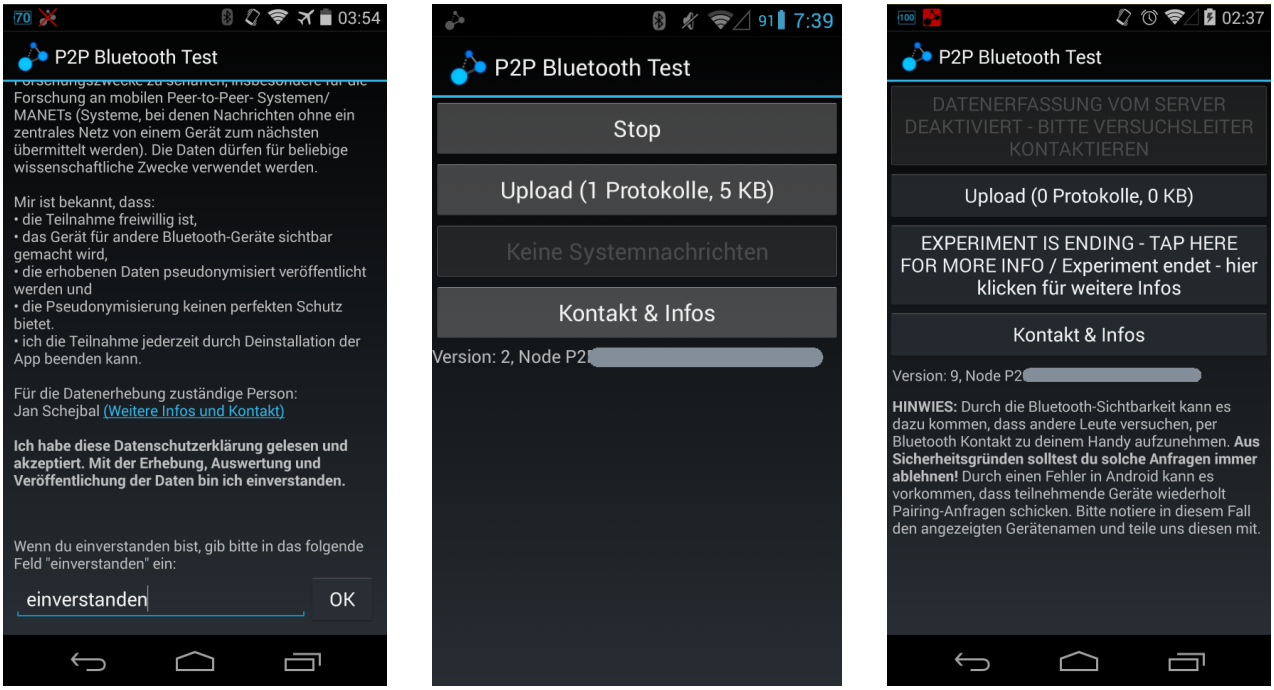

Figure 4.6.: Screenshots of the application UI. Left: The bottom of the consent screen. Center: A very early version with running data collection, missing the note at the bottom. Right: The final version after the end of the experiment, with a triggered kill switch and a server message.

message. If no messages are available, the button is grayed out and displays that no system messages are present.

The application version and the node ID are displayed. This allows easy verification of the installed version, and allows users to report the node ID when e-mailing about issues. If a user chooses to report the ID, the report can be linked to the affected device, providing easy access to Android version information and the log itself for troubleshooting. Additionally, a notice instructing users to be careful about incoming Bluetooth transfers and to report unwanted pairing popups (see section 5.3.1) is included.

By clicking the upload button, the user may opt to upload data using a mobile connection. If the user attempts to trigger an upload without a WiFi connection, a warning about the amount of data being transferred is given and the user is asked to confirm.

A status notification is shown by the background Service. Clicking the notification launches the main Activity, allowing the user to check the application status and control the application. If Bluetooth is not configured correctly (e.g. visibility is disabled or a wrong name is set), the notification will indicate it. The user can then trigger reconfiguration in the main Activity, which will handle the entire process of calling the corresponding system Activity and handling the result (e.g. logging the limited visibility bug described in section 5.3.2).

Events that influence the application status call the `updateStatus()` method of the background Service. The main sources of such events are the data collection manager (when starting and stopping), the Bluetooth manger (on Bluetooth status changes), and the upload thread in the logging manager. The upload thread uses a Handler to invoke methods in the main thread. The `updateStatus()` method updates the notification and, if the main Activity is active, informs it of the status change so the Activity UI can be updated. To update the notification, the Service pulls the state of the relevant parts of the application in `getNotification()`. The Activity pulls the required information through the bound background Service.

Due to privacy and complexity concerns, the application does not display any details about the state of the broadcast network (e.g. connected nodes). This was occasionally criticized by users. Adding such a feature could improve long-term participation and provide an easy way to verify that the application is working correctly. Another possibility to involve the user in the experiment would be to provide a way of actually sending messages through the peer-to-peer network. This was considered, but again not implemented to avoid additional complexity.

4.6.2 Background architecture

4.6.2.1 Background Service

The background Service (`BackgroundService`) is started at the first start of the user interface or at boot time and stays active permanently, even if data collection is stopped. It is responsible for displaying the status notification, keeping the application "alive" (running), and holding instances of the various modules. The Service also serves as the connection between the main Activity and the parts running in the background.

Starting at boot time is done using a Broadcast Receiver (`BootBroadcastReceiver`), which is registered in the manifest. The main Activity binds to the Service using `bindService()`, and the Service starts itself on bind if it is not already running. The Service sets itself up as a foreground service using `startForeground()`, making it less likely to be killed by the operating system if memory is scarce. In addition, it returns that `START_STICKY` flag during startup. This way, it will be automatically restarted by the system if it should ever be killed due to extreme memory pressure. This behavior was verified by exponentially increasing the size of an environment variable using `export a=\$a\$a` in a console application until the system killed all services (including the console application itself). The data collection application was correctly restarted. (Some potential issues that could prevent such a restart are described in section 5.2.2).

The background Service verifies some basic conditions (e.g. it checks for one of the incompatible Android versions, see section 5.3.1) and creates and holds persistent instances of a `LoggingManager` and `DataCollectionManager`. It is also responsible for handling the "kill switch" described in section 4.6.2.13.

4.6.2.2 Logging manager

The logging manager collects all log messages and is responsible for writing them to persistent storage and managing the collected logs. It periodically uploads the logs and checks for new system messages from the server (see section 4.6.2.13). Additionally, it performs NTP lookups for time synchronization.

Each node has a randomly-generated node ID. Internally, the node ID consists of 16 bytes. Only the first half, also known as the *public* node ID, is displayed to the user and used throughout the application. The full node ID is only used by the LoggingManager. It is included in every log and in the requests to the logging server. As such, it represents a shared secret between the node and the server and can be used to detect impersonation attempts.

The log format is described in section 4.4.2. Active log files are stored in one directory, which should normally only contain the current log. If the application is forcibly terminated before a log can be properly closed, partially written logs will remain in that directory.

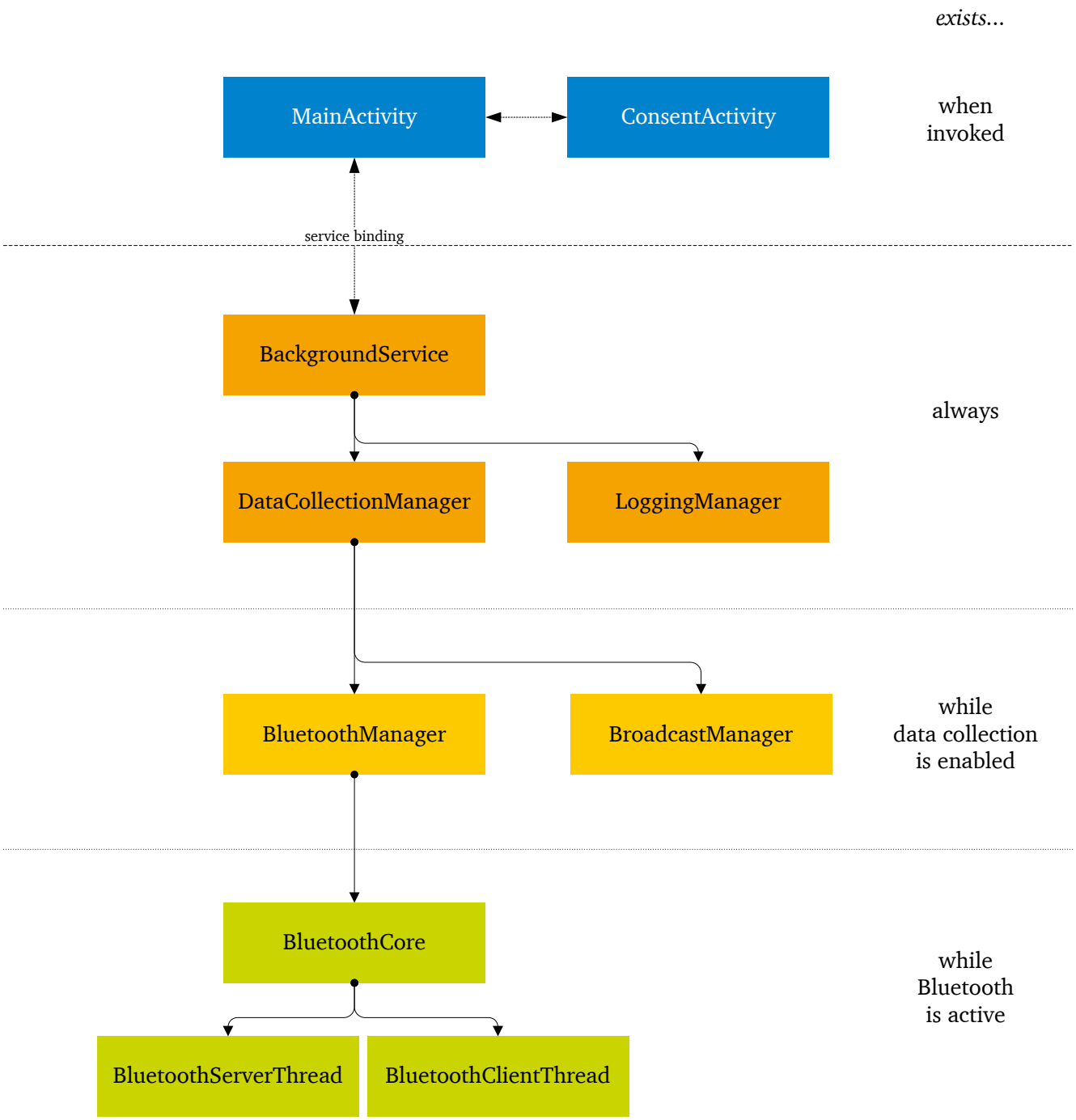

Figure 4.7.: Overview of the software architecture, showing the lifetime of each component

Logs are opened on demand (i.e. when a component attempts to log data) and closed either explicitly by other components (e.g. the background Service closes the log when data collection is stopped), or automatically after a certain predefined amount of time (10 minutes) or when a certain number of entries has been reached. The automatic opening of logs allows components to simply log whatever is needed without having to check the state of the log. Automatic log rotation limits the size of the logs and the possible amount of data lost if logs get corrupted through crashes. The log handler reacts to shutdowns of the device by listening to `ACTION_SHUTDOWN` Broadcasts and closes the current log when a shutdown occurs to prevent data loss. Once a log is closed, it gets moved to a staging directory, and the upload thread is triggered.

The upload thread attempts to upload every file in the staging directory and queries the server for "system messages". Such messages can be provided by the server based on the node ID, the Android version, the software version, or to all devices. Unless the upload was triggered manually by the user (see section 4.6.1), communication only occurs when WiFi is available. See section 4.6.2.13 for details about the messaging system.

During upload and whenever a new log is opened, the logging manager performs an NTP time sync, writing the result to the log. An uptime ID, which fulfills the same purpose as the instance ID introduced in section 4.5.1.1, is held by the logging manager and written to each log. If two logs share the same uptime ID, the elapsedRealtime timer was not reset between the creation of these logs - they *belong to the same uptime*. This can be used to apply e.g. NTP data found in one log to events in another log with the same uptime ID, as it is done in section 5.5.

The upload is performed using a POST request to the server via HTTPS. A self-signed SSL certificate is used for uploads and the application is designed to require an exact match. This way, attackers cannot eavesdrop on or tamper with the connection, and man-in-the-middle attacks are prevented even in the case of compromised certificate authorities.

Logs are deleted from the node as soon as the server has confirmed that the log was successfully uploaded. A HMAC is sent together with the log and verified by the server for integrity checking. This ensures that potential transmission errors (e.g. encoding issues) are detected.

Since every file must have a matching HMAC, a casual attacker cannot easily flood the server with random data to fill up the disk. This is only a slight obstacle, since the HMAC key can be reverse-engineered from the application. An acknowledgement of the fact that this is indeed not secure, together with a request not to sabotage the experiment, is included in the application in a way that potential attackers would see it. Better protection would have required a complex user-registration process with protections against automated registrations. This would have affected participation and was deemed unnecessary.

The logging manager can be configured using compile-time settings to send copies of the logged data via UDP or write them to the Android system log to allow easier debugging. UDP debugging allows the device to be observed without being connected via USB for ADB access. This is important to observe sleeping behavior, which may be different when a power source is connected. However, WiFi activity can also influence this, so the most accurate results can only be obtained by offline collection and later retrieval.

4.6.2.3 Data collection manager

The data collection manager (`DataCollectionManager`) lives in the background Service as a long-lived instance. When data collection is started or stopped, the data collection manager instance persists and receives the start or stop command. The data collection manager, in turn, creates or destroys instances of

the underlying components that only live for the duration of one data collection session: The Bluetooth manager, and the broadcast manager.

4.6.2.4 Bluetooth manager

The instance of the Bluetooth manager (`BluetoothManager`) persists for the duration of one data collection session. During such a session, the ability to communicate may be disrupted e.g. because Bluetooth is disabled. The Bluetooth manager observes and logs the Bluetooth state and creates/starts an instance of BluetoothCore when Bluetooth is enabled, and stops/destroys it when Bluetooth is disabled (or the data collection itself is stopped).

4.6.2.5 Bluetooth core

The Bluetooth core (`BluetoothCore`) operates only if Bluetooth is enabled. It is instantiated and held by the Bluetooth manager. The core coordinates all Bluetooth operations. It holds one instance each of a Bluetooth client (`BluetoothClientThread`) and server (`BluetoothServerThread`) as well as a state machine for coordinating discovery and communication.

Events indicating the start and end of a Bluetooth discovery process are received by the core and passed to the state machine. Events indicating discovered Bluetooth devices (`BluetoothDevice.ACTION_FOUND`) are handled and the devices are logged and added to a found device list as appropriate. Only devices with a name matching the naming pattern (normally a fixed prefix followed by 16 hex characters) are logged to avoid logging non-participants. (This approach turned out to be flawed, see section 5.4.2).

The state machine supports three states - discovery, communication, and waiting - and provides built-in timeout handling for the states.

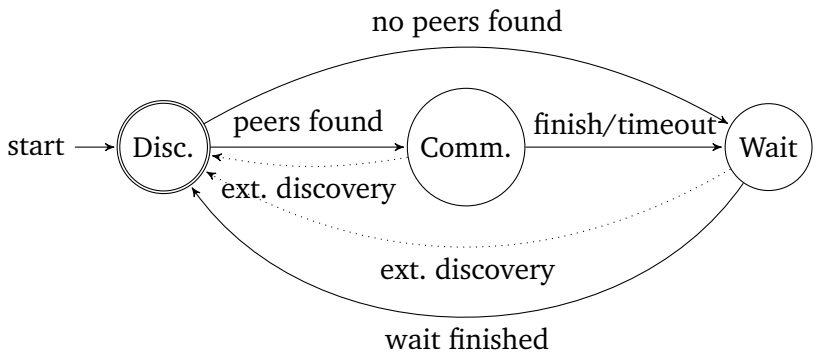

Figure 4.8.: Bluetooth core state chart

Whenever a discovery is started, the “discovery” state is immediately entered. This is important since other software can trigger a discovery. When this state is entered, outgoing connections are immediately terminated. The list of currently discovered devices is cleared, and discovery is triggered if it is not already running (i.e. when the state is entered during the regular state cycle). The state ends either when discovery ends or the state times out (in the latter case, the discovery is cancelled). The list of devices discovered in the current pass (`currentDiscoveryList`) becomes the list of devices discovered during the last discovery cycle (`lastDiscoveryList`), which is used by the later states. If no devices were discovered, the “waiting” state is entered, otherwise, the state machine proceeds to the communication state. The timeout for this state is defined by `CONFIG.DISCOVERY_TIMEOUT`. No explicit wake lock is held during this state.

The communication state is used to create outgoing connections. During this state, a wake lock is held. The Bluetooth client is triggered and attempts to connect to every device in the `lastDiscoveryList`. This state ends when the client reports completion (i.e. one connection attempt was made to each device and the connection either finished or failed) or after a timeout (`CONFIG.COMMUNICATION_TIMEOUT`) has been exceeded. At the end of this state, the wake lock is released and the client is instructed not to create new connections, but ongoing connections are not necessarily killed. The state machine enters the "waiting" state.

The "waiting" state always times out (unless interrupted by an externally-triggered discovery), and is always followed by the "discovery" state. The timeout is calculated dynamically based on `CONFIG.DISCOVERY_INTERVAL`: Discovery is started as soon as the discovery interval has passed, measured *from the end of the last discovery*. The discovery interval is thus the *minimum* interval between the end of one discovery and the next discovery (unless forced by an external application, of course). If the communication phase takes a long time and the timeouts are configured accordingly, the communication phase can be longer than the discovery interval. In this case, the "waiting" state times out immediately and the next discovery starts immediately after the communication phase.

The state machine uses real time, not CPU uptime, for timeouts, using `AlarmManager` via the custom `AlarmHelper` abstraction layer. It should be noted that incoming connections are handled independently from the state machine and are *not* interrupted when discovery starts, but performance will be severely degraded due to the discovery process.

4.6.2.6 Bluetooth server thread

One instance of this class is held by BluetoothCore while data collection is running and Bluetooth is operable. On instantiation, it immediately starts to listen for incoming Bluetooth connections. As soon as a Bluetooth connection is accepted, a ProtocolConnection for handling it is created and the thread continues to listen for new connections. Thus, multiple incoming connections are possible at the same time.

The server thread keeps track of existing connections. A non-reference counted wake lock is obtained when a connection is received. This wake lock is released when all connections are finished. A timeout ensures the wake lock is not held indefinitely in case a connection hangs permanently. The list of connections can be queried to detect if an incoming connection from a certain address exists. This is used by the client thread to avoid outgoing connections to a node while an incoming connection from the same node exists.

4.6.2.7 Bluetooth client thread

The Bluetooth client thread lies dormant until triggered by the Bluetooth core when entering the communication state. It will then attempt to connect once to each of the given devices sequentially in a random order. It does not acquire its own wake lock, since the Bluetooth core already holds one during the communication state.

The class contains methods that will be called on the main thread (e.g. to provide a list of devices to connect to, or to stop communication). The actual thread runs the main loop, which will, in order:

Check if a request to terminate connections has been made, or the current connection is finished or failed. If this is the case, the connection and/or socket are closed (if they still exist), and set the connection and socket variables to null.

If an active connection exists, the thread waits for it to terminate.

Otherwise, it checks if there are any devices to connect to. If not, the Bluetooth core is notified (since all devices have been processed and it should leave the communication state). Otherwise, the thread takes the next device to connect to from the list (possibly waiting for a new set of devices to be provided), checks that it has no current incoming connection, and then proceeds to connect to it.

The check for concurrent incoming connections aims to avoid unnecessary traffic: The protocol determines at the beginning of the connection that certain messages are missing on the other node and thus need to be exchanged, then exchanges them. With two concurrent connections, the messages missing at the start of the second connection would be exchanged over that connection, even though they were already being exchanged over the first one. However, this check is not critical; the program will not malfunction if this check returns an incorrect result due to a race condition. For this reason, no complicated locking was implemented.

As soon as the connection has been established, it is handed off to a ProtocolConnection instance and the thread waits for it to finish.

4.6.2.8 Broadcast manager

The broadcast manager `BroadcastManager` is the core of the experimental broadcast network which is run between the nodes. Like the Bluetooth manager, the broadcast manager instance persists for the duration of one data collection session (irrespective of the Bluetooth state). The broadcast manager holds a `BroadcastState` instance representing the state of the broadcast protocol. This instance is created (loading the state) when the broadcast manager is created, and saved when the broadcast manager is closed or periodically (after the configured number of new messages has been generated).

While active, the broadcast manager generates a new message in the node's channel approximately at the configured interval (using real time via an `AlarmHelper`). Message creation may be delayed if the phone is sleeping.

It also provides `BroadcastProtocolConnections` by acting as a `ConnectionFactory`, and maintains a peer information map for optimized state synchronization. The details of this mechanism are explained in section 4.5.1.1.

4.6.2.9 Broadcast state

The broadcast state (`BroadcastState`) represents the state of all channels as known by the local node. On instantiation, this class reads the broadcast state from a file and stores it to such a file upon request.

The state itself is stored in a map mapping channel IDs to their states. Timestamped channel states are used to support the optimizations described in 4.5.1. The ID used by these optimizations to ensure the validity of relative timestamps in timestamped channel states is also held by this class.

It provides methods for serializing the state to a String (for logging in the `BroadcastState` log entries), querying the presence of channels and messages as well as adding them.

The String serialization of the broadcast state consists of the String serializations of each channel state, each (including the last one) followed by a space.

Additionally, it contains two crucial functions for synchronization of messages between nodes: `getOffer` provides a list of `ChannelStates` that should be sent to another node in the protocol message `MESSAGE_STATE` (see 4.5). `getMessagesToRequest` compares the offer from the remote node to the

local state and generate a list of messages that should be requested (i.e. messages that are present only on the remote node and still of interest to the local node).

4.6.2.10 Channel states (message sets)

A `ChannelState` represents a set of messages from a certain channel. As such, it is used not only to store which messages are present on the local node, but also to announce the available messages to other nodes and to represent the set of messages being requested during synchronization. It contains the ID of the channel (equal to the node id of the node owning the channel), an integer specifying the message ID of the leftmost message in the message bitmap, and a message bitmap with 512 bits.

This means that a `ChannelState` cannot represent a set of messages where the difference between the oldest and newest message ID is larger than 511. Since only the most recent 512 messages of each channel are relevant, this does not pose a problem. Older messages that cannot be represented are automatically pushed out of the channel state when adding new ones.

The `ChannelState` class supports serialization and deserialization to a custom, compact binary format, operations for adding messages and shifting the message bitmap, and a function to determine which messages are present in one channel state but not in another. The latter is used during synchronization and will ignore "old" messages (i.e. ones that could not be represented by the current channel state because their IDs are below the current window covered by the message bitmap).

The binary serialization consists of the 64-bit channel ID, the 32-bit message ID of the oldest message in the bitmap, and the 512-bit bitmap itself. The String serialization is the hexadecimal representation of the binary serialization.

4.6.2.11 Timestamped channel state

A `TimestampedChannelState` is used locally to represent a `ChannelState` with two additional change timestamps that get updated whenever the state is changed.

The first timestamp is used to optimize synchronization. It is based on `SystemClock.elapsedRealtime()` (which is guaranteed to increase monotonically). As such, it loses its meaning when the system is rebooted, and thus it declared with the `transient` keyword, excluding it from serialization. See section 4.5.1.1 for details of the optimization using this timestamp.

The second timestamp is based on the real-time system clock, i.e. an absolute value. It is persistent, i.e. serialized together with the `TimestampedChannelState`, and used by the `BroadcastState` to determine for how long the channel remained unchanged. This allows it to stop offering channels that have not been updated for a certain amount of time, and to completely discard ones that exceed an even further threshold. Details of this are explained in section 4.5.1.2. Since the real-time system clock is used, misconfigured clocks can lead to issues (e.g. accidental purging of channels).

4.6.2.12 Protocol handling

`ProtocolConnection` provides an abstract base class for running a protocol over a Bluetooth socket, supported by the other classes in the protocol.base package. All protocols based on this use TLV *protocol messages*, which consist of two Integers indicating the type and length of the data, and the data itself. Protocol messages must not be confused with the broadcast messages distributed in the experiment's

broadcast network. Like in the protocol section (sec. 4.5), “message” refers to protocol messages, i.e. a TLV bundle sent over Bluetooth, in this section.

As non-blocking I/O is not supported on Bluetooth sockets, two threads per connection are used to handle input and output. The input handling thread of `ProtocolConnection` reads type and length and then calls an implementation-specific handling function. Reading and parsing the message data is left to the individual protocol implementation to allow efficient parsing, however, `ProtocolConnection` verifies that the correct amount was read to ensure programming mistakes are easy to locate. To send a message, the protocol implementation can add it to a message queue. The output handling thread reads this queue, writes type and length of the message, lets the message sends itself, then verifies that the correct length was sent.

This approach avoids having to buffer the entire message. During the first message exchange with a new node, the broadcast protocol may queue thousands of payload messages, each about 1 KB in size. Buffering the messages would unnecessarily waste memory. Using this approach, the message contents are generated on-the-fly as messages are sent. In a real network, the messages would be read from a message store at send time. (In the experiment network, message bodies are randomly generated).

If either of the threads encounters an exception (e.g. I/O exception, protocol violation), the connection is terminated. Terminating the connection as soon as one stream (direction) is closed or broken may seem unnecessary. Unfortunately, the behavior of the Bluetooth socket makes it impossible to handle it differently. In fact, closing the OutputStream of a Bluetooth socket on Android immediately terminates the entire Bluetooth connection, even dropping data already written to that stream.

This behavior can easily cause confusion, as it is different from the behavior of TCP sockets. TCP sockets will deliver all written data, even if the OutputStream or even the socket itself are closed. (Both behaviors were verified experimentally on Android). Protocol implementations must ensure that they do not close any stream of the Bluetooth socket or the socket itself before they are done with the socket. (See section 4.5 for an explanation how this is handled in the protocol).

A simple protocol used for early measurements and to debug the protocol handling mechanism is provided in the protocol.simple package. It is not used by the application.

The actual protocol used by the experiment’s broadcast network is implemented by the protocol.broadcast package. `ProtocolBroadcastConnection` extends the abstract `ProtocolConnection` class and provides the protocol. The remaining classes in the package extend `ProtocolMessage` and represent the different types of messages. The protocol itself is described in detail in section 4.5.

4.6.2.13 Server messaging system

The upload thread periodically queries the logging server for messages (using HTTPS like for the log upload). The requests include the full node ID, application version and Android version. This allows the server to deliver system messages to individual nodes or to all nodes running a certain version, e.g. to notify them of a required update.

System messages consist of a text that is displayed to the user, and an URL that is opened when the user clicks the message. Users get notified of new messages by the application. These system messages are not related in any way to protocol messages or broadcast messages.

A “kill switch” that disables the communications part of the application is also implemented using the messaging system. Specially prefixed messages will cause the node that receives this message to change the Bluetooth name and stop data collection. Renaming the device causes it to be treated as a non-participant by other nodes, preventing incoming connections, and stopping data collection disabled

discovery and outgoing connections. While intended to be used in case of misbehaving nodes that are disrupting the network (e.g. by causing pairing popups), it was only used towards the end of the experiment to protect nodes from a privacy issue discovered during the experiment (see section 5.4.2).

The kill message does not affect any other part of the phone's operation and can only be sent via the secure channel from the server.

4.6.2.14 NTP

An Android SNTP[4] client is part of the Android OS, but not available to applications. Thus, a copy of the source file is included in the application.

When the screen is off, the WiFi radio is set to a low-power mode. To maximize accuracy, the WiFi adapter is explicitly locked into a high-performance mode for the duration of the NTP lookup. While this lock mode (`WIFI_MODE_FULL_HIGH_PERF`) is officially supported only on API level 12 and newer, it was introduced in Android 2.3 as a hidden feature [5]. It may not be supported by all devices, but provides significant improvement on devices that do support it. On a Galaxy Nexus, a sample of 34 NTP requests without a WiFi lock had a median round-trip time of 302 milliseconds. Introducing a WiFi lock reduced this to 40 milliseconds (with a sample size of 30 requests).

4.6.2.15 Configuration

Configuration variables that affect the behavior of the application are defined (and documented) in `CONFIG.java`. Important values that affect the performance of the experiment and/or were changed between versions are explained in table 4.1.

		Value in version...		
Name	Function	before 8	8	9
MAX_LOG_TIME	Close a log after at most x minutes	10	10	10
MAX_LOG_LINES	Close a log after at most x log lines	10000	10000	10000
UPLOAD_WARN_AGE	Warn if logs have been waiting for upload for longer than x hours	24	**48**	48
MAX_UPLOAD_PAUSE	Retry upload (and message fetching and NTP) at least every x minutes	5	**30**	30
CLIENT_CONNECTION_TIMEOUT	Abort a Bluetooth connection attempt if it takes more than x seconds	15	15	15
DISCOVERY_TIMEOUT	Stop discovery if it takes longer than x seconds	30	30	30
DISCOVERY_INTERVAL	Wait at least x seconds between discovery phases	60	60	**120**
COMMUNICATION_TIMEOUT	Stop the communication phase after at most x seconds	90	90	90
BATTERY_LOG_INTERVAL	Log the battery state every x minutes	5	5	5
MAX_MESSAGE_LENGTH	Maximum size of a protocol message in byte	1M	1M	1M
BROADCAST_TIMEOUT	Abort a connection if no protocol message is received for x seconds	20	**10**	10
BROADCAST_PAYLOAD_SIZE	Size of the broadcast message payloads	1024	1024	1024
BROADCAST_MESSAGE_CREATION_INTERVAL	Create a new broadcast message every x minutes	3	**10**	10
BROADCAST_SAVE_EVERY	Save the broadcast state every x created broadcast messages	5	5	5
BROADCAST_CHANNEL_EXPIRY_SOFT	Soft expiry timeout in days (see 4.5.1.2)	7	7	7
BROADCAST_CHANNEL_EXPIRY_HARD	Hard expiry timeout in days (see 4.5.1.2)	180	180	180
BROADCAST_MAX_CHANNELS	Limit for the number of channels maintained (DoS protection)	—	—	**10000**

Table 4.1.: CONFIG.java configuration options

[4] Simple Network Time Protocol
[5] Android Issue 9781 - `http://code.google.com/p/android/issues/detail?id=9781`

4.6.3 Permissions

The Android permission system restricts what functionality and data is available to an application. Access to privacy-sensitive or otherwise critical functions is restricted to applications that have the corresponding permission. Permissions are declared in the application manifest. During installation, the user is shown the permissions required by the application and asked to confirm this. Requesting too broad permissions can scare users off and make them cancel the installation process.

By only requesting the necessary permissions, the application implicitly proves to the user that it doesn't access certain types of data it should not access. For example, the user can be sure that the application does not collect location data, since the corresponding permissions (e.g. `ACCESS_FINE_LOCATION`) are not requested. Consequently, the Android operating system would not allow the application to access location data even if the application attempted to do so. The privacy policy, which is more detailed, only provides the user with *claims* what the app is and is not collecting. The Android permission system helps build trust by giving the user a quick summary of what data can and cannot be collected.

The application attempts to use as little permissions as possible. The following permissions are required:

- **`BLUETOOTH`**: Required for most Bluetooth operations
- **`BLUETOOTH_ADMIN`**: Required to trigger discovery and change the Bluetooth device name
- **`INTERNET`**: Required to connect to the log upload server for log uploads and message checking and for NTP time synchronization
- **`WAKE_LOCK`**: Required to configure wake locks (see section 4.6.5)
- **`ACCESS_NETWORK_STATE`**: Allows the application to detect if WiFi is available to avoid uploading logs via mobile networks, which could create costs for the user
- **`RECEIVE_BOOT_COMPLETED`**: Allows the application to start automatically when the phone is rebooted

The low number of permissions and the lack of permissions considered invasive or critical to privacy was positively noted by several participants during installation. This helped build trust, making it easier to convince potential volunteers to participate.

`BLUETOOTH_ADMIN` would also allow the application to enable Bluetooth directly. However, it does not allow to enable discoverability programmatically and enabling Bluetooth that way is discouraged in the documentation [27, BluetoothAdapter]. Therefore, the recommended `ACTION_REQUEST_ENABLE` Intent is used instead, which requests confirmation from the user.

`ACCESS_WIFI_STATE`, which would allow reading WiFi network names, is explicitly not requested. The same goes for `BATTERY_STATS`. This would allow to collect detailed information about battery usage. Due to the potential privacy implications (as this would provide information about other applications, device usage patterns etc.) this is not used. Starting with Android 4.4, this permission is no longer granted[6] to regular (non-system) applications and can thus no longer be used in practice.

[6] https://android.googlesource.com/platform/frameworks/base/+/android-4.4_r1/core/res/AndroidManifest.xml

4.6.4 Randomness

As this is a real, physical experiment, large amounts of entropy are introduced through external events (e.g. noise interfering with connections, movement of nodes), making artificial randomness much less important than in simulations.

Nevertheless, the `SecureRandom` class (a cryptographically strong PRNG) is used as a matter of principle where possible, and protections against Android PRNG weaknesses are included. Unfortunately, the code for randomizing the connection order used the default PRNG for the largest part of the experiment. This is not believed to be a problem, since the order by which devices are discovered changes and the existing PRNG still provides reasonable randomness. Nevertheless, it is documented here for completeness and was fixed in version 9, which is the version deployed on the 30C3.

4.6.5 Wake locks

The Android OS automatically puts the CPU into a low-power state if the screen is off unless it is needed for a task. This can disrupt background activities like data transfers. For this reason, the CPU can be forced to stay active using so-called wake locks. This comes at the expense of battery power, so it should be avoided unless necessary. Unfortunately, the Android Bluetooth documentation provides little guidance when these wake locks are actually required. While testing showed it to be mostly unnecessary on the tested Galaxy Nexus device, it is possible that other devices behave differently. The additional drain caused by using wake locks seems acceptable (see section 6.6.1).

To be on the safe side, wake locks are used during the active communication phases, but not during discovery. This is based on the assumption that the CPU will likely have to be active most of the time during the communication phases anyways. For outgoing connections, the Bluetooth core holds the wake locks, while the Bluetooth server thread holds them for incoming connections. Details can be found in the corresponding sections 4.6.2.5 and 4.6.2.6.

4.6.6 Source code

To avoid the lack of transparency criticized in other publications (see section 2.4), the source code corresponding to each released version was checked into a Git repository as part of the release checklist.

The source code of all versions of the application that generated any of the included log data is published together with this thesis to allow verification of the results.

The code can be used to help understand the logs and discover and mitigate potential issues with the data. However, **under no circumstances should this code or the protocol be reused for real networks**.

The code and protocol are designed and written for the purpose of this scientific experiment only, where no adversarial threat is to be expected. They are neither designed nor suitable for use in real networks, and they are neither designed to be portable nor reusable. It is not recommended to attempt to use any of this code for any purpose other than verification of this experiment and the resulting data. Attempting to reuse it for other purposes is likely to cause security issues - which in the context of censorship-resistant networks can directly translate to the deaths of activists.

The only code where even the remote possibility of reuse was considered is the protocol base package de.janschejbal.p2pbluetoothtest.protocol.base, which handles the encoding and decoding of protocol messages and allows bidirectional, asynchronous communication over Bluetooth. If no better libraries

and APIs are available, reusing parts of this code for non-critical applications could be considered after thorough review.

The code is provided exactly as-is except for the redaction of a cryptographic key. Since any “cleanup” could mistakenly alter the code and would thus make it less useful for verification purposes, no cleanup was performed. Thus, some outdated comments and “TODO”-markings may be present.

The author firmly believes in academic transparency and open communication, and has provided this code and these warnings for the benefit of others. In exchange, the author kindly requests that neither the quality of the code nor the presence of these warnings is held against him.

4.7 Log analysis

The collected logs were pseudonymized (see section 4.9) and then analyzed.

4.7.1 Custom analysis

Initially, custom Java code was developed to scan and analyze the logs.

This code also performs consistency checks on the data to ensure correct operation of the software. For example, using the log entries about received and created messages, the expected BroadcastStates are calculated and compared to the BroadcastStates included in the logs. “Impossible” situations, like a node relaying a message it does not have, are detected. Inconsistencies can occur, mostly due to lost logs. Found inconsistencies were analyzed manually to ensure they were not the result of data-invalidating bugs.

Some data extracts about NTP timing and node presence generated by this tool are used for the charts in chapter 6. Additionally, this code generates an intermediary event log for consumption by the visualization tool described in section 4.7.3.

4.7.2 Log database

As a scan of the logs can take hours, this approach quickly proved infeasible. The pseudonymized logs were therefore imported into an SQLite database. While SQL queries are less powerful than custom code, they are much faster to develop, and sufficient for most tasks.

For example, the following query reveals that 4.9M successful broadcast message transfers were logged, with a query time of less than 7 minutes:

```
SELECT COUNT(*) FROM log_readable WHERE c1='Received' AND c2='PAYLOAD'
```

One large table is used to store all log data, each row containing one line from a log. Each row indicates its origin by naming the reporting device (i.e. log directory), the name of the log file, and the 1-based line number. The two timestamps from the log are stored as integer values. The remaining columns from the original log line are stored in individual columns as references to a string table to preserve space. A view that recreates the human-readable logs by repeatedly JOINing the log table with the string table is provided.

The SQLite database has an uncompressed size of 4574 MB. Applying NTFS file compression keeps the database accessible, but reduces the on-disk size to 1739 MB. Compressing the database into a solid 7z archive with aggressive compression settings reduces the size to 407 MB. However, the database cannot be used directly in 7z compressed format.

```
CREATE TABLE logs (device INTEGER, log INTEGER, line INTEGER, wallclock INTEGER, uptime INTEGER,
                   tag INTEGER, c0 INTEGER, c1 INTEGER, c2 INTEGER, c3 INTEGER, c4 INTEGER);

CREATE TABLE stringmap (s TEXT);
```

Figure 4.9.: Schema of the log database. See section 4.4.3 for the meaning of *wallclock* and *uptime*.

For comparison, the original pseudonymized logs require 771 MB[7] in the format of choice, a single ZIP file containing all logs. Storing the logs in one large ZIP archive allows compression of the logs while keeping them accessible, and avoids flooding the file system with small files. Additionally, most programming languages provide a reasonably easy way of accessing files inside a ZIP archive.

For the purpose of log analysis, SQLite is just as suitable as other database engines. Despite common misconceptions, SQLite can perfectly handle the required gigabyte-sized databases. The limitations of SQLite (e.g. limited concurrency and scalability) are not relevant for this use case. While other database engines (e.g. PostgreSQL) provide more functionality, SQLite requires almost no set up.

In appendix A, the power of SQL-based analysis is shown in detail.

4.7.3 Visualization tool

A visualization tool displaying a graph of the network was developed to aid debugging, using HTML5, JavaScript, and the JavaScript InfoVis Toolkit.[8] Using the extract of the log data provided by the custom analysis tool, this tool displays which messages are present on which node and indicates Bluetooth discoveries, connections, and message transfers on the graph. A timeline allows navigating through the data, and a time lapse showing the evolution of the network can be shown. The graph can be re-arranged manually

In the initial phase, this tool proved invaluable in understanding the network as well as finding and debugging potential issues. Unfortunately, it quickly became a victim of the experiment's success: The network and the amount of data collected is too large to meaningfully visualize, and would require an unrealistic amount of resources to render using this technology. For this reason, development of the tool was abandoned.

4.8 Recruitment of volunteers

For initial beta testing where short-time participation was sufficient, volunteers were recruited via personal contact at a Pirate Party meeting in Frankfurt and the local hacker space in Darmstadt.

While flyers were posted at university, this approach did not lead to any significant number of signups. Attempts to recruit volunteers by presenting the project during lectures were also mostly unsuccessful. Personal contact in the student computer room, however, proved highly successful. Surprisingly, most people who had a compatible phone were willing to participate. Consequently, this was the main way of recruiting volunteers for the production phase.

As with any other recruitment method, this can lead to selection bias. This needs to be taken into account by users of the collected data.

Recruiting via personal contact also made it possible to give some additional instructions, ensure correct activation, troubleshoot minor issues, and observe problematic behavior on some devices. This made the

[7] SI units, i.e. 1 KB = 1000 byte
[8] `http://thejit.org/`

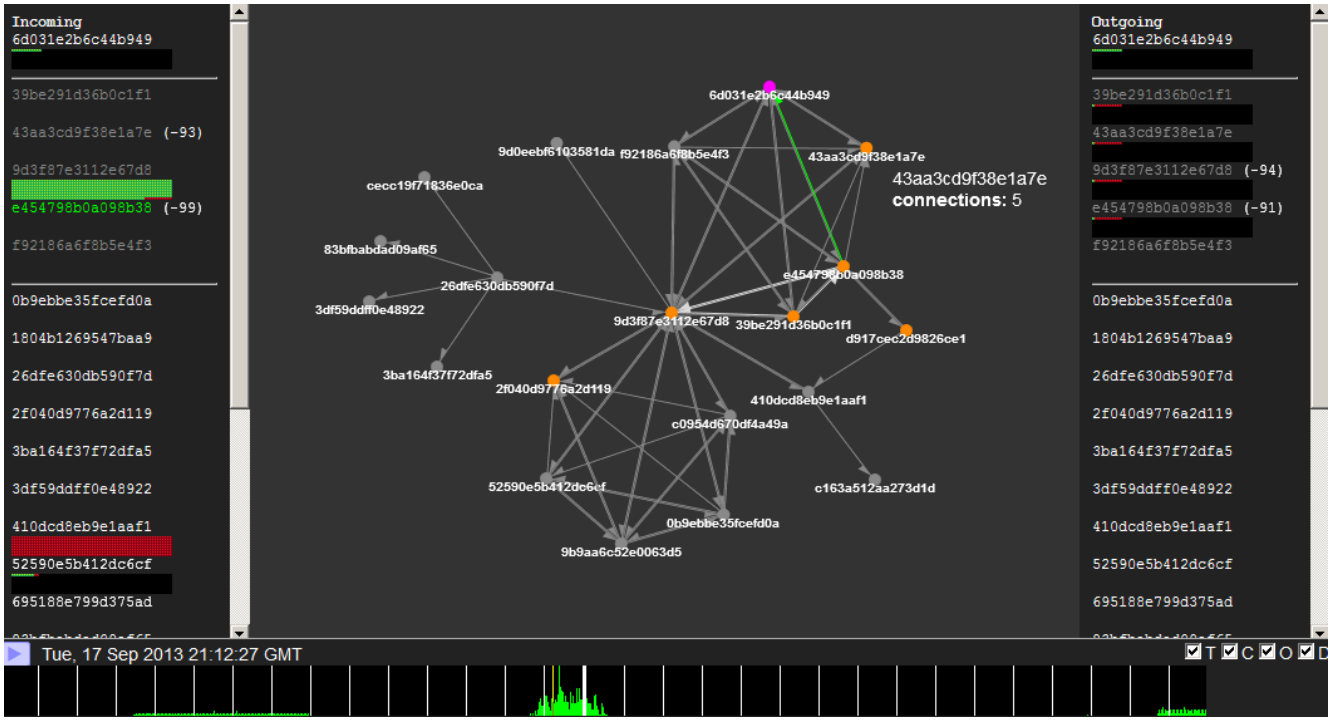

Figure 4.10.: The visualization tool displaying a small subset of the data. The timeline can be seen at the bottom. The left sidebar shows incoming discoveries and connections to the selected node using the color of the node IDs. The maps below each node ID indicate which messages from that node are present on the selected node. The right sidebar shows which devices the selected node has discovered and outgoing connections. Here, the maps below each node represent which messages from the selected node are present on each of the remote nodes. In case of errors, mouseover hints provide the exception messages.

observations documented in section 5.3.2 possible. Participants were asked to send an e-mail if any issues arose or if they decided to stop participating for any reason. Without information from a participant, it is impossible to determine if a node stopped participating due to the choice of the participant, a unreported crash of the application, or some other reason. Unless the node is seen and successfully connected to by others, it is also impossible to distinguish a node that stopped participating from a node that is participating and working correctly, but cannot upload logs (e.g. because no WiFi connection is available).

4.8.1 30th Chaos Communication Congress

The author held a talk at the 30th Chaos Communication Congress and mentioned this project at the end, asking for volunteers. Additionally, flyers were printed and distributed throughout the congress center, and volunteers were recruited by personal contact. This proved highly successful, leading to a second wave of participants.

For consistency, the author suggests the following definitions to be used when splitting the data set:

Consider all participants who were first seen after 2013-12-27T00:00Z and before 2013-12-31T00:00Z as belonging to the *30C3 group*.6.2 For this purpose, “first seen” shall be defined as the smallest “wallclock” time reported by the device, ignoring bogus timestamps before 2013-01-01 (i.e. a device reporting wallclock timestamps of 1999-01-01 and 2013-12-30 was “first seen” on 2013-12-30). This results in a list of 191 devices. A SQL query providing this list is given in figure 4.11.

```
SELECT * FROM (
    SELECT device, MIN(wallclock) AS first_seen_wallclock, DATETIME(MIN(wallclock)/1000,'unixepoch') AS readable
    FROM log_readable WHERE DATETIME(wallclock/1000,'unixepoch') > '2013-01-01'
    GROUP BY device ORDER BY first_seen_wallclock
) WHERE readable > '2013-12-27 00:00' AND readable < '2013-12-31 00:00';
```

Figure 4.11.: SQL query to select the 30C3 group

Some devices from Darmstadt were likely present at the 30C3, as well as some devices that did not upload logs. By the nature of the experiment, it is not possible to determine the physical location of two meeting devices. A 30C3-device meeting another device could have been carried from the Congress to any other location, including Darmstadt. For simplicity, the author proposes that in addition to the devices belonging to the 30C3 group, any device that saw or was seen by a device from the 30C3 group should be considered to have been *at the Congress* no matter when this contact happened.

These simple definitions were chosen to ensure consistency between works using this data set. Timing correction was intentionally not applied to avoid ambiguity (e.g. which NTP responses to consider bogus). While these definitions are not perfect, they are a reasonable approximation that can be easily applied to the data.

4.9 Privacy

To respect the privacy of the participating volunteers, only limited data is collected. Notably, the following data is *not* collected:

- Location data
- Identifiers of visible or connected WiFi access points
- Unique hardware identifiers (IMEI, Bluetooth addresses)
- Information about other applications
- Information about device activity (e.g. if the screen is on)

All nodes are only identified using random pseudonyms. Since these pseudonyms are visible to users, they are mapped to new pseudonyms for the released data set.

Pseudonymization is performed using a custom Java program that reads and parses all logs, HMAC-hashing node IDs with a secret key. This creates a mapping that is repeatable using the secret key, but cannot be repeated without knowledge of the key. Since node IDs appear in the logs in different forms, this is not completely trivial. IDs are normalized by removal of the prefix and truncating to a fixed length prior to pseudonymization. This ensures that the same node ID is always normalized to the same value, independent of the form in which it originally appears. To avoid leaking information about the original node IDs, lists (e.g. in serialized `BroadcastStates`) are sorted according to the pseudonymized IDs, and the log directories are processed in such order.

There is a theoretical possibility that users could be re-identified. For example, knowledge that a user's device was in a crowded lecture hall with other participants at certain times, and powered off at others, could allow identifying this device in the publicly released logs. The data could then be used to find a limited amount of information about the user (e.g. times when the device was charging) that could be used to make assumptions about his location or activity.

A privacy policy/consent form (shown in appendix B) is shown to the user when the application is first installed. During recruiting, a verbal summary (and if necessary, translation) of the privacy policy

was provided. Users have to express consent by typing the word “einverstanden” (“agreed”) before data collection can be started. The privacy policy informs users not only about the data being collected, but also about the risk of re-identification, the fact that their device will be visible, upload and NTP polling behavior, and the purpose of the data collection. It also informs users that uninstalling the application will stop data collection. While this prevents the device from collecting data, it may remain visible and thus data about the device can be collected by other devices. This can lead to unintentional data collection. How this issue was handled after discovery is described in section 5.4.2.

The upload server is configured to avoid IP logging as far as possible (however, see section 5.4.1 for accidental overcollection).

One downside of not collecting any personal information (e.g. names or e-mail addresses) of users is that users cannot be contacted after they uninstall the application.

Do not attempt to re-identify users when using the data set!

5 Challenges

5.1 Lessons learned

This section provides advice for future data collection projects.

5.1.1 Unexpected behavior

Unexpected behavior has to be expected, especially when running on a large, heterogeneous set of devices and using unreliable wireless communications. A small insight into the issues observed in this experiment is given in section 5.2.

5.1.2 Logging

The log format should be carefully designed to allow easy analysis. Using protocol buffers[1] would make parsing significantly easier and allow storing hierarchical data hierarchically.

Globally unique IDs should be used wherever possible, even if not strictly needed. For example, having globally unique connection IDs would have significantly simplified connection analysis. All relevant data is in the logs, but it requires significant effort to extract it (by matching related log lines) because connection IDs may be reused.

5.1.3 Timing

Having accurate timing information is critical for many purposes. For this reason, NTP synchronization should be performed aggressively, if needed even using mobile networks. (Consent required!) The software should track time independently from the system clock, and log the accuracy of the real-time estimate. Exchanging timing information between nodes should also be strongly considered. Attempting to achieve sub-second accuracy for all nodes is futile - instead, effort should be put into *reliably* having accuracy within a few tens of seconds, and ensuring the potential for deviation is known.

A robust NTP client should be used. Robustness can be achieved e.g. by validating the Originate Timestamp field in the response against the request and ingoring (but logging) obviously bogus responses.

Of course, great care should be taken that the time on any servers involved in the experiment is accurate.

5.1.4 Consent and opt-out

Asking participants who wish to opt out of the data collection to explicitly declare so (e.g. by sending an e-mail) instead of just uninstalling the application would have provided precise information about opt-out times and avoided or simplified most of the clean-up effort described in section 5.4.2. Uninstallations should also be logged where possible (however, this is difficult to implement in practice).

[1] `https://code.google.com/p/protobuf/`

Researchers are often more aware of potential privacy risks of collected data, and unfortunately, most users do not read consent forms and privacy policies. Thus, even if participants are willing to provide consent for the collection of certain data, the researcher must exercise caution and avoid overly invasive collection to protect the privacy of the participants. The consent requested from the participants should extend beyond the necessary minimum by a reasonable amount. This allows minor adjustments of the scope and provides protection in cases of unintentional overcollection. The risk of unintentional overcollection should be disclosed in the consent form, including the risk that such overcollection may not be noticed in time and the data could end up in the published data set. This must not be taken as a license to be careless - it is, however, a real risk that should not be hidden from participants.

Participants should be informed in the consent form that data may be processed using third party services. Without this, it is difficult to determine which resources can be used to analyze the data: Is it appropriate to rent a powerful dedicated server and transfer and analyze the data there? Is it appropriate to do the same with a virtual machine? Or using a cloud service like BigQuery[2]? The plan to do all analysis on the fully pseudonymized, published logs became impossible due to overcollection, requiring large parts of the analysis to be performed on the author's computer.

It is always better (within reason) to request more consent without using it, than requesting insufficient consent. Especially if participants are anonymous like in this case, it is impossible to request broader consent later!

5.2 Operating system issues

5.2.1 HTTP keepalive problems

HTTP/HTTPS connections can be kept across requests to avoid delays from the TCP and SSL handshakes. This behavior is called *HTTP keep-alive*. After a certain timeout is reached, these connections are closed and can no longer be used. The client then needs to create a new connection for subsequent requests.

Android provides two common mechanisms for making HTTP requests, the deprecated AndroidHttpClient and the newer HttpUrlConnection. Handling of connections with keep-alive is broken in AndroidHttpClient, and the suggested solution is to use HttpUrlConnection since AndroidHttpClient is deprecated no longer under active development [3].

Unfortunately, the same problem also occurs with HttpsUrlConnection and causes log uploads to fail. The client keeps the connection used to upload a log in a connection pool. After the configured keep-alive timeout, the server closes the connection, sending a TLS Close Notify shutdown alert, followed by a FIN packet. The Android system acknowledges the FIN packet, but does not close its side of the connection and keeps the stale connection in the pool. During the next log upload, this half-closed connection is reused, causing the server to respond with an RST packet. Consequently, the HTTP request fails, but the stale connection is finally removed from the pool. Thus, the next request will succeed again. This can be seen in figure 5.1.

To avoid this issue, keep-alive can be disabled either on the server side (e.g. using `keepalive_timeout 0;` in nginx), or on the client side by using `System.setProperty("http.keepAlive", "false");`. Explicitly setting a keep-alive timeout header in the response (e.g. using `keepalive_timeout 95 30;` in nginx) does not resolve the issue.

Disabling keep-alive increases overhead, as every single connection needs to perform a full TCP three-way handshake, followed by an SSL handshake. In this case, this solution was chosen because uploads

[2] `https://developers.google.com/bigquery/`
[3] Android Issue 42199 - `http://code.google.com/p/android/issues/detail?id=42199`

	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.50.207	130.83.239.76	TCP	37843 > https [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=2626236 TSecr=0 WS=64
2	0.019178	130.83.239.76	192.168.50.207	TCP	https > 37843 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1380 SACK_PERM=1 TSval=406286861 TSecr=2626
3	0.021102	192.168.50.207	130.83.239.76	TCP	37843 > https [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=2626239 TSecr=406286861
4	0.024822	192.168.50.207	130.83.239.76	TLSv1	Client Hello
5	0.044025	130.83.239.76	192.168.50.207	TCP	https > 37843 [ACK] Seq=1 Ack=446 Win=15552 Len=0 TSval=406286867 TSecr=2626239
6	0.045086	130.83.239.76	192.168.50.207	TLSv1	Server Hello, Certificate, Server Hello Done
7	0.049879	192.168.50.207	130.83.239.76	TCP	37843 > https [ACK] Seq=446 Ack=898 Win=17536 Len=0 TSval=2626243 TSecr=406286867
8	0.058036	192.168.50.207	130.83.239.76	TLSv1	Client Key Exchange, Change Cipher Spec, Finished
9	0.082422	130.83.239.76	192.168.50.207	TLSv1	New Session Ticket, Change Cipher Spec, Finished
10	0.087958	192.168.50.207	130.83.239.76	HTTP	GET /msg.php?node=P2P-[redacted]&version=8&android=4.3 HTTP/1.1
11	0.107886	130.83.239.76	192.168.50.207	HTTP	HTTP/1.1 200 OK (text/html)
12	0.143461	192.168.50.207	130.83.239.76	TCP	37843 > https [ACK] Seq=1022 Ack=1330 Win=21120 Len=0 TSval=2626255 TSecr=406286883
13	65.254808	130.83.239.76	192.168.50.207	TLSv1	Alert (Level: Warning, Description: Close Notify)
14	65.255145	130.83.239.76	192.168.50.207	TCP	https > 37843 [FIN, ACK] Seq=1353 Ack=1022 Win=17728 Len=0 TSval=406303143 TSecr=2626255
15	65.421294	192.168.50.207	130.83.239.76	TCP	37843 > https [ACK] Seq=1022 Ack=1353 Win=21120 Len=0 TSval=2626965 TSecr=406303143
16	65.458971	192.168.50.207	130.83.239.76	TCP	37843 > https [ACK] Seq=1022 Ack=1354 Win=21120 Len=0 TSval=2626970 TSecr=406303143
17	634.945082	192.168.50.207	130.83.239.76	TCP	[TCP segment of a reassembled PDU]
18	634.946018	192.168.50.207	130.83.239.76	TLSv1	[SSL segment of a reassembled PDU]
19	634.946151	192.168.50.207	130.83.239.76	TLSv1	POST /upload.php?node=P2P-[redacted]&hash=[redacted]
20	634.966300	130.83.239.76	192.168.50.207	TCP	https > 37843 [RST] Seq=1354 Win=0 Len=0
21	634.967065	130.83.239.76	192.168.50.207	TCP	https > 37843 [RST, ACK] Seq=1354 Ack=2893 Win=21120 Len=0
22	634.967453	130.83.239.76	192.168.50.207	TCP	https > 37843 [RST, ACK] Seq=1354 Ack=2977 Win=21120 Len=0
23	634.970892	192.168.50.207	130.83.239.76	TCP	35686 > https [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=2631117 TSecr=0 WS=64
24	634.989820	130.83.239.76	192.168.50.207	TCP	https > 35686 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1380 SACK_PERM=1 TSval=406445604 TSecr=26311
25	634.991414	192.168.50.207	130.83.239.76	TCP	35686 > https [ACK] Seq=1 Ack=1 Win=14656 Len=0 TSval=2631120 TSecr=406445604
26	634.996661	192.168.50.207	130.83.239.76	TLSv1	Client Hello

Figure 5.1.: Wireshark dump showing the keepalive issue. The contents of the SSL connection were decrypted using the server's private key.

are sufficiently rare for this not to be a problem, and the solution could be deployed without changes to the already-deployed client software. Using an alternative HTTP library for the request could avoid the issue without the need to disable keep-alive.

The issue has been reported to and confirmed by the Android Open Source project [4].

5.2.2 Service restart issues

A Service that returns `START_STICKY` should be automatically restarted by the system if it is killed e.g. due to memory pressure. A fellow researcher, Jan Hendrik Burdinski, informed me about two flaws in restart behavior.

He pointed me to an issue where services are not properly restarted after such a kill. On older Android versions, the `onStartCommand()` function is not called when this happens, which will often render the service unusable. He suggested using a workaround where initialization is started using a `MessageHandler` with a `WeakReference` to the service, triggered by a delayed message sent from `onCreate()`. [5] Since no devices with affected Android versions were available for testing at that time, this workaround was not implemented.

This issue [6] was fixed in commit 3a28f22eb65c5734 on the 1st of March 2011 and thus only affects older devices. According to the repository history, the fix was included in 3.2.4. Information about other 3.x version was not available in the repository, though from the commit date and the release dates, it seems likely that 3.0 was affected while 3.1 was not. (It should be noted that 3.x devices are extremely rare, representing around 0.1%[7] of the devices active in the Play store).

Additionally, he advised against using the `START_REDELIVER_INTENT` flag, as this may include an exponential backoff that can prevent the service from restarting in a timely manner.

[4] Android Issue 61706 - `http://code.google.com/p/android/issues/detail?id=61706`
[5] `http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html`
[6] Android Issue 14340 - `http://code.google.com/p/android/issues/detail?id=14340`
[7] `https://developer.android.com/about/dashboards/index.html`

5.2.3 Interference from SE Android

The application includes a workaround[8] for PRNG security issues that were present in some Android versions. This workaround writes to `/dev/urandom` to add a seed to the system PRNG.

On a Samsung Galaxy S4 with Android 4.2.2, this caused the application to be terminated by SE for Android, a recently enabled SELinux-based security measure. (SE Android is present on some other devices, but not set to "enforcing" mode there and/or supplied with policies that do not ban writing to `/dev/urandom`).

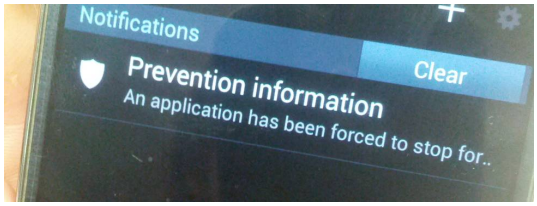

Figure 5.2.: Notification indicating that the application was terminated by SELinux

The current[9] version of the workaround contains a change that is supposed to make it compatible with devices where is not writeable, but it is unclear if this will prevent the issue described above (since an attempt to write to `/dev/urandom` will still be made, thus possibly triggering SE Android).

5.2.4 Crash reports

Four crash reports were received during the Congress. Three came from Samsung Galaxy devices running Android 4.1, indicating null pointer exceptions in the updateState method.

```
java.lang.RuntimeException: Error receiving broadcast Intent { act=android.bluetooth.adapter.action.SCAN_MODE_CHANGED
        ↪flg=0x8000010 (has extras) } in de.janschejbal.p2pbluetoothtest.bluetooth.BluetoothManager@42a9bbd8
[...]
Caused by: java.lang.NullPointerException
at de.janschejbal.p2pbluetoothtest.bluetooth.BluetoothManager.updateState(BluetoothManager.java:61)
at de.janschejbal.p2pbluetoothtest.bluetooth.BluetoothManager.onReceive(BluetoothManager.java:111)
at android.app.LoadedApk$ReceiverDispatcher$Args.run(LoadedApk.java:755)
... 9 more
```

Figure 5.3.: Stack trace for the Null Pointer Exception crash in updateState

This was likely caused by the OS returning a null value for the local Bluetooth name.

Another crash that occurred in the UI rendering methods of the OS was reported from a ZTE Blade device with Android 4.2.

Due to limited retention in the Android Developer Console, the details of some crash reports were lost. The author remembers a crash report related to the above-mentioned SELinux issue, as well as a small number of crashes (approx. 1-3) likely originating from the Android OS and the Bluetooth stack. No crashes that would clearly show issues with the application itself were reported, and no widespread crashing was seen. As crashes are only reported if the user chooses to do so, this is no definitive proof that the application is stable, but it makes it unlikely that serious crash bugs affecting many users exist.

[8] `http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html`
[9] as of 2013-11-24

5.3 Bluetooth bugs

5.3.1 Pairing popups

In order to communicate, Bluetooth devices need to be *paired*. Android supports two types of Bluetooth/RFCOMM sockets: Regular ones which require a proper pairing with user interaction, and "insecure" ones which create an unauthenticated "just-works" temporary pairing, introduced in Bluetooth 2.1. Naturally, the latter kind was used for this experiment.

Android versions before 2.3.6 often cause visible Bluetooth pairing dialogs on one or both involved devices (even if the other device is running a more current version). An example can be seen in figure 5.4. For this reason, devices with older versions than 2.3.6 were excluded from participation. Since app compatibility can only be defined with "API level" granularity, and Android versions 2.3.3 through 2.3.7 have the same API level, device compatibility was set to include API level 10 devices and the application refused to work on devices with Android versions 2.3.3-2.3.5.

Figure 5.4.: Pairing popup on a participating device

After this fix was deployed, the issue was only reported one more time: A participant reported a non-reproducible instance of the pairing popup issue on two devices running 4.1.2 and 4.2.2, which went away after a reboot. It is unclear if this was possibly triggered by a third device or not.

A remote "kill switch" was implemented. Should a device disrupt the network by causing pairing popups on other devices, a special message could be provided to the application which would disable outgoing connections and change the Bluetooth name to prevent incoming connection attempts. This was not used during the experiment, but was triggered after the experiment had ended (see section 5.4.2).

On a Sony Ericsson Xperia device, a notification from a software called "LiveWare manager" appeared whenever a device connected, and suggested to configure automatic actions to be performed when this device connects.

A Nexus 4 with stock Android 4.4.2 was displaying "Bluetooth connected" notifications (see figure 5.5). The cause of this is unclear; it may have been caused by third party software.

5.3.2 Visibility timeout

To allow other devices to discover ("see") a device, it has to be configured to be *discoverable* (visible). The Android API allows applications to request this by starting an Activity with `ACTION_REQUEST_DISCOVERABLE`. However, this only enables discoverability for two minutes. Starting

Figure 5.5.: Screenshot provided by a participant, showing the unwanted “Bluetooth connected” message

with Android 2.3.5, adding an `EXTRA_DISCOVERABLE_DURATION` extra with a value of 0 should make the device permanently discoverable.[10]

As devices with older android versions cannot participate due to other issues, participating devices should not be affected by this issue, in theory. Unfortunately, in practice, some of these devices *are* affected. For example, this issue was observed on a Galaxy Ace with Android 2.3.6. Affected devices ignore the value of `EXTRA_DISCOVERABLE_DURATION` and display a dialog requesting discoverability for 120 seconds. The Activity result code indicates the value that was actually set. This can be used to detect this issue and disable the warning which instructs the user to reconfigure Bluetooth.

Additionally, some Android 4.x devices contain manufacturer-specific changes that prevent the timeout configuration from working correctly. On these devices, the dialog shows that permanent discoverability is requested, and the Activity result code is 1 (at least on one of the tested devices). Nevertheless, Bluetooth is only enabled for the timeout configured in the settings. Unless the user manually disables the discoverability timeout in the settings, the device will stop being discoverable after two minutes. It is unclear if all devices affected by this issue allow the user to configure the timeout.

Affected devices will not be visible and cannot receive connections. However, they can still participate in the experiment (or other Bluetooth-based peer-to-peer networks), as they can discover and create outgoing connections to other, visible devices.

As visibility cannot be set programmatically, it is not possible to work around this issue by automatically re-enabling visibility when the timer expires.

[10] `https://android.googlesource.com/platform/packages/apps/Settings/+/cdd9d033b0f16c705433f009812e69bed977e1cf`
`https://android.googlesource.com/platform/packages/apps/Settings/+log/android-2.3.5_r1/src/com/android/settings/bluetooth/RequestPermissionActivity.java`

5.3.3 Bluetooth name

Some devices cannot be reliably/persistently configured to use the specified Bluetooth name. The name cannot be set or is regularly reset e.g. to the original, manually configured name. The exact cause is unknown and likely to be a bug in the OS or device-specific Bluetooth components.

5.3.4 Inability to accept connections

Sometimes, the Bluetooth stack on devices may break and stop accepting incoming connections. Rebooting the device or toggling Bluetooth off, then on again will resolve the issue.

This is a known issue with some Android versions as described in [11] and [12], and likely responsible for a large number of the failed connections observed in the experiment.

```
D/BTIF_SOCK(16304): service_uuid: f5f20845-3507-413d-9727-abc4a8a15659
D/BluetoothSocket(6678): connect(), SocketState: INIT, mPfd: {ParcelFileDescriptor: FileDescriptor[49]}
W/bt-sdp(16304): process_service_search_attr_rsp
E/bt-rfcomm(16304): RFCOMM_CreateConnection - already opened state:2, RFC state:4, MCB state:5
E/bt-btif(16304): bta_jv_rfcomm_connect, RFCOMM_CreateConnection failed
W/bt-btif(16304): invalid rfc slot id: 1977
E/BluetoothClientThread(6678): Connecting Exception
E/BluetoothClientThread(6678): java.io.IOException: read failed, socket might closed or timeout, read ret: -1
E/BluetoothClientThread(6678):   at android.bluetooth.BluetoothSocket.readAll(BluetoothSocket.java:504)
E/BluetoothClientThread(6678):   at android.bluetooth.BluetoothSocket.waitSocketSignal(BluetoothSocket.java:481)
E/BluetoothClientThread(6678):   at android.bluetooth.BluetoothSocket.connect(BluetoothSocket.java:323)
E/BluetoothClientThread(6678):   at de.janschejbal.p2pbluetoothtest.bluetooth.BluetoothClientThread.run
                                                                                    ↪(BluetoothClientThread.java:89)
```

Figure 5.6.: Log output seen on a client attempting to connect to a device unable to accept connections

5.3.5 Reuse of message IDs, node ID transfers

Application data, including the node ID and the broadcast state, can be backed up, e.g. using the `adb backup` command or a third-party backup application on a rooted device. Restoring the backup can lead to a number of unwanted conditions. If the backup is restored onto another device without disabling the original device, two devices with the same node ID can exist. Even if the original device is disabled or the backup is restored onto the original device, the old broadcast state will be restored. This causes the node to reuse message IDs that were used before. The impact on the network is negligible; however, users of the data should be aware that this can happen.

No node IDs reporting different hardware models were found in the logs.

Additionally, the broadcast state is only saved every 5 generated messages. If the application is forcefully terminated (e.g. due to extreme memory pressure, updates, user action, automatic “task killer” apps, or crashes), the broadcast state may be lost and the messages may be regenerated or re-downloaded from other nodes.

In total, 2502 message IDs were used more than once by the same node.

[11] Android Issue 41415 - `http://code.google.com/p/android/issues/detail?id=41415`
[12] Android Issue 41110 - `http://code.google.com/p/android/issues/detail?id=41110`

5.3.6 Corrupted transmission

At some point, some data became corrupted, causing node 00fcd028d80e18af to receive a bogus message from node a34615413018af2b with a very high ID. [13] (The nodes will be identified by the first 4 characters of the node ID below). As messages are generated based on the last known ID, further messages from 00fc also got high IDs. The impact on the network is negligible, but it is unclear what caused this. There are no logs indicating that any node sent such a message to a346, but the log from a346 indicates that it received it from 00fc. Analysis of the full logs reveals that 00fc sent messages 265, 266, 267, ..., and a346 received 265, 266, 1211195978, followed by an input exception due to an invalid length. It is likely that the data was garbled during transmission, although the error could also have been introduced e.g. by memory errors while encoding or decoding the message. Nevertheless, this shows that software should not assume that the error detection mechanisms in Bluetooth will reliably prevent transmission errors.

In a real network, messages would be cryptographically signed, so the invalid message would be rejected during the signature verification step.

5.3.7 Total failure of Bluetooth

The device of the author suffered total failure of Bluetooth during the experiment. Enabling Bluetooth was either impossible or quickly resulted in a crash of some Bluetooth-related components. Reboots did not solve the issue and the issue could not be traced to a specific cause. The device had been rooted in the past, but root access has been lost during an update before the experiment was started. A full reflash of the phone using a factory image solved the issue only temporarily.

Logs showed segmentation faults related to `/system/lib/hw/bluetooth.default.so`. The problem could not be traced and the device was returned under warranty. As this behavior was not reported from any other device, a hardware failure should be considered as a possible explanation.

5.3.8 Device reboots

Two user reported random reboots of their Galaxy S3 devices running Android 4.1.2. One user was able to confirm that the device was not particularly hot. This is likely an issue with device-specific low-level Bluetooth components.

5.4 Privacy issues

Unfortunately, three issues led to the unintentional collection or retention of data. The author strongly believes that while erring is human, such mistakes should be openly admitted to allow others to learn from them, and avoid making them in the future.

5.4.1 Failure to anonymize error logs

IP logging was disabled for the access logs in the nginx web server using a custom log format. Unfortunately, this is not possible for error logs. For this reason, error logs were supposed to be anonymized daily during log rotation using the following postrotate setting:

[13] Node IDs are pseudonymized IDs as found in the released logs

```
postrotate
    [ ! -f /var/run/nginx.pid ] || kill -USR1 `cat /var/run/nginx.pid`
    perl -pe 's/client:.*?,/client: 127.0.0.1,/' -i error.log.1
endscript
```

This was successfully tested by forcing a log rotation and verifying that the logs were anonymized. Later, it was discovered that IP addresses were stored in the logs without anonymization. The cause for this was that the postrotate script is run in the current working directory, not in the log directory. During testing, logrotate had been invoked from the log directory. Thus, the script worked correctly and the error was not discovered. However, when logrotate was run as part of a cron job, this was no longer the case. Since only a relative path to the error log was given, the anonymization command failed to find a file, leaving the logs unanonymized.

Upon discovery, this issue was remediated by running the following command (formatting added for better readability):

```
{
    ls /var/log/nginx/error*.gz --sort=time -r | xargs zcat;
    cat /var/log/nginx/error.log.1;
            cat /var/log/nginx/error.log;
}
        | perl -pe 's/client:.*?,/client: 127.0.0.1,/'
            | gzip > nginx-errorlogs-anonymized.gz
```

The first part (within the braces) outputs all logs. The output is then piped into the anonymization script, compressed, and written to disk. The original logs and all known copies were deleted and the logrotate configuration was adjusted.

This issue is not very serious - the logs were not available to third parties, were only stored on an encrypted disk, were not used, and did not contain very critical information. Only requests that triggered errors were logged. A review of the anonymized logs showed that such requests came mainly from two sources: Apple devices blindly requesting icon files, and scans for vulnerable systems (e.g. outdated phpMyAdmin installations). The former caused IP addresses of web site visitors to be logged. No data directly linked to participants was logged due to this mistake.

5.4.2 Discovery of former participants

The privacy policy stated that participants can stop participating by uninstalling the application. While this will stop their phone from collecting and uploading any data, other, still participating nodes may log some information about the ex-participant. This possibility was unfortunately overlooked in the original design.

Nodes identify possible participants by their Bluetooth device name. Any device matching the naming pattern (the “P2P-” prefix, followed by 16 hex characters) is considered to be a participant and logged. Only logging devices that respond to service discovery requests for the service implemented by the application would not be sufficient, since many participating devices are unable to communicate properly for various reasons (see e.g. sections 5.3.4 and 5.3.2).

The correct way to approach this, had the issue been discovered in time, would have been to instruct users to rename the Bluetooth device after uninstalling. As the identity of the participants is not known, it was impossible to reach out to former participants who had already uninstalled the application (and who were thus affected by this issue).

Some devices remained active well past the end of the experiment. To protect these, the kill switch (described in section 4.6.2.13) was activated for all devices. The kill switch causes the application to automatically replace the prefix in the Bluetooth name with “disabled”, thus stopping other nodes from treating such nodes as participants.

Sanitizing the logs

Due to this issue, the logs contained data which should never have been collected. This data had to be removed from the collected logs before analysis and publication. This was difficult for a number of reasons.

First, it was necessary to determine when a user removed the application. Since uninstalls were not logged, this information had to be derived from other sources. Any of the following events can be considered a reliable indication that a certain node still had the software installed and running at the time when the event occurred:

1. The node successfully establishes a connection with other nodes. Since this requires successful service discovery of a service matching the UUID of the Bluetooth service used by the application, this means that the application is still running and communicating, even if the connection fails immediately after establishment.
2. The node collects a log line.
3. The node uploads logs to or fetches messages from the collection server.

Since the node ID is stored in the application-specific folder and thus deleted when the node is uninstalled, seeing a node ID at a certain time means that the application was not uninstalled before that time.

The web server logs show when a certain node was fetching messages or uploading logs, including the node ID. Since the log retention time was initially set too low, server logs are only available from 2013-11-10 onwards. File timestamps from uploaded logs provide additional timing information.

Unfortunately, the author forgot to install an NTP daemon on the server, making the timestamps unreliable. The server was 171 seconds behind on 2014-02-16. This makes it difficult to use the server logs as a reliable time base. Under the two assumptions that the clock drift is approximately constant and that the server was set to a correct time at some point in the past, server-reported events may have occurred later than the indicated time, but not (significantly) earlier. Thus, if one of the server-based sources (logs or upload timestamps) indicates that the node was active at time t, the author assumes that the node was indeed active at that time.

The logs collected by nodes can also suffer from timing inaccuracies. This is problematic for two reasons: If the device on which the software was uninstalled has a clock set to the future, log data collected after the deinstallation will appear as if it was collected before. Likewise, if a node with a clock set to the past discovers the device of a former participant, the log will appear to have been collected while the device was still active. This has to be considered when sanitizing the logs.

The access logs were collected using the following command (formatting added for better readability):

```
{
    ls /var/log/nginx/access*.gz --sort=time -r | xargs zcat;
            cat /var/log/nginx/access.log.1;
            cat /var/log/nginx/access.log;
    } | gzip > nginx-accesslogs.gz
```

and subsequently the same pseudonymization procedure already used on the logs was applied to the contained IDs. The times from upload timestamps and access logs were combined into a “last seen by server” table and inserted into the database.

Each of the following four conditions indicates that a discovery entry is safe to keep:

1. The device that made the discovery successfully connected to the discovered device after the discovery, determined using the relative elapsedRealtime timer and the corresponding uptime ID to ensure the timer values can be compared.
2. Any device successfully connected to the discovered device after the discovery, determined using real-time timing.
3. The discovered device generated a log line after the discovery, determined using real-time timing.
4. The discovered device contacted the server after the discovery, determined using real-time timing.

To ensure the privacy of the participants, real-time timing is used under worst-case assumptions, i.e. the log entry is anonymized unless it can be guaranteed that one of the conditions is true. Thus, if the discovering device does not have usable timing information for the time of the discovery, only condition 1 can be used to determine whether the data can be kept. If a connection event does not have reliable timing information, it cannot be used in condition 2. The same applies for log lines and condition 3.

Even if real-time timing information is available at the time of discovery, some uncertainty applies (see section 5.5). The latest possible time is used - possibly assuming the discovery happened later than it actually did, i.e. erring towards removal. Likewise, the earliest possible time is used for events which confirm that the app was still installed (e.g. successful connections, seen log lines).

Completely deleting these entries is not feasible - connection attempts may have been made to these devices, the count of discovered devices would have to be edited, etc. The risk of introducing errors into the logs by doing such large manipulations would be too high. For this reason, these log entries are simply anonymized to P2P-xxx...xxx. This is done by creating lists of log entries to anonymize and to keep, then re-importing the database. The import script uses these lists to anonymize the corresponding rows before inserting them into the database. To avoid mistakes, consistency is checked using both lists (every entry must hit exactly one list, and all entries in both lists have to be used exactly once).

5.4.3 Logging of device names

Although only devices showing a Bluetooth name matching the naming pattern were logged during the discovery process, other device names could be logged in certain situations. For example, device names of incoming connections were logged even if they did not match the naming pattern. Thus, incorrectly named devices which were making outgoing connections had their full device name logged. This is a serious issue, since some these device may contain real names of participants.

To fix this, the pseudonymizer was configured to report any values not matching the expected format and replace them by xxx...xxx (with a fixed length). The log pseudonymization was rerun and the resulting logs were searched for the reported values to verify complete removal.

5.5 Real-time timing information

The easiest way to obtain real-time information is, unsurprisingly, using the NTP data. However, even this is not as easy as it seems - NTP responses can be invalid (see sectuib6.5), clock drift limits the timeframe around an NTP event for which the information is useful, long round-trips on congested networks make responses inaccurate, and devices are not always connected to reliable WiFi, leading to gaps in NTP data coverage. Some devices never manage to log a single NTP event during their lifetime in the network.

Data from a logged NTP event provides information about the difference between the relative `elapsedRealtime()` timer and real time. This difference will be referred to as “updiff” below. The

system clock cannot be reliably used, since it can be set manually by the user or automatically by the operating system. Thus, an NTP event only provides useful information as long as it is guaranteed that the `elapsedRealtime()` timer was not reset between the NTP event and the time of interest. Such times can be identified using the uptime ID, introduced in section 4.6.2.2.

5.5.1 Deriving real-time timing from NTP data

The following process can be used to determine real-time timing information: For each device, each uptime (i.e. a time span identified by the uptime ID) is split into one-hour segments using the `elapsedRealtime()` timer value. An assumption is made about the maximum clock drift rate to be expected per segment. The minimal and maximal updiff of all NTP events falling into a segment are taken, and extended by the assumed maximal clock drift value. Thus, the minimal updiff min_updiff(s_0) of the segment s_0 is the minimum of the updiffs of the individual events, minus the maximal clock drift per segment. Accordingly, the maximal updiff max_updiff of the segment is the maximum of the individual updiffs, plus clock drift.

Not all segments, however, have NTP information. Timing information can be derived from other segments within the same uptime. As two segments s_a and s_b are $|a-b|$ hours apart (neighboring segments are considered one hour apart, since the second segment covers one additional hour), we know that the clocks can have drifted at most $d \cdot |a-b|$ where d is the assumed max-drift-per-segment value. Note that other sources (e.g. NTP data, other segments, connection-based derivation of timing data) may provide us with better values, i.e. a larger min_updiff or a smaller max_updiff. Thus:

$$\text{min_updiff}(s_a) \geq \text{min_updiff}(s_b) - d \cdot |a-b|$$

$$\text{max_updiff}(s_a) \leq \text{max_updiff}(s_b) + d \cdot |a-b|$$

5.5.2 Improving timing using connections

Using knowledge of the protocol (see protocol definition in section 4.5) and the logged exchanges, further information can be derived. Let h_1 be the real time when device 1 send a HELLO message, s_2 the time when device 2 received the HELLO message and sent a STATE message, and r_1 the time when device 1 received this STATE message and sent the REQUEST message.

From the protocol definition, we know $h_1 < s_2 < r_1$. Thus, if we have real-time timing information for h_1 and r_1, we can use this to derive timing information for s_2. The logs from device 1 provide an `elapsedRealtime()` timer value, and there are min_updiff, max_updiff values for the corresponding segment as per our assumption. The logs from device 2 provide an `elapsedRealtime()` timer value for s_2. (Note that the two devices have independent, unrelated timers!) The minimum real time of an event can be obtained by adding the min_updiff to `elapsedRealtime()` timer value. (Likewise for the maximum real time, using max_updiff). Let these values be called h_1min, h_1max etc. for each of the events above. $h_1min < s2 < r_1max$ provides upper and lower bounds on the real time at s_2. Together with the second device's `elapsedRealtime()` timer value for s_2, these can be used to calculate min_updiff and max_updiff for the second device. Since this value only represents one particular point in the segment, at least the per-segment clock drift value must be subtracted/added to these values.

Since this provides new timing data, re-running either expansion could now further improve the data. Alternating between these two expansion methods for several times significantly improves timing information. Out of 52771 segments, 29898 had NTP timing data. Expansion based on neighboring segments provided some (not necessarily very accurate!) timing information for 50205 of them, leaving 2566 without any information. Running the connection based expansion reduced this to 2045. The next iteration of the segment expansion brought this down even further to 1328, quickly stabilizing at 1319 after one more pair of expansions.

This can be implemented purely in SQL, corresponding queries are provided together with the data.

5.5.3 Estimating drift

By measuring the difference between the minimal and maximal updiff of all valid NTP events during each uptime, and dividing by the duration of the uptime (the difference between the highest and lowest `elapsedRealtime()` timer value reported), we get a rough estimate for maximum clock drift. (Valid NTP events have a round-trip time between 1 and 1999 milliseconds, inclusive). This is a rough estimate: Network delays make NTP responses less accurate. Given the above definition of valid NTP events, up to 4 seconds of false absolute clock drift can be reported. This makes reports during short uptimes mostly useless, but does not significantly influence longer uptimes. Clock drift that changes direction and thus partially cancels itself out is also underreported. With this in mind, the worst per-hour drift rate reported in an uptime longer than 5 minutes is 9 seconds per hour. Three other devices reported drift rates of 7 to 8 seconds. These four devices came from three different manufacturers, indicating that this is not caused by a specific faulty model.

In order to determine guaranteed real-time information for log sanitization (see section 5.4.2), a very conservative assumption of up to one minute per hour/segment was made for the clock drift rate. For regular analysis, lower clock drift rates can likely be assumed safely. If values do not have to be guaranteed to be correct, the NTP data can also be used to estimate how reliable the system clock is. If a device is consistently reporting very small differences between system and NTP time over a long time, one might choose to trust the system clock. Usually, this will provide very accurate results. However, it can also provide blatantly false values, so caution needs to be taken. For log sanitization, the risk of inaccurate values is not acceptable, and thus this method was not used.

6 Observations

6.1 Participating devices

293 nodes (identified by unique node IDs) submitted logs. They represent 92 different models (identified by the model string) from 21 manufacturers, with 14 different Android versions. In total, 200 distinct configurations (a configuration being defined by the reported BUILD variable values) were observed. This shows how highly heterogeneous both the Android ecosystem and the device set participating in this experiment are.

Additional nodes that did not submit logs were seen by other nodes. Some of these were removed during the log sanitization process explained in section 5.4.2. 10 such nodes remained, resulting in at least 303 distinct node IDs participating.

Deleting the application data or uninstalling the application destroys the node. Should the application be activated again afterwards, a new node ID will be generated. Since no hardware identifiers are collected, it is impossible to say accurately how many physical devices participated.

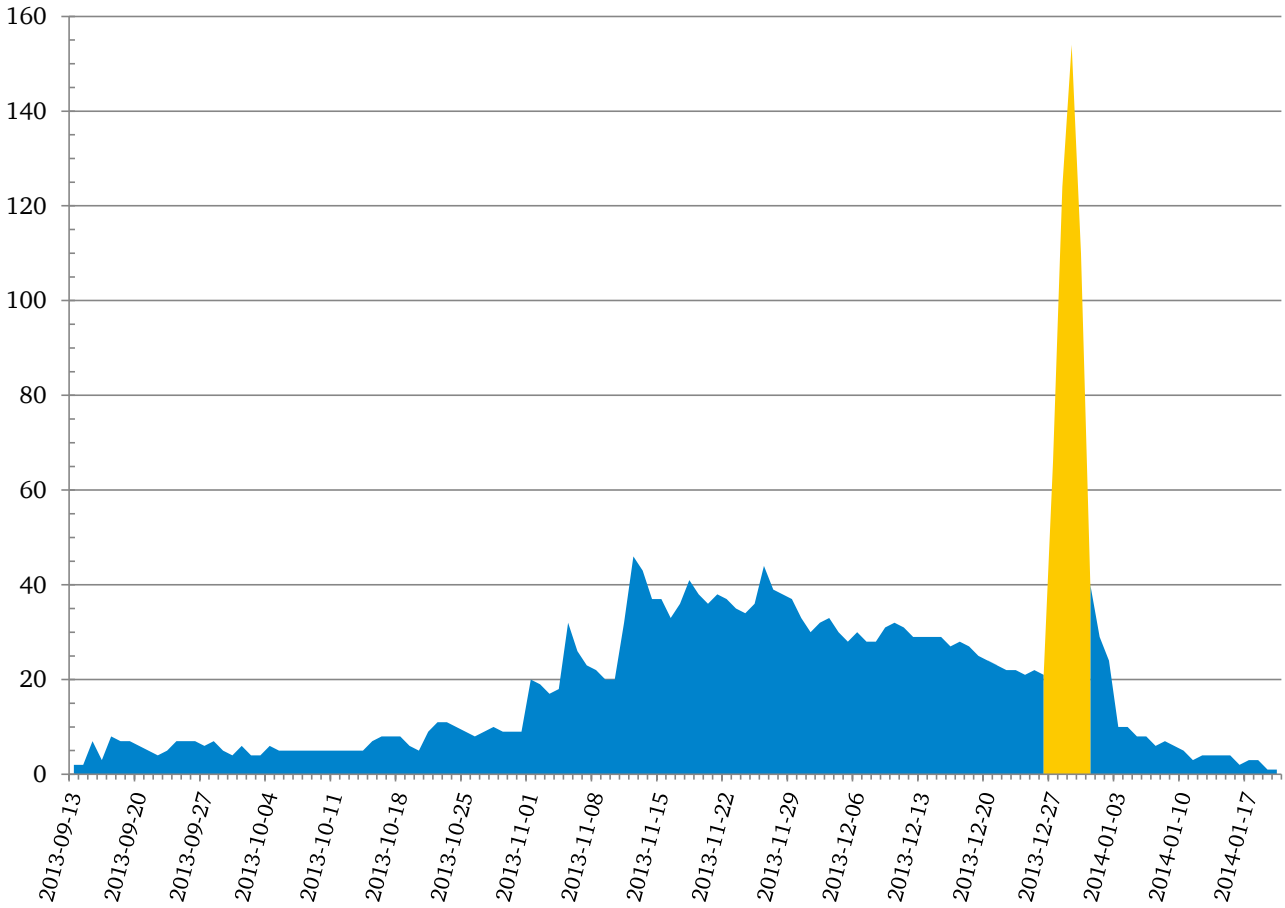

Figure 6.1.: Number of reporting nodes over time (the highlighted timeframe represents the Congress)

The participation of individual nodes can be seen in figure 6.2.

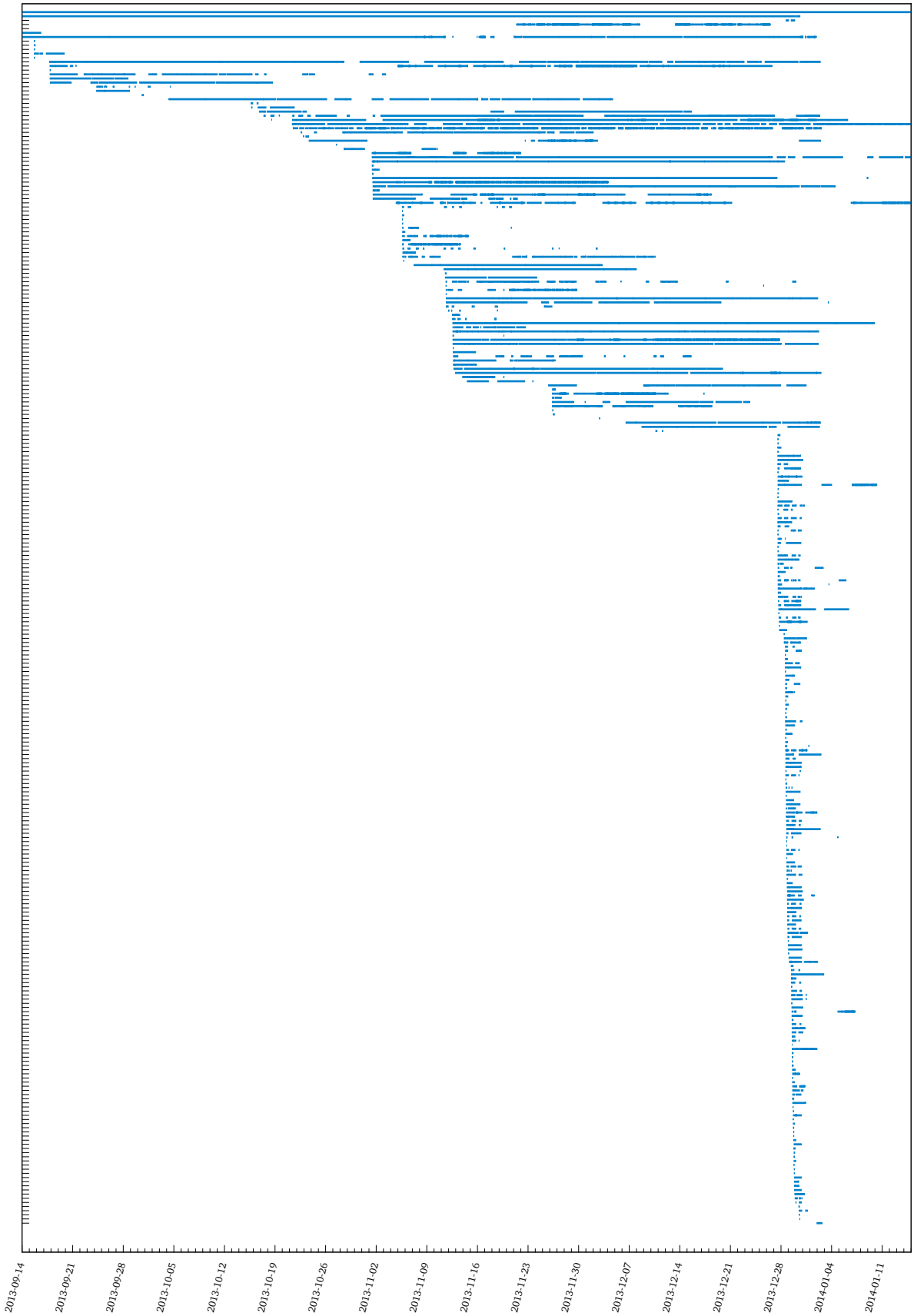

Figure 6.2.: Node presence over time. For each node, the (system clock-based) times during which events were generated are shown. The times where the author was actively recruiting new participants are clearly visible, as is the Congress.

6.2 Node degrees

Using a table `extract_discoveries`, which contains only log lines indicating that a participant has been found, determining the degrees of nodes (in regards to discovery) is trivial:

```
SELECT device as node, COUNT(DISTINCT c2) as out_count FROM extract_discoveries GROUP BY device ORDER BY out_count DESC;
SELECT AVG(out_count) FROM (SELECT device as node, COUNT(DISTINCT c2) as out_count FROM extract_discoveries GROUP BY device);
```

The average degree is 26.9, and only 47 nodes have a degree of more than 50. Considering only devices from the 30C3 group (see section), the average degree is 32.8, and the maximum degree is 109.

In terms of successfully established connections, the average degree is 22.6 (out) and 21.6 (in).

6.3 Distribution of messages

The most interesting result is the performance of the broadcast network itself, as it practically proves the power of such networks. To measure the effectiveness of the network, the number of nodes each message reaches is calculated. The steps to do so are provided in appendix A in detail, giving a practical example for SQL-based log analysis. More complex examples are provided on the web site hosting the data set.

Figure 6.3 shows how many nodes are reached by messages, depending on the generation time of the message. Only messages for which the generation time is known with some reliability are considered, and only nodes that reported the receipt of the message in a log are counted.

A very large number of messages never sent to any other nodes are not shown. There are several reasons why a message may never leave the generating node. For example, Bluetooth can be disabled, the node may be unable to communicate with other nodes due to software issues, or the node can simply be out of range of any other nodes.

A large dot at a position indicates that a large number of messages generated at the time on the x-axis reached the number of nodes on the y-axis. For example, the topmost dot in figure 6.3 indicates that a small number of messages generated on 2014-12-27 reached approximately 168 nodes. The larger dots below indicate that many messages generated on 2014-12-27 reached 150-160 nodes.

Note that the chart does not show *when* the message reached these nodes, only the date when the message was generated is shown. Thus, messages generated earlier have a better chance of reaching many nodes: They can reach all nodes active at that time *and* any node that becomes active later.

During the low-density phase of the network, the number of nodes reached varies, although there is a hard top limit. The limit represents the total number of working nodes that could be realistically reached. Since nodes stop distributing messages a week after they first receive it, and new messages push older messages from storage, few if any of the early messages survived long enough to make it to the Congress.

During the Congress, however, most messages either never left the originating device, or they were distributed across the entire network. This can be seen by the relative lack of markers below the crowded top line. Some of the messages generated earlier made it to the Congress, and were distributed to almost all devices there. Towards the end of the Congress, this distinction became less obvious. Messages generated towards the end only had limited time to spread, and could only profit from the nodes active during that timeframe. Messages generated early profited from all nodes - both ones that operated only in the beginning, and those that started operating later and received the message from the network.

By only counting messages that arrive within 24 hours, the impact of node density becomes even more obvious. In the low density scenario, most messages only reach a small fraction of nodes within 24 hours.

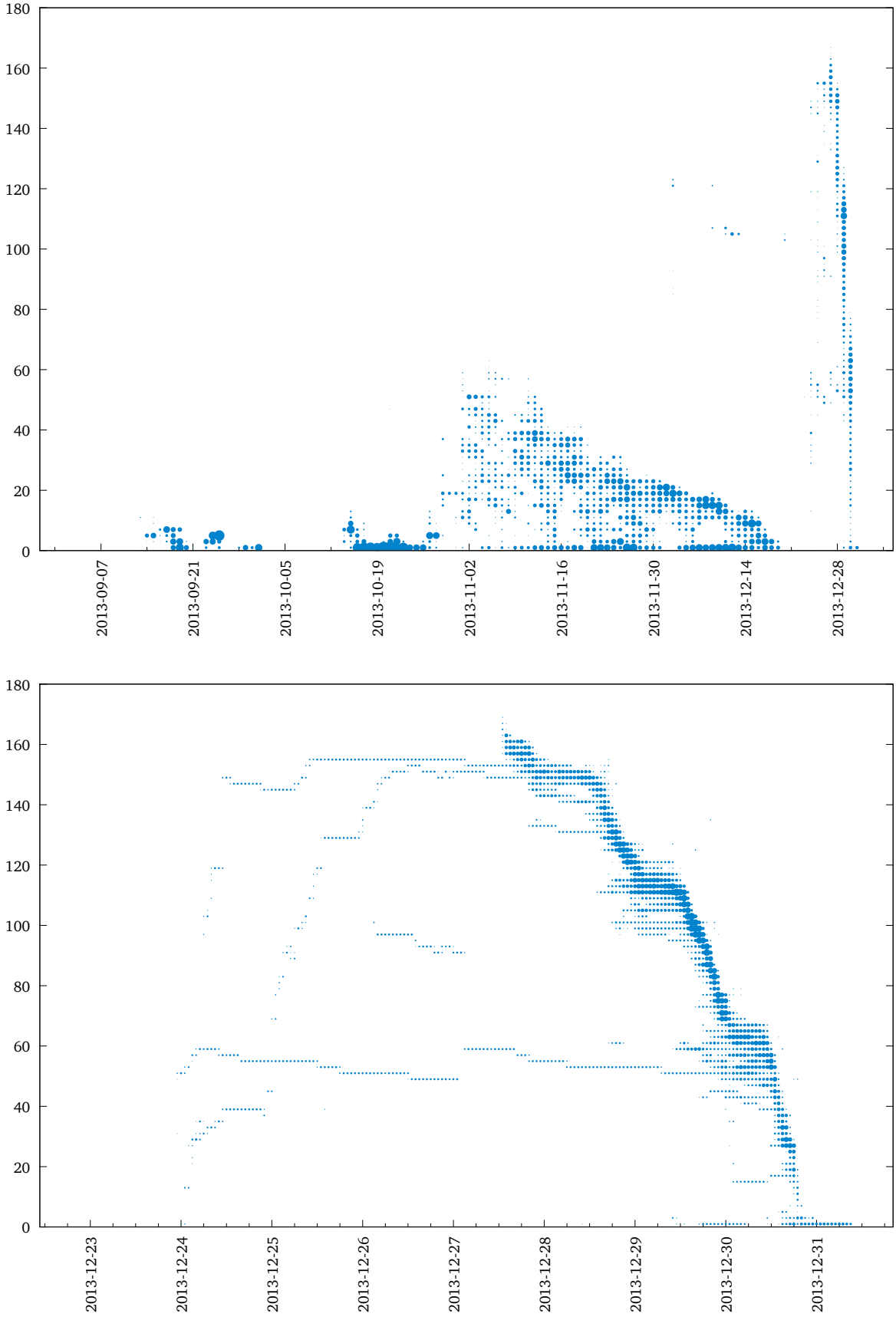

Figure 6.3.: Number of nodes reached depending on the time when the message was generated. Top: all time, bottom: finer granularity around the Congress.

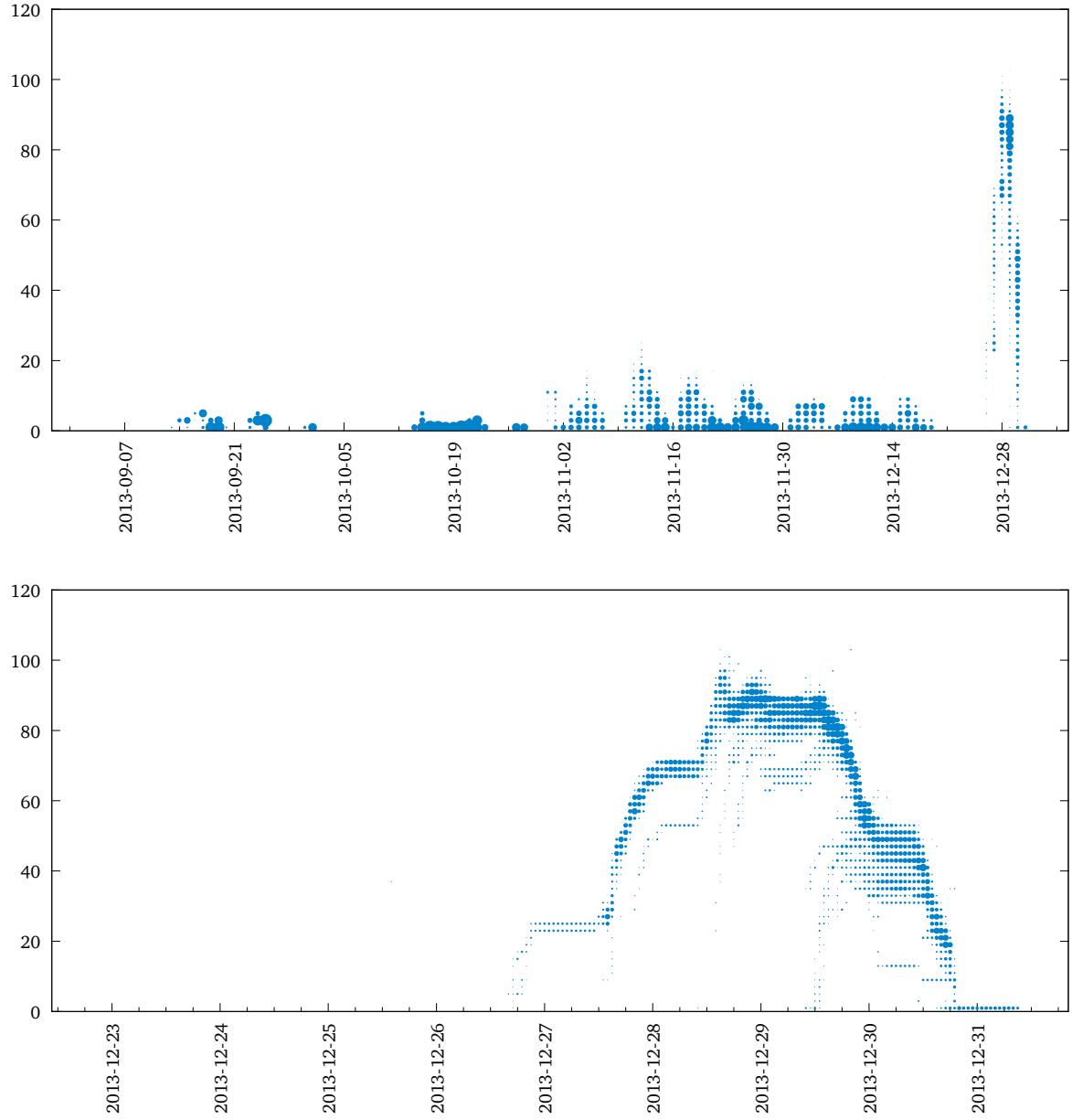

Figure 6.4.: Number of nodes reached within 24 hours depending on the time when the message was generated. Top: all time, bottom: finer granularity around the Congress.

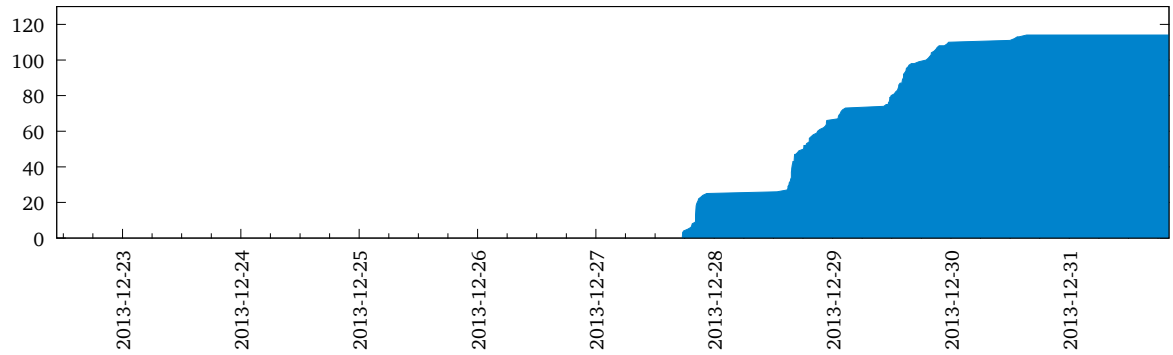

Figure 6.5.: Spread of a selected message: Number of nodes with message a800ce7c9046ab8b:7321 over time.

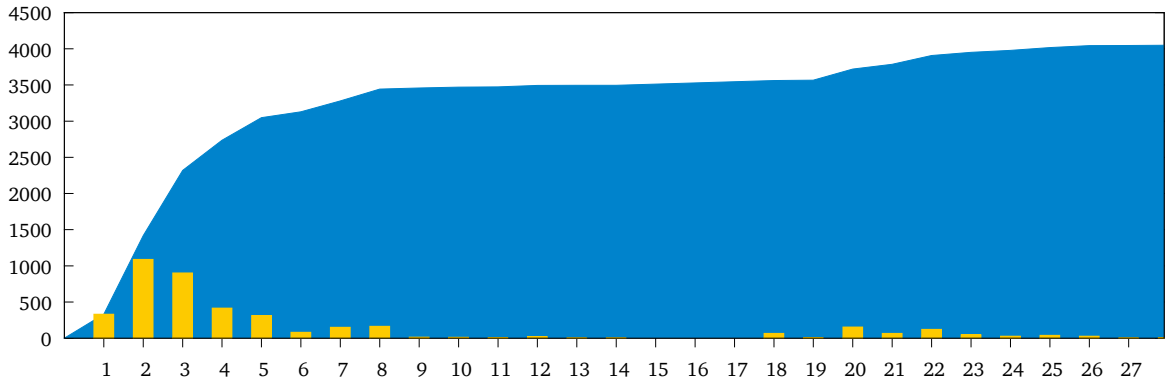

Figure 6.6.: Message propagation example during Congress: Message receivings within x hours of generation (blue) resp. during the x-th hour after generation (yellow).

This can be seen in figure 6.4, where large clusters appear at the bottom of the chart. In the high density scenario, most messages reach a large number of nodes - large clusters are seen near the maxima. This indicates that most messages reach *nearly* all of the nodes that are active and able to communicate at the Congress within that 24-hour timeframe. However, significantly more stragglers are seen, indicating that some messages take longer than 24 hours to deliver.

Since only nodes where accurate timing information exists for both message creation and reception are considered, the numbers in figures 6.3 and 6.4 are not directly comparable.

To get a better idea of message distribution, an arbitrary message was selected (based on number of timing-enabled nodes reached and creation date). Message 7321 from node a800ce7c9046ab8b was created on 2013-12-25 20:23:52Z (i.e. before the Congress) and reached a total of 114 nodes. As messages generated by that node during the Congress also reached significant distribution, it is likely that the node was present at the Congress. Its distribution can be seen in figure 6.5. The number of reached nodes rises as new nodes are constantly joining the network.

Figure 6.6 shows how quickly messages were received after generation, considering messages generated during the first ten minutes after 2013-12-29 16:00Z. Each time a node receives one of these messages for the first time is counted, i.e. a message delivered to 100 distinct nodes counts as 100 receivings for the purpose of the chart. To remove the influence of new nodes joining, only nodes that already had received at least one message before 2013-12-29 15:00Z are counted. This demonstrates that at least for messages during this segment of the Congress, more than 50% of message deliveries happened within 3 hours. However, it also shows that less than 10% of the deliveries happened within the first hour.

6.4 Throughput

The throughput between nodes varies widely. To measure throughput, the number of payloads transferred during a connection is counted, and divided by the time between the first observed message and the last observed message. Additionally, only connections including an offer (`MESSAGE_STATE`) are considered. This guarantees that timing starts before the first payload message is sent, as the remote node only sends payload messages in response to a request, which is sent by the local node after receiving and logging the offer. The measured timeframe is thus guaranteed to start before the first PAYLOAD message can be sent. This is done to ensure that the reported speed is never higher than the actual speed, even if e.g. a bunch of messages get stuck in a buffer and then processed at once. Naturally, this means that the reported speeds are lower than the actual speeds in most cases.

When only connections with at least 100 payloads are considered, speeds measured this way range between 0.8 KB/s and 235.9 KB/s. This provides us with a lower bound for best-case real-world Bluetooth speeds of 1.9 Mbit/s. This corresponds well to the theoretical maximum of 2.1 for the net data rate in Bluetooth 2.1 EDR.[1] The maximum speed was observed between two Nexus 4 devices. Transfer rates above 100 broadcast messages per second can be seen in 350 connections.

Current versions of Bluetooth include a "High Speed" mode which uses WiFi technology for faster data transfers. Unfortunately, accurate information on the Bluetooth capabilities of phones is rare. The data rates indicate that Bluetooth HS was not used by the phones participating in the experiment, or at least that it did not allow the phones to exceed the speed of regular Bluetooth-only connections.

Building simple statistical metrics (e.g. median) from these values would be misleading. Only a lower bound on the actual transmission speed is established. By only considering connections with a certain number of messages, slow connections are removed from the data set disproportionately, since they cannot transfer a large number of messages before the connection times out.

6.5 Timing

NTP timing data is available for 273 nodes, with 195.2k events total. For 153 NTP events, round trip times of more than 30 seconds have been reported. Three events include negative round trip times and are likely invalid. 193.6k events (covering all 273 nodes) have round trip times between 1 and 1999 ms. Only these events are considered for the following statistics.

Out of these, only 50 nodes have at least one NTP event indicating a difference of more than 10 seconds between the system clock and NTP time. 26 show at least one difference above 30 seconds, reducing further to 17 nodes for a one-minute and 8 nodes for a ten-minute difference. At the same time, 182 nodes have at least one event indicating less than 1 second of difference, and all but 8 nodes have at least one event with a difference of less than one minute.

Figure 6.7 shows the differences seen by each node, providing a good idea on the reliability of system clocks.

Notable outliers are an NTP event containing invalid, random-looking values, and a node that had its system clock reset to January 30, 1970. The invalid values likely stem from the fact that NTP uses UDP and does not include a magic value in the packets. Thus, a non-NTP packet received on the correct port at the right time can be interpreted as a NTP response by simple clients such as the one used here.

Clock drift patterns vary wildly - among the observed patterns were near-linear, uncorrected drift, near-linear corrected drift, alternating linear increases/decreases at two distinct rates with jumps in between, and many others (see fig. 6.8 for examples).

Additional information about timing issues, including pre-deployment testing, can be found in section 4.4.3. Section 5.5 provides information on how to use NTP information map timestamps in log data to real time.

[1] Bluetooth specifications: `https://www.bluetooth.org/en-us/specification/adopted-specifications`

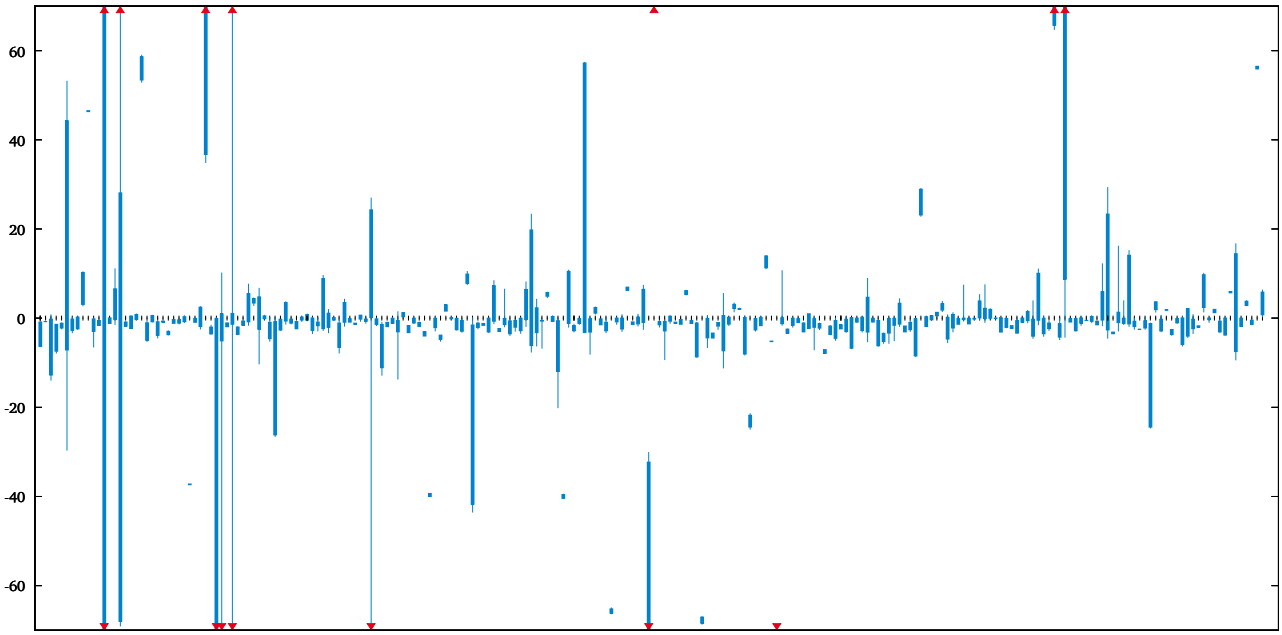

Figure 6.7.: This chart shows the minimum, maximum and 5th/95th percentiles of the differences between NTP and system time for each node (in seconds; red arrows indicate values exceeding the chart range).

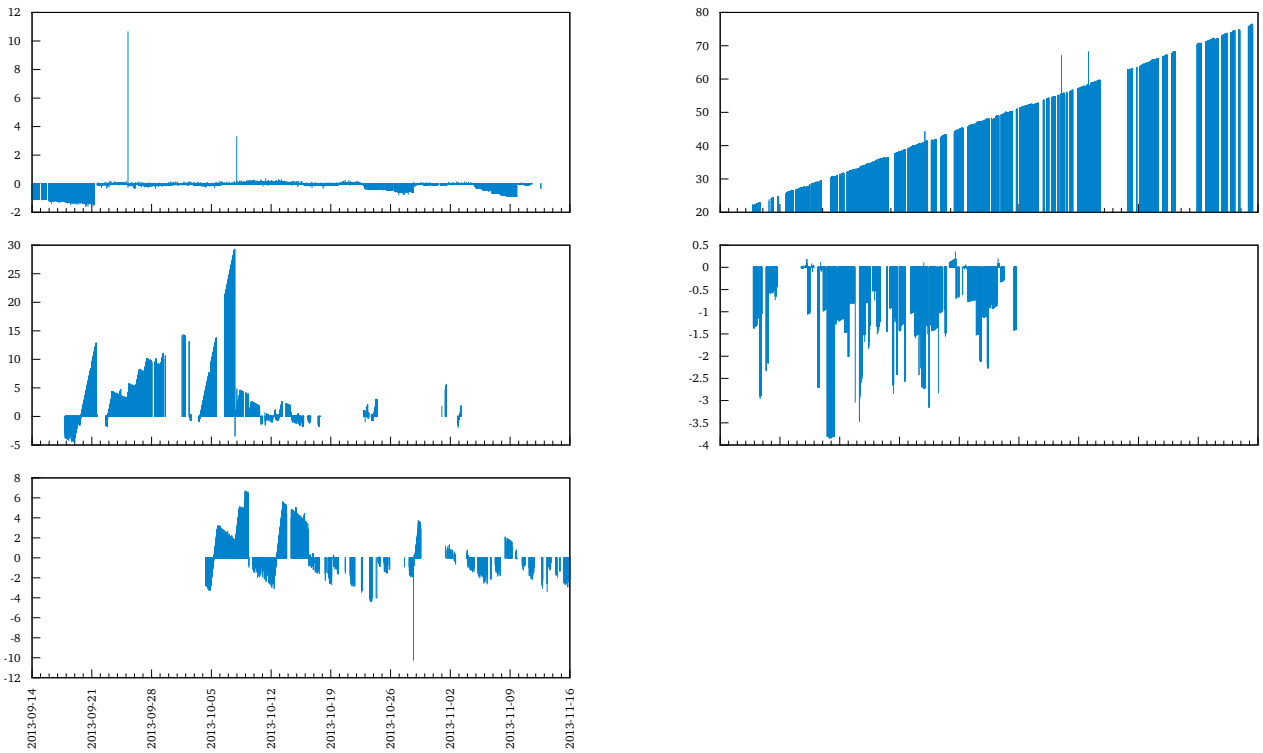

Figure 6.8.: Examples of clock drift patterns, based on raw (unfiltered) NTP data. Difference in seconds.

6.6 Impact on regular use

6.6.1 Battery/power consumption

The battery drain caused by the application varied wildly depending on the device and the circumstances (e.g. if other devices were nearby or not).

A test was performed where a Galaxy Nexus and a Nexus S were placed in one room, fully charged, with the application running and Bluetooth configured correctly. After 25 hours, the Galaxy Nexus reported a battery level of 47%, while the Nexus S reported a battery level of 31%. This test was performed with version 6. Version 7 introduced wake locks during certain phases of the communication between devices. The test was repeated with version 7. After 25 hours, the Galaxy Nexus reported 48%, while the Nexus S reported 18%. Thus, the wake lock appears to have minimal influence on power consumption on the Galaxy Nexus, but may have some influence on the Nexus S.

It should be noted that these measurements are not very accurate and can be influenced by many factors. Accurate power measurements would require specialized hardware, and actual power usage would still be influenced by randomness introduced by the wireless environment.

The production version of the application, version 8, increased some timers (e.g. generating messages less frequently). This was intended mainly to allow better analysis of message propagation and caused only negligible improvements in power consumption when tested in a two-device environment.

When no other devices are nearby, i.e. the node was only performing discovery, but not communicating with other nodes, the observed battery usage was minimal on the Galaxy Nexus (approximately 10 percentage points per day more than regular standby battery drain).

As the application logs the battery state, it might be possible to derive further battery usage information from the logs. However, since other activities on the device are not recorded for privacy reasons, it will be difficult to determine with certainty whether observed battery drain was caused by the experiment, other background applications, regular standby power consumption, user interaction with the phone, or other reasons (e.g. low temperatures affecting battery performance).

In the field, a number of users using different devices complained about high battery consumption, and impact on battery life was the main complaint amongst users. It is likely that many users silently uninstalled the application after they decided that the battery usage was not acceptable to them. Two users reported that their Nexus 4 discharged fully within 14 hours with little use except for the application. (Both devices were presumably in range of each other for extended periods of times, increasing power consumption due to communication). Another user even reported a full discharge in 2 hours, without specifying a device type. One user also mentioned that his HTC One was getting very hot, comparable to playing a game that creates a high load.

6.6.2 Memory usage

After 168 hours (1 week) of operation, the service showed a memory usage of 12 MB on a test device. Memory consumption was not logged across the fleet and may be higher when the node is heavily involved in the network. However, given the small amount of data that needs to be stored for other nodes, it is assumed that the additional load (in terms of memory) is minimal.

6.6.3 Bluetooth

One participant who used a Bluetooth headset to listen to music on a regular basis reported that the application interfered with this use. Since the application performs a Bluetooth discovery every few minutes, this is expected. Bluetooth discovery is a resource-heavy operation requiring channel hopping. The Android API documentation explicitly warns that “Device discovery is a heavyweight procedure. [...] existing connections will experience limited bandwidth and high latency” [27, BluetoothAdapter].

Some billboards in the Frankfurt subway sent unsolicited advertisements for a Ski resort to visible Bluetooth devices in the vicinity. An example of such a billboard on a platform of the station “Berger Straße” is shown in Figure 6.9.

Figure 6.9.: Bluetooth-spamming billboard with instructions to “turn Bluetooth on and visible, wait, and accept the message”

7 The power of decentralized communication networks

This section provides an outlook on the power of decentralized networks. As such, it is not directly related to the data set and the rest of this thesis. However, it shows potential applications of the research made possible with the collected data.

Most current approaches for censorship resistance like Freenet are unsuitable for scenarios where the government is willing to shut down regular communication networks, since they require the internet or at least real-time communication between nodes (for example to allow lookups in distributed hash tables). Many assume that the government will try to censor specific content, but will not try to shut down the entire network if selective censorship is impossible. Some expect the government to follow constitutional or legal limitations. As has been shown in the Arab Spring, these assumptions can no longer be taken for granted. For this reason, a new approach for censorship resistance is required. This chapter provides thoughts on the design of such an approach.

A censorship-resistant communication system that strives to be resilient against a determined nation-state adversary:

- must be able to function without the Internet and fixed communication lines,
- must be resistant against attempts to take down the entire network and
- must assume that the adversary has significant resources, possibly exceeding the combined resources of all participants of the network.

These requirements impose serious limitations on the design of such a system. For this reason, the system cannot be expected to provide all positive properties that many existing systems can provide.

At the same time, a censorship-resistant system can only be useful during a crisis if a sufficient number of people use it. This is difficult to achieve unless enough people are familiar with it before the crisis breaks out. For this reason, the system should not only provide a valuable service (resilient censorship-resistant communication) during a crisis; it should also be usable and attractive before such a crisis breaks out. This leads to an additional requirement:

- While the internet is available, the system should provide an interesting service with acceptable performance, comparable to regular internet-based services.

A communication system based on mobile peer-to-peer communications, supported by Internet links where available, can be used to fulfill these requirements.

It should be noted that the “censorship-resistance” scenario is the worst-case scenario for such a network - a scenario where a powerful active adversary is attempting to disrupt the network. A network able to work even in such difficult circumstances will be reliable and useful even during less serious situations. Thus, even if using the network for censorship resistance is not intended (see below for important caveats!), *designing* it to work even in this scenario will ensure its usefulness in other scenarios (e.g. emergencies) even if attackers attempt to actively disrupt (e.g. flood) the network. Constraints can be appropriately loosened in such scenarios to improve other properties like ease of access at the cost of losing resilience against powerful attacks.

7.1 Dangers of censorship-resistant networks

Using technical means for censorship resistance can be dangerous. Especially in the case of failing totalitarian governments, users may face serious real-world consequences like beatings, “disappearance” or murder.

Stephan Urbach, a former Telecomix activist with contacts in various totalitarian countries, warned of the dangers of providing such networks in a private discussion on this topic. The behavior of such governments cannot be accurately predicted, neither by external observers nor by the local population. Persecution often comes without warning signs - the first and only indication that using such a network is dangerous can be the disappearance or death of other users. At the same time, desperate citizens will use whatever appears to work, without realizing the risks involved, and often without any regard for such risks even if they are warned about them. For this reason, it can be dangerous to develop and provide such tools, at least for the purpose of censorship resistance. Developers of such tools need to understand that they could be indirectly responsible for the deaths of their users. [28]

The extent of persecution can depend on many factors. A failing government may be too busy with more important, higher-value targets to persecute regular, anonymous users. For example in Egypt, the military stayed out of the conflict at first, and later protected the civilian population from government attacks. Consequently, local WiFi networks were successfully operated for data exchange without serious consequences for the operators. On the other hand, people in Bahrain are being persecuted and killed for using censorship-avoidance techniques, even if they do not have any kind of “opposition leadership” role and do not themselves publish content criticizing the government. The situation in Egypt would also likely have been very different had the military chosen to take the side of the government. [28]

As the adversary will likely attempt to trace and attack senders of unwanted content, it must be possible to publish content in a way that cannot be easily traced. However, anonymity does not need to be provided by the censorship-resistant system directly as long as the system can be easily used through other, anonymizing systems. Again, this strongly depends on the specific situation. In some places, a mesh network with non-obvious distribution paths may be enough to ensure the safety of senders; elsewhere, a government may be able to trace them by observing the distribution across many nodes.

Due to the possibility of regular users being persecuted, it would be highly desirable to make it difficult to detect people using such a system. This would put severe limitations on the system (e.g. because connections to unknown nodes would be impossible) and thus limit its usefulness. Again, this is situation dependent - in one location, openly operating with visible devices may be perfectly safe, while in another, even encrypted links between trusted devices may be enough to draw the ire of the regime with fatal consequences. As said before, the capabilities of the attacker may not be known in time.

The risk for participants can be reduced when the system is used by everyone, not just critics of the regime, making it less suspicious to operate a node - especially if the distributed network functionality is part of a commonly used and tolerated application. In some countries, any kind of communication system not controlled by the government could be immediately and violently suppressed. In others, where the government is attempting to maintain a semblance of freedom, it may be possible to establish such a network as long as it happens before a crisis and the censorship-resistance is not advertised too openly. Thus, absolute stealthiness may not be as important as providing a working system that is attractive for the broad masses.

7.2 Existing solutions

7.2.1 Current solutions

Currently, both server-based and peer-to-peer communication solutions are being used.

Most popular communication and publishing services like e-mail, Twitter, Facebook, web sites and blogs are server-based. To send or publish a message, the sender creates a direct, real-time connection to a server and delivers the message. The receiver/reader can then retrieve the message, again using a direct connection to a server (not necessarily identical with the server that originally received the message). If the servers are disabled or connections to them are blocked, the system cannot be used.

Peer-to-peer based communication systems usually store messages on a number of other nodes, e.g. using distributed hash tables. The recipient can then request the message from any node who is storing it, either using a direct connection or by routing the request through other nodes. This assumes that such a request (and the response) can reach its destination within a short time. In overlay networks communicating freely over the internet, this is usually the case. However, once real-time communications fail (e.g. internet access is disabled in a country), these systems fail to deliver the requested message in an acceptable timeframe.

7.2.2 Before the internet - the FidoNet

To ensure a message is available to a recipient, it must be stored in a location that is accessible to him. In a publishing network, every node is a potential recipient, so for each node, there has to be an accessible location storing the message.

A similar problem existed before the advent of the internet. Long-distance calls (required for data transfer) were very expensive, making it infeasible to request messages from nodes outside the local calling area. A message was only available if it was stored on a local node.

Early systems like Echomail in the FidoNet [29] solved this by replicating each message on all nodes[1] in the network. (Most end users' systems only accessed the nodes, but were not directly part of the network). Nodes regularly connected to other nodes, following a hierarchy, and exchanged batches of messages.

For example, a Fido-node located in Darmstadt was used by users living in Darmstadt who could reach it at local call rates. This Fido-node would make regular connections to another node, a Fido-hub, upload all messages posted by users in Darmstadt and download messages posted by outside users. The connection was still expensive, but as all messages could be quickly sent in a compressed batch, it was significantly cheaper than sending the messages directly.

The Fido-hub would in turn make connections to a higher-tier node, the Fido-host, and again exchange messages. At the highest hierarchy level, the message would be batched together with all messages from a Fido-zone (e.g. Europe), and the batch would sent to the other Fido-zones (e.g. North America) using expensive, international connections. The message would then be distributed down along the hierarchy. For example, after the message reached a Fido-hub, it would be sent to the Fido-node the next time it connected to the Fido-hub. This way, the message from Darmstadt, Europe and every other message from Europe would reach all nodes in the US, using only a single international call.

The strict hierarchical structure of the FidoNet was managed manually [30]. Flooding and other malicious or problematic behavior was prevented by manually excluding "bad" nodes. However, the hierarchical approach leaves the system vulnerable to attack. Important nodes can be disabled, and their

[1] "Nodes" is used as a generic term for members of a network. In the FidoNet hierarchy, only the lowest tier of nodes was called a *node*. To avoid confusion, terms from FidoNet terminology will be prefixed with "Fido-" here.

operators can be persecuted. Additionally, disagreements between the operators can make operation of the network difficult. This was one of the factors that ultimately contributed to the demise of the FidoNet.

7.2.3 Usenet

Usenet is a distributed but server-based world-wide discussion system established in 1980 [31] and still in use today. It follows a non-hierarchical approach. Each Usenet server exchanges messages with a number of other Usenet servers, his *peers*. Peerings between servers are usually managed manually. Each server *carries* a certain set of groups and will only store and distribute messages posted to these groups. Messages are kept for a certain *retention time* before being discarded.

Users may post on any server that allows posting. In practice, a message posted on any large server will reach most other servers within seconds. This makes Usenet censorship-resistant in regards to selective censorship.

However, as the system is still server based, disabling the servers or preventing access to them will render the system useless. Additionally, nation-state adversaries may attempt to take down the entire Usenet by attacking servers or by flooding.

The same properties that make Usenet censorship-resistant also make it very difficult to fight such disruptive behavior. In theory, servers that repeatedly support or tolerate abuse could be excluded from Usenet by ceasing peerings and/or filtering by path. However in practice, even servers allowing unauthenticated anonymous posting are tolerated, leading to widespread abuse. Problems limited to certain newsgroups can be kept away from a server by ceasing to carry that group. For example, many servers do not carry binary groups, i.e. groups where posting of large binary files is accepted. Automatic content filters are also employed, for example to prevent posting of binary content to non-binary groups and to reduce spam. This is complemented by manual and semi-automatic spam removal systems.

Nevertheless, Usenet is a great example for an existing, practical broadcast-style peer-to-peer communication network. The Usenet approach to communication has not only worked well in the limited environment of the 1980s, it managed to last to the current day.

Its use is not limited to low-bandwidth plaintext communication, either: In the form of the "binary Usenet", the old broadcast approach designed in the 1980s for distributing text is used today to distribute and replicate Terabytes of data per day[2] to servers across the globe. While this does not represent a true peer-to-peer network where the individual end nodes themselves participate in the broadcast scheme, it does again show the power of (semi-)decentralized networks and the broadcast approach.

Since only the servers participate in the peer-to-peer-style broadcast distribution of Usenet articles, only well-connected servers have to deal with the full load resulting this approach. However, communication bandwidth, processing power and storage capacity have grown exponentially, making it feasible to involve individual end-nodes in a peer-to-peer broadcast network.

7.3 Concept overview

This section describes how a distributed, censorship-resistant communication architecture could be designed.

The proposed solution consists of a layered architecture. In the center of it is a broadcast network providing channel-based message distribution. This broadcast network can be used by applications to distribute

[2] see e.g. `https://www.binsearch.info/groupinfo.php`

their data, as described in section 7.6. The broadcast network in turn can use various technologies to actually exchange data between nodes, as described below.

In the proposed network (which also served as a blueprint for the simplified broadcast network shown in the experiment), messages are organized into *channels*. Each channel is associated with a public key that needs to sign every message in that channel. Messages within a channel are given sequential numbers (*message IDs*) by the sender. A message is uniquely identified by the channel it belongs to and its ID within the channel. This is visualized in figure 4.1 in section 4.3.

This allows for much more efficient synchronization compared to assigning random identifiers to all messages, since in most cases, contiguous ranges of messages are exchanged.

Messages are exchanged whenever two nodes *meet*. This can happen using various methods, which can also be arbitrarily combined. The Bluetooth connections used in the experiment are just one example. For nodes connected to IP networks, TCP connections on that network could be used to meet. Consequently, such meetings can happen using the Internet - or local networks, long-distance fiber connections from universities, or whatever else is available.

Each time nodes meet, they have to agree on a shared set of channels they are both interested in (see section 7.5.2 for details). Afterwards, they exchange messages from these channels so that at the end of the connection, both devices have all messages that were present on either of the nodes (i.e. if node A has a set of messages M_A from the channels they have in common, and node B has a set of messages M_B, then after a successfully completed connection both of them have $M_A \cup M_B$, subject to the restrictions explained below).

Messages can be set to expire depending on the needs of the applications built on top of the network. For example, messages can be timestamped and set to expire after a certain time, they can expire when a certain number of newer messages was posted, or there can be explicit expiration messages that mark everything up to the point as expired (and possibly provide a summary). Expired messages will not be exchanged during meetings, but may be kept by nodes.

Other methods can be used to transfer messages: Broadcasting messages via amateur radio, satellite transmission or UDP broadcast/multicast can be used to quickly distribute a message to all nodes capable of receiving it. For nodes unable to connect to any network, messages can be stored on removable media (e.g. microSD cards) and physically transported to other nodes. All these transfer methods can be arbitrarily combined.

For example, a message could be written anywhere in the world and instantly distributed via the Internet to all nodes that have Internet access. One such node could package all messages daily onto a microSD card and send it to a remote town in a neighboring country where the government has disabled Internet access. A node in said remote town could read the messages from the card and distribute them locally via Bluetooth. As the messages slowly reach another node in that town via Bluetooth, this node relays them via WiFi broadcast to nodes in the vicinity, and sends a copy to a university in a nearby city using a direct dial-up connection. From there, it is distributed to all universities using the still intact inter-university network.

This makes the network highly resistant to censorship, since blocking one technology (e.g. by jamming Bluetooth signals) will not prevent the others from working. It will work quickly as long as Internet is available, as nodes can use dedicated servers that speed up distribution of messages. However, it will also work without the Internet using methods such as the ones described above. In such a situation, short-range Bluetooth/WiFi transmissions - as were explored in this experiment - will likely be the main method of local message dissemination, and can even provide long-range transport if enough people

travel and thus move message-laden nodes from city to city. Individually arranged long-distance/high-throughput links (e.g. via dial-up, dedicated lines, or physical media exchange) can further improve performance.

7.4 Feasibility

This section discusses the feasibility of such distributed systems, and shows that they are indeed feasible.

7.4.1 Group size

If the node density is too low, nodes may never meet, making any communication impossible. With an increasing node density, nodes meet more often, allowing messages to be distributed quickly. However, if the nodes are also active users of the system, more nodes increase the load on the system. Additionally, at some point, spectrum saturation becomes a problem and limits the network.

The experimental network was very sparse during most of the experiment. The Congress provided an opportunity to collect data in a situation with a relatively large node density.

To illustrate the impact of node density, a highly simplified model can be used. We assume a city with a population p, e.g. 1 million, some of which are users of a mobile peer-to-peer application. Each person has k random encounters, e.g. 15, per arbitrarily defined time unit. Note that these are k random individuals, *not* k users of the application! Transfers are always successful, but messages are not redistributed until the next time unit.

In this model, the probability of a user getting a message is equal to the probability to the user meeting at least one other user that has the message. The probability that the user encounters another user with the message in one encounter, is $\frac{m}{p}$, with m being the number of message-bearing users. The probability that the user does not encounter another user with the message in one encounter is thus $1-\frac{m}{p}$. For the k independent encounters, this probability is $(1-\frac{m}{p})^k$. Thus, the probability that the user does encounter at least one message-bearing user and gets the message in one time slot is

$$1-\left(1-\frac{m}{p}\right)^k$$

Let n be the total number of users within the population, and thus $n-m$ the number of users yet to receive the message. In each time slot, each of the $n-m$ users without the message have a chance of getting the message, resulting in

$$\Delta m = (n-m)\cdot\left(1-\left(1-\frac{m}{p}\right)^k\right)$$

new users getting the message on average. p and n are constants of the model. For a fixed number of users already having the message (m), n determines how many new users get the message in the next time slot. When $m << n$, the number of users getting the message (i.e. the speed of growth of m) scales roughly linearly with n. However, the growing m is itself included in the formula (in later time slots) - to the power of k.

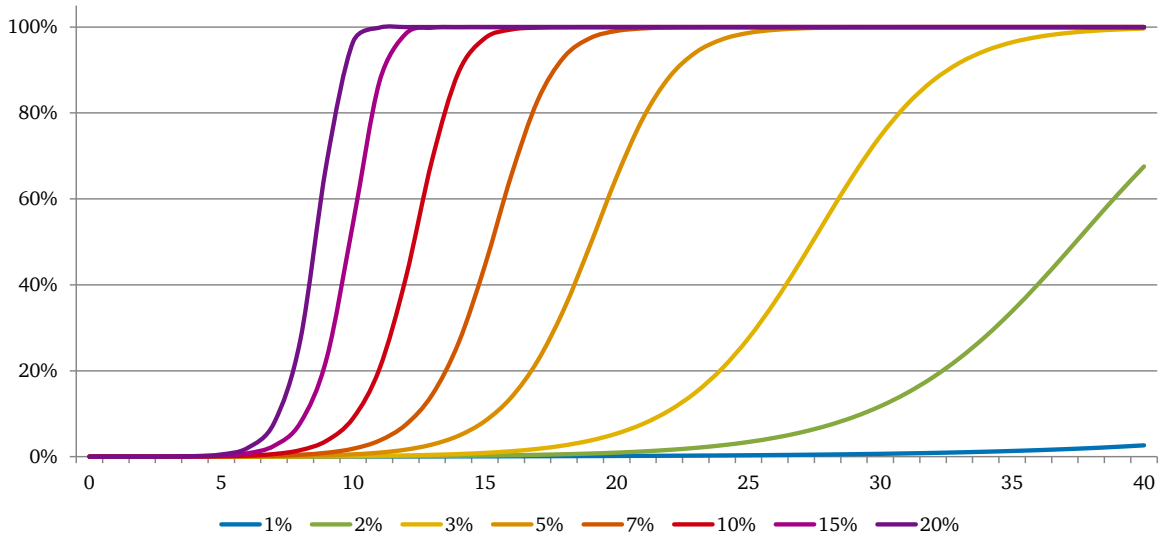

Figure 7.1.: Theoretical distribution of messages in the simplified model, using $p = 1$ million, $k = 15$, over 40 time units. The values on the y axis represent the fraction of nodes a message has reached at the given time. Each line represents one scenario where the indicated percentage of the population use the software (e.g. 1% in the lowest scenario, resulting in $n = 10,000$).

This becomes even more obvious if a situation where a given *fraction* of users has the message is considered. With 10% of the users having a message, i.e. $m = 0.1 \cdot n$, the formula is:

$$\begin{aligned}\Delta m &= (n - 0.1 \cdot n) \cdot \left(1 - \left(1 - \frac{0.1 \cdot n}{p}\right)^k\right) \\ &= 0.9 \cdot n \cdot \left(1 - \left(1 - \frac{0.1 \cdot n}{p}\right)^k\right)\end{aligned}$$

The factor on the left alone provides a linear increase with the size of the network. However, the size of the network is also included within the base, causing the number of nodes receiving the message to increase polynomially with n (with an order of k).

This can also be easily understood logically - the more users there are in total, the more opportunities for encounters between the message bearers and users there are, spreading the message faster. Additionally, the more message bearers there are, the higher the probability of a random user encountering one.

After one time slot, the new number of message-bearing users is:

$$\begin{aligned}m' = m + \Delta m &= m + (n - m) \cdot \left(1 - \left(1 - \frac{m}{p}\right)^k\right) \\ &= m + \left((n - m) - (n - m)\left(1 - \frac{m}{p}\right)^k\right) \\ &= n - (n - m)\left(1 - \frac{m}{p}\right)^k\end{aligned}$$

(i.e. the total number of users, minus the number of users who did not have the message and were unable to obtain it).

Of course, in reality, meetings are not random. Daytime will have more meetings than nighttime. Some people stay at home all day and meet no one. Users of public transit will have more useful encounters than users driving a car. People are much more likely to meet other people in their own area, peer group or workplace, than totally unrelated people who are a good distance away. On the other hand, some users may intentionally help transfer messages, e.g. by taking additional trips to meet other important nodes, setting up a manually managed hierarchy of messengers, placing static nodes in highly frequented places, or attaching nodes to vehicles.

Additionally, such a system can be augmented by using other means to transfer messages, e.g. special nodes that forward messages to other, distant nodes using remaining internet infrastructure, private communication lines, dial-up connections etc.

7.4.2 Message load

The amount of messages that will be carried by the system depends on many variables and is impossible to estimate accurately. To ensure that every node can access every message, all nodes would need to carry all messages in the proposed architecture. This makes the expected amount of messages crucial for the estimation of feasibility of the system. For this reason, multiple ways of estimating the expected amount of messages are provided in this section.

The alternative microblogging service/network identi.ca had 7.2M messages in the first six months of 2012, i.e. an average of around 34k messages per day. This was measured by comparing the sequential message ID of first messages of January and July (87899652 and 95053253). Most of these messages are spam.

Due to the decentralized nature of Usenet (each server carries different groups), it is impossible to collect precise statistics. The server “eternal-september.org”, which carries a typical selection of groups, receives around 100k to 200k messages[3] on most days.

Reddit, a large US-based social media platform with a worldwide user base, had 260 million comments in 2012, i.e. an average of 712k messages per day. [4] This includes all topics/sections and all countries. The largest “subreddit”, i.e. topic-specific group, had an average of 100k comments/day (one-week average, first week of June 2013). [5] Limiting the p2p based network to selected topics would be possible.

Based on these numbers, it seems sufficient if the system is able to handle about 100k messages per day even in adverse conditions (e.g. no internet).

Twitter announced on March 21 2012 that it handles 340M tweets per day from 140M active users [32]. Handling such a large amount of traffic would be difficult. However, it is not to be expected that the proposed system will ever reach such a popularity. Additionally, not all messages are relevant to all users. By splitting the network into smaller groups (e.g. by language), not all messages need to be distributed to all nodes, reducing the load on the nodes significantly. Group management is discussed in detail in section 7.5.2.

Messages on Twitter are limited to 140 characters (due to the use of Unicode, this may be more than 140 bytes). Messages on Reddit may be longer, but most stay well below 1 KB. Usenet messages vary in size, but are usually between 1 and 5 KB including headers and quotes and less than 2 KB of actual content. Some additional space is needed for metadata, including a digital signature (which can be e.g. 64 or 384 bytes, see section 7.5.1 for details). 1 KB is used as a reasonable assumption for the total size of one message.

[3] http://www.eternal-september.org/stats/index.html
[4] http://blog.reddit.com/2012/12/top-posts-of-year-and-best-of-2012.html
[5] http://stattit.com/subreddits/by_comments_avg_1d/

The system may be affected by intentional flooding. Being able to handle a larger amount of messages will provide more room for trade-offs. For example, it allows to make the network easier to access, even if this also increases the amount of flood messages that an attacker can generate.

Assuming 1% of the German population use the system, sending an average of 5 messages per day, would result in about 4M messages per day. However, it should be considered that the network can be split into sub-networks, reducing the number of messages that need to be distributed to all nodes. Designing the system so it can handle between 0.1M (100k) and 10M messages per day seems reasonable.

Technological advances will further increase the capacity of a network as the participating devices become more powerful. For this reason, the following estimates are based on a load of 100k messages per day, each 1 KB long.

7.4.3 Storage and bandwidth

Assuming an average message size of 1 KB[6] including metadata and digital signature, the total amount of message data generated per day, D_{day}, would be 100 MB.

Keeping old messages for k days would require $k \cdot D_{day}$ of storage. For $k = 10$, this would be 1 GB, which is easily possible on current smart phones (either using internal storage or a microSD card).

Every node needs to receive every message only once. Thus, the total incoming message traffic per node would be equal to the amount of message data D_{day}. Assuming unicast transfers, every message would need to be sent once to each participating node. Total message traffic that would need to be sent would thus be $n \cdot D_{day}$, with n being the number of nodes. Each node would thus on average need to send D_{day} of message traffic.

Additional bandwidth would be required for protocol overhead, including handshakes to determine which messages need to be exchanged.

Transfer rates observed during the experiment reached over 200 broadcast messages per second for the fastest connections, as shown in section 6.4.

Messages in the experiment consisted of 1024 bytes of dummy payload plus 12 bytes of message metadata, protocol overhead not included. At a transfer rate of 160 messages per second, this corresponds to a net data rate of 165 KB/s of message data. Transferring 100 MB of message data at this rate would take around 10 minutes. This means that a full set of messages for one day could be exchanged e.g. during the wait at a bus stop.

While connectivity will not always be as good as in this example, especially if the spectrum gets congested by many concurrent users, it clearly demonstrates the feasibility of such a network from a pure bandwidth aspect.

Using WiFi instead of Bluetooth could further increase the data rate. Using broadcast would further limit the amount of data that needs to be sent and reduce spectrum congestion, making the exchange of data significantly more efficient in both time and battery power required. Broadcast transmissions via Bluetooth are not possible on Android, however, WiFi Direct does support them. By applying forward error correction, the impact of packet loss can be limited, allowing highly efficient one-to-many data transmission.

[6] SI units, i.e. 1 KB $\bar{1}$000 byte

7.4.4 CPU

In most security schemes, messages will need to be signed for sender authentication. The main source of CPU load when receiving messages are the network/message parsing code, hashing of the messages, and the asymmetric signature verification itself.

Compared to asymmetric signature verification, the CPU cost of hashing small messages is negligible [33, 34]. As described below in section 7.5.1, it can be expected that several hundreds of messages per second can be verified on a mobile CPU. This means that depending on the CPU and chosen signature scheme, verification speed would likely be similar or slightly higher than the transmission speed observed above. Thus, all messages generated in the network during a day could be easily verified within several minutes.

If verification speed threatened to become a bottleneck, the node could resort to only verifying randomly chosen message and store the rest for later verification. Verifying some messages immediately would allow early detection of malicious nodes that send fake messages in order to waste resources. All other messages would be verified later, as available CPU time allows.

Thus, even significantly higher message loads than the above-mentioned 100k messages/day could be easily handled, and such a network would be feasible from the CPU aspect, too.

7.5 Design decisions

7.5.1 Signature algorithm

Since every message includes a signature, signature size is important to make efficient use of the limited storage and bandwidth that is available. Signatures must also allow fast verification, as nodes may have to verify the signature of every message they receive. A small public key size is useful, but not as important as the signature size. The signature creation speed is not important, since message creation/signing is rare compared to the other operation. Using a proven and popular signature scheme like RSA, DSA or ECDSA makes implementation easier, and decreases the probability of attacks against the signature scheme being discovered. A security level equivalent to 128 bits for symmetric algorithms will be used for the comparisons below. This corresponds to RSA and DSA key lengths of 3072 bits and ECC key lengths of 256 bits and is considered sufficient for 2031 and beyond by NIST [35].

RSA provides extremely fast signature verification times, especially when implemented using small public exponents. It outperforms [34, 36] even fast schemes like ECDSA and Ed25519 for a comparable security level. However, signature size is the same as the key length, i.e. 3072 bits (384 bytes) [35].

DSA and ECDSA provide a signature size of 512 bits (64 bytes) at this security level [35], with ECDSA outperforming DSA in signature verification speed and public key size [34]. Both DSA and ECDSA require secure randomness for signatures, increasing the potential impact of insecure PRNG implementations.

Innovative asymmetric schemes promise even better properties. For example, the Ed25519 scheme [37] promises batch verification at speeds comparable to (non-batch) RSA signature verification [34, 37], with a signature size of 64 bytes and public key size of 32 bytes (for the 128-bit-equivalent security level mentioned above). Additionally, no randomness is required for signature generation. Ed25519 is used in Serval/MeshMS.[3] However, innovative schemes like Ed25519 have received less public scrutiny than commonly-used ones like RSA and ECDSA.

Ultimately, the choice of algorithm is with the designer of such a system. The benchmark values from a single-core 1000 MHz Cortex A8 CPU show that a RSA verification takes ~1.1 megacycles [34]. This

means that the CPU is able to perform ~900 RSA verifications per second. ECDSA verification is slower by approximately a factor of five. It is thus safe to assume that current mobile CPUs can verify several hundred of messages per second, with the exact numbers depending on the CPU and the chosen algorithm. A simple benchmark using OpenSSL on a Galaxy Nexus and a Nexus S showed 120 and 130 ECDSA-p256 verifications per second, respectively.

7.5.2 Subscription management

In smaller networks with non-malicious participants, it is feasible to simply distribute any message to everyone. As soon as the network has to defend against malicious participants attempting to flood the network, some limitations need to be imposed. Simply limiting the number of messages a participant can send is insufficient if an attacker can simply create an unlimited number of fake identities (Sybil attack [38]).

A simple approach, especially if the network is used for a microblogging application, would be to only subscribe to the manually-selected channels of "friends". However, this would strongly limit distribution - messages from a sender would only be distributed through his friends, not any other network nodes. If a sender never met his subscribers directly, his messages would not be distributed at all.

For this reason, some kind of subscription management needs to be implemented.

The easiest way to do this is to provide central membership management. With this concept, a central entity decides which nodes are permitted to participate in the network. It then provides credentials (e.g. certificates) to them, or releases a message that these nodes (represented by their keys) are authorized to participate. Participating nodes subscribe to the channels signed with these keys, thus accepting and distributing the messages in them. This also makes it possible to use short IDs for the channels, since collisions can be prevented. Less centralized approaches would likely need to derive the channel IDs from public keys and make them significantly longer to make collisions not only unlikely, but also infeasible to create intentionally for an attacker.

Certificates would be provided by the sending node together with the first message of a channel unknown to the receiving node. Thus, any node with any message from a channel would have the corresponding certificate to prove that this channel deserves distribution. Alternatively, nodes would synchronize a "master channel" containing authorization messages from the central authority before synchronizing normal messages. At the end of the master channel synchronization, both nodes would have the same (but not necessarily complete) list of approved channels. This could also be used to optimize the actual synchronization, since channel IDs would not need to be explicitly transmitted. A "master channel" approach would also allow removal of misbehaving nodes by the central authority, by posting an un-approval/revocation message to the master channel. The central authority could also designate additional channels as "moderator channels", possibly delegating some of its powers to them. Thus, the actual power to decide over the network could be distributed across multiple different organizations.

The number of messages sent in a certain timeframe can be limited per channel to limit the impact of flooding attacks. By limiting the number of fake accounts/channels the attacker can get (through central authorization of accounts), and limiting the number of messages per channel, the maximum bandwidth of the attack has an upper bound. It is unnecessary to prevent all attacks as long as the attacks that remain possible are so limited that the network can handle the additional load.

This approach can be further decentralized by replacing the central entity with some form of collaborative, distributed process (e.g. through the use of distributed signature schemes). Such distributed admission controls have been extensively researched in the context of MANETs, and are out of scope for

this thesis. The broadcast network itself can provide a reliable transport for messages of already admitted nodes.

User admission could also be controlled based on a web-of-trust, where users manually confirm other users. Messages could then be prioritized based on the strength of the (indirect) trust between the node operator and the sender. A similar approach is described in [1], which also explains why this is effective at preventing agents of an oppressive government from gaining too much influence on the network. However, any web-of-trust based approach has to be carefully evaluated, since it may reveal information about the social network of participants. This could be extremely dangerous especially in totalitarian governments.

Even if effective admission control exists, local attacks on the network will always remain possible. By initiating large numbers of communication attempts and/or forcing many failed proof-of-work verifications, batteries can be drained. This can be countered by detecting devices that display bad behavior and refusing further communications based on existing identifiers (e.g. MAC or Bluetooth addresses). However, these identifiers can be spoofed. Other local attacks, e.g. radio jamming, simply cannot be prevented. However, since such attacks are localized, this does not affect the functioning of the network as a whole.

Approaches based on proof-of-work

A proof of work consists of calculating a hard-to-calculate value (thus spending *work*) in a way that a third party can *easily* verify that the work was spent. Finding partial hash collisions is a popular method to implement this and in fact recommended in the original paper [38] on the Sybil attack. To create a proof-of-work for a given value k, for example a public key to be authorized or the hash of a message to be distributed on the network, the proving party has to find a value n so that the binary representation of $hash(k, n)$ starts with a certain number t of zeros. This number t defines the *difficulty*, i.e. the amount of work required (on average).

Since a cryptographically strong hash function is used, the only way to do this is to pick an arbitrary number n, calculate $hash(k, n)$, and repeat with changing values of n until a value matching the condition is found. The distribution of hash values is random for practical purposes in cryptographically strong hash functions. Thus, the prover will need to perform 2^t hash calculations on average to obtain a valid proof of work. The verifying party only has to perform a single hash calculation to verify the value n.

The actual work spent on a certain proof of work is randomly distributed (since the first value for n could be a match). This is not a problem issue, since an attacker needs to calculate many proofs to confirm a large number of actions (e.g. creation of a channel) to successfully overload the network.

A proof-of-work concept could allow fully decentralized self-admission, while still limiting the amount of fake identities an attacker can create.

To join the network, the participant would create a key pair, then calculate a proof-of-work for the public key and use it as a “certificate”, showing that channels signed by the key should be allowed to participate in the network. For example, the difficulty would be adjusted that on average, one day of calculations with a current computer is needed to join the network. And an additional proof of work could be required every month to maintain membership.

A powerful attacker with significant computing power can still create a *large* number of fake identities, but can no longer create an *unlimited* number them. Thus, if the network has enough bandwidth available to tolerate the load caused by this number of fake identities, the proof-of-work approach can be sufficient.

However, this will make it difficult for users to join the network, especially in poorer countries where modern IT hardware is not common. Mobile phones, for example, could not meaningfully join the network, since computations designed to take a day on a computer would take weeks or months on a less powerful mobile CPU. Choosing the proof of work difficulty so that a mobile phone can calculate it in a day would in turn allow an attacker with access to a computer to create dozens of false identities per day on a single modern computer. While legitimate users could calculate a proof of work on a computer and transfer the key to a mobile phone, not all users may have access to a computer. As described in [38], an attacker that has n times as many resources as the *weakest* node able to join the network is able to create at least n fake identities.

Additionally, the hash-based proof-of-work scheme described above can be calculated very efficiently using GPUs or ASICs. Thus, an attacker able to purchase dedicated hardware can outperform legitimate users by many orders of magnitude, further increasing the potential number of fake nodes. Some proof of work schemes (based e.g. on scrypt [39]) are designed to be less susceptible to this problem, but cannot remove the disparity between common general-purpose hardware and specialized hardware completely.

For these reasons, using proof-of-work schemes is likely not an effective solution on its own. However, proof of works can be easily applied to individual messages, instead of being used to authorize channels/identities. This can be used to create two separate broadcast networks, running in parallel on participating nodes: One of them using centralized channel authorization, and the other one using (hard) per-message proof-of-works. The latter could be used by non-member nodes to communicate with the central authorization point, with messages being prioritized by the difficulty of the provided proof of work.

An attacker could not disrupt the main network (assuming effective centralized admission control) and thus could not disrupt communication between nodes already using the network. The admission network would accept as much load as it could. The attacker might be able to overload it and prevent messages from real nodes from passing through. However, this would require him to spend significant computational resources.

A better approach for communication of non-admitted nodes would likely be to let an already admitted node to forward the message after manual confirmation by the user. This way, a new participant would only need to find a current participant willing to forward his message.

7.6 Potential applications

The network, as described above, would provide a reliable, authenticated set of one-to-many/one-to-all broadcast channels. A number of useful applications could be built on top of such a service.

7.6.1 Twitter-like microblogging (one-to-many communications)

The most obvious application would be a Twitter-like microblogging service, since channels can be directly mapped to the message stream of a user. To allow confidential communications, public keys for encryption could be included in user IDs. A message meant to be read by a single user could then be encrypted to this user's key, then posted to the message stream of the sender. A message to a group of users could be sent by encrypting it with a symmetric key, and adding a key management message which contains one entry for each group member. Each entry would contain the symmetric key, encrypted to the public key of the corresponding member.

Messages would only reach nodes that subscribe to the channel of the sender. This actually matches the behavior on Twitter, where direct messages can only be sent to followers. Since users would need to automatically subscribe to channels of unrelated users (possible *all* users of the network, or all users in a certain region) to ensure efficient distribution (see section 7.4.1), it would be possible to allow sending messages to users who do not *actively* subscribe.

Any such kind of one-to-many communication is perfectly suited for this kind of network.

7.6.2 E-mail-like one-on-one communication with reduced network load

With sufficiently large and possibly overlapping groups, this allows worldwide e-mail like messaging. The message itself is encrypted to the recipient, possibly using a scheme where it is impossible to identify the recipient from the message without having his key. (This would - at least in a naive implementation - require every node to attempt to decrypt every message, but would mostly prevent traffic analysis). The encrypted message would then be distributed using the broadcast network. In the simplest approach, every node subscribes to every other node's channels and thus distributes all messages.

If bandwidth is not sufficient for this, sufficiently large (and possibly regional) groups can be created, which could be overlapping (i.e. have nodes which are members in multiple groups). This way, messages would not need to be distributed to every node and would still reach the recipient quickly (assuming the groups are large enough to ensure fast distribution).

A simple example shows how this could work. It is assumed that the network is so large that a message will get quickly distributed to every interested node even if only 10% of the nodes participate in the distribution of this message. Every node is assigned a random group number from 00 to 99 (e.g. by hashing the node ID). Messages are tagged with the group numbers of sender and recipient. Since groups are large, this does not allow an attacker to perform meaningful traffic analysis. Messages are distributed by every node whose first digit of the group number matches the first digit of the sender's group number. Thus, a message sent by a node in group 21 gets distributed by all nodes in groups 20, 21, 22, ..., 29 - which form 10% of the network.

If the recipient is not in one of these groups, every node whose *second* digit of the group number matches the second digit of the *recipient's* group number distributes the message, too. Thus, if the message is sent by a node in group 21 to a node in group 76, all nodes in groups 20, 21, 22, ..., **26**, ..., 29 as well as 06, 16, **26**, ... 96 will distribute the message. As per our assumption that 10% of the network will quickly distribute the message, the message will quickly reach most nodes of group 26 (as well as all other "2x" groups, which together form 10% of the network). From there, it will be distributed again amongst all "x6" groups, which together form a different 10% section of the network, and include the recipient. See Figure 7.2 for a visualization of this concept.

Much more elaborate and efficient schemes than this simple example could be used - yet even this simple example manages to reduce network traffic by over 81%, since messages get distributed to only 19 of the 100 equally-sized groups of the network at most (only 10 if the recipient and sender share the first digit of the group number).

Grouping based on geographic or topological location could be used for optimizations. For example, messages where sender and recipient are in the same zone (however a zone would be defined) could be limited to nodes in the zone. Messages across zones could be either transported by nodes that are members of both zones (e.g. someone who travels between two cities for zones defined geographically), or by using a third exchange zone: The message would be tagged with three zones (sender zone, recipient zone, exchange zone), quickly distributed in the first zone, reaching a travelling node subscribing to the

Second digit \ First digit	0	1	2	3	4	5	6	7	8	9
0	00	10	20	30	40	50	60	70	80	90
1	01	11	21	31	41	51	61	71	81	91
2	02	12	22	32	42	52	62	72	82	92
3	03	13	23	33	43	53	63	73	83	93
4	04	14	24	34	44	54	64	74	84	94
5	05	15	25	35	45	55	65	75	85	95
6	06	16	26	36	46	56	66	76	86	96
7	07	17	27	37	47	57	67	77	87	97
8	08	18	28	38	48	58	68	78	88	98
9	09	19	29	39	49	59	69	79	89	99

Figure 7.2.: Using broadcast groups to limit network traffic

exchange zone, and be quickly distributed in the entire destination zone once any exchange zone member brought it there.

All of these can limit the total amount of nodes involved in the transfer of a message, thus reducing the load on the network and allowing for more messages to be transported. However, they involve trade-offs in complexity, limited distribution speed, risk of traffic analysis, and the risk of missing out fast paths that do not match the zones.

For example, assume a message should be sent within a city in a country where internet access is no longer available. The fastest way to do so may be to relay it locally to a neighbor operating a dial-up link to the outside world, send the message over that link, let other (internet-connected) nodes distribute it, and have a friend of the recipient download it via another dial-up link. With geographically defined zones, the message would not cross the dial-up link and thus have to rely on possibly slower short-range radio distribution. On the other hand, this would keep the dial-up links free for messages that have to cross the border.

A sensible compromise could be to use such mechanisms not to limit distribution, but to prioritize messages.

7.6.3 Message boards and discussion groups (many-to-many communications)

Message boards and discussion groups (similar to Usenet groups) can be easily and efficiently implemented by treating them as a set of many one-to-many communications. Every participant of a discussion group subscribes to all channels of all other participants. Since the broadcast network already delivers every message to every node in the group, many-to-many communication is not more expensive than one-to-one communication.

All nodes participating in a discussion group should share a broadcast group, but a broadcast group could consist of as few or as many discussion groups as needed. This way, the size of the broadcast group can be adjusted to contain enough participants for quick distribution of messages, but can be kept small enough to limit resource use.

7.6.4 Social networks

Social networks can also be built on top of a broadcast network by using the original one-to-many approach. Instead of only broadcasting written messages, every action (e.g. likes, ratings/votes, shares, ...) would be broadcast. The recipient would then need to analyze the actions of all users relevant to the content to display the result (e.g. calculate the total number of likes). Actions could be packed into packages sent at regular intervals to avoid having to sign and broadcast a separate message for every minor action.

If certain actions should only be visible to certain people (e.g. friends), they can be encrypted just like one-to-one messages, as described above. Encryption can be optimized by introducing group key management, where a symmetric key is encrypted to the public key of every friend, the resulting package of encrypted keys is broadcast as a message, and subsequent messages to this group are encrypted with the symmetric key, then broadcast.

Broadcast group management could be based on social ties (this is risky as described in section 7.5.2), or by splitting the user base into appropriately sized groups. (Users need to subscribe to any group that contains any friend or other person from whom they want to see messages or actions - this must be taken in account when selecting group size).

Discussion platforms where users can rate contributions can be implemented the same way. However, dishonest users or attackers could refuse to relay ratings/votes they do not agree with, while quickly distributing votes they do agree with. This would not prevent the real votes from finally coming through (after all, such dishonest users would be nothing else than an attacker attempting to exert censorship, which the network is designed to handle). However, the delay would skew the votes, potentially causing the content to be less (or more) visible. This has to be taken into account when implementing systems where the visibility of content depends on the votes of users. If votes are only visible to friends and encrypted for everyone else, it is much more difficult for an attacker to determine what kind of vote he is currently distributing, limiting the effectiveness of this attack.

Any such feature would greatly increase the message load and would likely exceed the capacity achievable in a distributed mobile system with today's technology.

7.6.5 Multiple services

The infrastructure used by the broadcast network does not need to be limited to one of the above services. Different services can share the infrastructure and possibly use some shared services (e.g. authorization), while keeping their own logical network with its own group management for the actual messaging.

For example, a user would have separate channels for microblogging and for discussion. Users that do not use microblogging, but do participate in the discussion group, would only subscribe to the latter, controlled by the discussion system's group management. This way, group size and thus resource use can be limited. However, it would be also possible to distribute messages across services. Services with a low number of users would likely have joint group management with other services to increase group size and thus distribution.

By using separate logical networks, services could have different policies, e.g. for the maximum message size, the rate at which users can send messages, and for discarding old messages. For example, a service that provides the current temperature would only retain and distribute the most current message, while a discussion service might want to retain all messages younger than 30 days.

8 Conclusions and outlook

This thesis shows that mobile peer-to-peer networks are feasible, even though they face serious challenges. The biggest challenges are limitations of Bluetooth software stacks.

Once deployed, mobile peer-to-peer networks can provide a variety of services - from public microblogging to one-on-one communication. They can do so without relying on existing infrastructure, and as such, can be a valuable tool for emergency communication during natural disasters, everyday communication in underdeveloped areas, or communication amongst people living under oppressive governments. This thesis shows that the broadcast approach is feasible and can be used to implement these services. Both the concepts to do so and the pitfalls - including ones that can potentially endanger lives of activists - are explained.

The spread of the messages during the Congress demonstrates that given a sufficient node density, even primitive implementations of broadcast-based mobile peer-to-peer networks can distribute messages to a large fraction of active nodes. In the experiment, distribution speed was not comparable to infrastructure-based networking, but could have competed with old media like postal mail and newspapers. Given the intentionally simplistic nature of this implementation, the self-imposed limitations (e.g. no WiFi, purely peer-to-peer based architecture), the issues with the underlying software, and the congested spectrum at the Congress, this is remarkable. Future, practical networks can be expected to achieve significantly better performance. Some optimizations are suggested in section 4.3.

The SQL-based approach to log analysis has proven to be very convenient, fast, and surprisingly powerful. While some analysis still requires custom code, most can be done purely using SQL queries. This reduces both development and execution time significantly. The step-by-step example in appendix A provides an introduction for researchers unfamiliar with this approach.

8.1 The data set

Most importantly, however, this thesis provides a powerful and versatile foundation for future research in the form of a rich data set. This data was collected in the real world on devices found “in the wild”, and provides information ranging from device capabilities to Bluetooth encounters to detailed logs of real data transmissions. The data set can be used directly for analysis on different levels, from comparing Bluetooth connection performance vs. SNR for different devices to observing patterns of message distribution in the simulated broadcast network. However, it can also be used as a basis for simulations - for example, the observed delays and transfer rates can be used to improve accuracy when modeling a totally different type of network.

Due to the nature of this experiment, the data is highly realistic for networks deployed on existing, heterogeneous devices which the users already have. Other data sets, collected using a homogenous set of devices issued specifically for the experiment, miss many issues that can be seen in such a network. The collected information about these issues allows to mitigate and avoid them in future experiments and real networks. Since many of these issues only occur on certain devices, they can be impossible to find in limited testing without such information. These issues also show an important limitation of simulations: Even the best simulation is likely to miss these, leading to results that fail to transfer to the real world.

8.1.1 Limitations

On the other hand, the issues described above make the resulting data less applicable to scenarios with homogenous devices, and/or future devices that are not affected by such issues. Unlike in some other data sets, no location information is included. This makes the data harder to understand, but makes it possible to distribute the dataset freely. Other types of data that might be useful but were not collected for privacy reasons are listed in section 4.9. Accurate timing information is not available for all of the events, and not all devices uploaded all logs. A high node density is only available for a period of several days during the Congress, and node density in the remaining time may be too low for some purposes.

With these limitations in mind, the author hopes that this data set will enable new research in the field of mobile peer-to-peer networking. The data set is designed to be universally useful both within and beyond this field.

Since the focus of this thesis is collecting and providing the data set, only limited analysis is performed.

8.2 Future research

Various examples of possible research are given throughout this chapter, but any kind of research that needs any subset of the data can benefit from this data set. Bluetooth visibility data is often used to simulate the probability of successful connections. With this data set, real information about connection attempts and performance of connections is available in addition to that. This can be used to either replace Bluetooth visibility data for this purpose, or to improve the models used in these simulations.

Researchers wishing to perform their own data collection should pay special attention to chapter 5. It provides both a “lessons learned” section and documents the issues that were encountered.

When using the data set, the privacy of the participants should always be respected. Research should not single out individual users, and under no circumstances should attempts to re-identify users be made.

It follows from the observations made in this thesis that a network is unlikely to behave as expected when deployed on a set of heterogeneous devices. The only way to obtain trustworthy results for such scenarios is deploying it on real devices. Simulations can be useful as a tool to provide information for network design, if all their limitations are considered. However, they cannot prove the capabilities of mobile peer-to-peer network systems, and should not be used to evaluate such systems. Luckily, this thesis has shown that finding volunteers for a real test network is perfectly possible, even with limited resources and even if no incentives can be provided.

The broadcast network used in the experiment was intentionally kept very simple. A real broadcast network needs practically validated algorithms for message and peer selection. More efficient ways to communicate using existing hardware and APIs need to be designed, and new technologies, like Bluetooth LE, Bluetooth HS and WiFi Direct are waiting to be explored.

The data collected in this experiment can also be used to test the validity of simulations. This can be done by attempting to model the real network as accurately as possible, and comparing the simulation results to the actual results.

Further analysis of the interactions between nodes can and should be performed, as is commonly done with other data sets (see e.g. [40]). Outside of computer science, the possible impact of such networks, and acceptable performance properties (e.g. message propagation delay, reliability of message delivery) are topics worthy of further research.

8.3 Outlook

Over time, the challenges observed are likely to disappear. Bluetooth stacks will become more stable, and problematic older devices will be replaced by newer ones. Other technologies that provide even better performance (like WiFi Direct) are already on the rise. At the same time, mobile devices become more and more powerful and increasingly ubiquitous even in less developed areas. These improvements will make mobile peer-to-peer networks even more practical - and important - than they are today.

While research on the topic is important, what matters in the end are real implementations that have impact on people. Reliable, well-tested and well-reviewed applications that support peer-to-peer communication need to be developed. Peer-to-peer networks will likely not be able to compete with infrastructure-based networks, and users are unlikely to choose an application just because of the technology if it does not fulfill their needs well. For this reason, peer-to-peer support should be built into regular applications, using infrastructure networks while available, and transparently falling back to peer-to-peer if infrastructure networks fail. The peer-to-peer component has to be constantly tested on real devices on a large scale. Otherwise, unexpected issues will likely prevent it from working when it is most needed. Including censorship resistance in mainstream software also makes it less likely that users can be singled out and persecuted.

Well-working mobile peer-to-peer networks have the potential to change lives.

Errata, data set download and supporting data
A web site with errata, the generated data set, useful information about its use and a free electronic copy of this thesis will be maintained at:
`http://www.janschejbal.de/permanent/masterthesis`

Bibliography

[1] Giulia Fanti, Yahel Ben David, Sebastian Benthall, Eric Brewer and Scott Shenker. Rangzen: Circumventing Government-Imposed Communication Blackouts. Technical Report UCB/EECS-2013-128, EECS Department, University of California, Berkeley (Jul 2013).
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-128.html

[2] Paul Gardner-Stephen. The Serval Project: Practical Wireless Ad-Hoc Mobile Telecommunications (2011).
http://developer.servalproject.org/files/CWN_Chapter_Serval.pdf

[3] Paul Gardner-Stephen, Jeremy Lakeman, Romana Challans, Corey Wallis, Ariel Stulman and Yoram Haddad. MeshMS: Ad Hoc Data Transfer within Mesh Network. *Int'l J. of Communications, Network and System Sciences*, 5(8):496–504 (2012). doi:10.4236/ijcns.2012.58060.
http://dx.doi.org/10.4236/ijcns.2012.58060

[4] The Serval Project - 0.91 Upcoming release: "Glossy".
http://www.servalproject.org/archives/1104

[5] The Serval Project Wiki - Supported Devices.
http://developer.servalproject.org/dokuwiki/doku.php?id=content:servalmesh:supported_devices

[6] Paul Gardner-Stephen. Shopping Serval Style.
http://servalpaul.blogspot.de/2011/12/shopping-serval-style.html

[7] Alex McMahon and Stephen Farrell. Delay- and Disruption-Tolerant Networking. *IEEE Internet Computing*, 13(6):82–87 (2009). ISSN 1089-7801. doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2009.127.

[8] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall and H. Weiss. Delay-Tolerant Networking Architecture. RFC 4838 (Informational) (April 2007).
http://www.ietf.org/rfc/rfc4838.txt

[9] Jay Wyatt, Scott Burleigh, Ross Jones, Leigh Torgerson and Steve Wissler. Disruption Tolerant Networking Flight Validation Experiment on NASA's EPOXI Mission. In *Proceedings of the 2009 First International Conference on Advances in Satellite and Space Communications*, SPACOMM '09, pages 187–196. IEEE Computer Society, Washington, DC, USA (2009). ISBN 978-0-7695-3694-1.

[10] A. Lindgren, A. Doria, E. Davies and S. Grasic. Probabilistic Routing Protocol for Intermittently Connected Networks. RFC 6693 (Experimental) (August 2012).
http://www.ietf.org/rfc/rfc6693.txt

[11] Hervé Ntareme, Marco Zennaro and Björn Pehrson. Delay Tolerant Network on smartphones: Applications for communication challenged areas. In *Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition*, page 14. ACM (2011).

[12] Satoshi Nakamoto (pseudonym). Bitcoin: A Peer-to-Peer Electronic Cash System.
https://bitcoin.org/bitcoin.pdf

[13] Bitcoin Protocol specification.
https://en.bitcoin.it/wiki/Protocol_specification

[14] Nokia Research Center. Lausanne Data Collection Campaign.
https://research.nokia.com/page/11367

[15] Nokia Research Center. LDCC method.
https://research.nokia.com/page/11388

[16] N. Kiukkonen, J. Blom, O. Dousse, D. Gatica-Perez and J. Laurila. Towards Rich Mobile Phone Datasets: Lausanne Data Collection Campaign. In *Proceedings of the ACM International Conference on Pervasive Services (ICPS)* (2010).
http://www.idiap.ch/~gatica/publications/KiukkonenBlomDousseGaticaLaurila-icps10.pdf

[17] Nathan Eagle and Alex (Sandy) Pentland. Reality Mining: Sensing Complex Social Systems. *Personal Ubiquitous Comput.*, 10(4):255–268 (March 2006). ISSN 1617-4909. doi:10.1007/s00779-005-0046-3.
http://realitycommons.media.mit.edu/pdfs/realitymining.pdf

[18] Hamed Soroush, Nilanjan Banerjee, Mark Corner, Brian Levine and Brian Lynn. A Retrospective Look at the UMass DOME Mobile Testbed. *SIGMOBILE Mob. Comput. Commun. Rev.*, 15(4):2–15 (March 2012). ISSN 1559-1662. doi:10.1145/2169077.2169079.
http://forensics.umass.edu/pubs/soroush.mc2r.2011.pdf

[19] Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese and Christophe Diot. MobiClique: Middleware for Mobile Social Networking. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 49–54. ACM, New York, NY, USA (2009). ISBN 978-1-60558-445-4. doi:10.1145/1592665.1592678.
http://conferences.sigcomm.org/sigcomm/2009/workshops/wosn/papers/p49.pdf

[20] Yu Zheng, Lizhu Zhang, Xing Xie and Wei-Ying Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 791–800. ACM, New York, NY, USA (2009). ISBN 978-1-60558-487-4. doi:10.1145/1526709.1526816.
http://research.microsoft.com/apps/pubs/default.aspx?id=79440

[21] David Cavin, Yoav Sasson and André Schiper. On the accuracy of MANET simulators. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 38–43. ACM (2002).
http://infoscience.epfl.ch/record/49940/files/CSA02b.pdf

[22] Stuart Kurkowski, Tracy Camp and Michael Colagrosso. MANET Simulation Studies: The Incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61 (October 2005). ISSN 1559-1662. doi:10.1145/1096166.1096174.
http://www.cs.cornell.edu/People/egs/cornellonly/syslunch/spring06/papers/evaluation.pdf

[23] Sameer Singh. Smartphone Market Share by Country - Q3 2013.
http://www.tech-thoughts.net/2013/11/smartphone-market-share-by-country-q3-2013.html

[24] Kantar World Panel. Smartphone OS Market Share.
http://www.kantarworldpanel.com/smartphone-os-market-share/

[25] R. Friedman, A. Kogan and Y. Krivolapov. On Power and Throughput Tradeoffs of WiFi and Bluetooth in Smartphones. *Mobile Computing, IEEE Transactions on*, 12(7):1363–1376 (July 2013). ISSN 1536-1233. doi:10.1109/TMC.2012.117.

[26] David Eppstein, Michael T. Goodrich, Frank Uyeda and George Varghese. What's the Difference?: Efficient Set Reconciliation Without Prior Context. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 218–229. ACM, New York, NY, USA (2011). ISBN 978-1-4503-0797-0. doi:10.1145/2018436.2018462.
https://www.ics.uci.edu/~eppstein/pubs/EppGooUye-SIGCOMM-11.pdf

[27] Android Open Source Project. Android API Documentation.
http://developer.android.com/reference/

[28] Personal communications with Stephan Urbach (human rights activist with contacts in totalitarian countries, formerly active in Telecomix), November 2013.

[29] Chris Anderson. BASICS.TXT – A tutorial for newcomers and prospective members of FidoNet, Rev. 0.3.
http://www.skeptictank.org/files/aj/basics03.htm

[30] FidoNet Policy Document, Version 4.07 (June 1989).

[31] Ronda Hauben. On The Early Days of Usenet.
http://www.columbia.edu/~rh120/ch106.x10

[32] Twitter. Twitter turns six.
https://blog.twitter.com/2012/twitter-turns-six

[33] eBACS: ECRYPT Benchmarking of Cryptographic Systems - Measurements of SHA-3 finalists.
http://bench.cr.yp.to/results-sha3.html

[34] eBACS: ECRYPT Benchmarking of Cryptographic Systems - Measurements of public-key signature systems.
http://bench.cr.yp.to/results-sign.html

[35] Elaine Barker, William Barker, William Burr, William Polk and Miles Smid. Recommendation for Key Management - Part 1: General. In *NIST Special Publication 800-57, August 2005, National Institute of Standards and Technology.* (2005).
http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

[36] Nicholas Jansma and Brandon Arrendondo. Performance comparison of elliptic curve and RSA digital signatures.
http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf

[37] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe and Bo-Yin Yang. High-speed High-security Signatures. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'11, pages 124–142. Springer-Verlag, Berlin, Heidelberg (2011). ISBN 978-3-642-23950-2. http://ed25519.cr.yp.to/.
http://ed25519.cr.yp.to/ed25519-20110705.pdf

[38] John R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260. Springer-Verlag, London, UK, UK (2002). ISBN 3-540-44179-4.
http://research.microsoft.com/pubs/74220/IPTPS2002.pdf

[39] Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions.
https://www.tarsnap.com/scrypt/scrypt.pdf

[40] Lorenzo Isella, Juliette Stehle, Alain Barrat, Ciro Cattuto, Jean-Francois Pinton and Wouter Van den Broeck. What's in a crowd? Analysis of face-to-face behavioral networks. *CoRR*, abs/1006.1260 (2010). Informal publication.
http://dblp.uni-trier.de/db/journals/corr/corr1006.html#abs-1006-1260

Appendices

A Log analysis example using SQL

This appendix demonstrates how to extract some of the data needed for the charts in section 6.3.

NTP timing data is not always available, and system clocks can be inaccurate. For this reason, NTP data is used to determine which system clocks can be considered to be likely accurate, and their data is then used to assign times to events. Do to this, first, a helper table to measure the accuracy of system clock information is created as follows:

```
CREATE TABLE extract_ntptime AS
  SELECT device, log, line, wallclock, uptime, tag,
    CAST(c0 as integer) as ntptime, CAST(c1 as integer) as timeref, CAST(c2 as integer) as roundtrip,
    CAST(c0 as integer)- (wallclock - (uptime-CAST(c1 as integer))) as walldiff,
    CAST(c0 as integer)- CAST(c1 as integer) AS updiff
  FROM log_readable
  WHERE tag = 'NTPTIME';

CREATE TABLE helper_ntp_accuracy AS
    SELECT device, MIN(ntptime) as ntp_min_ntptime, MAX(ntptime) AS ntp_max_ntptime,
        MIN(walldiff) AS ntp_min_walldiff, MAX(walldiff) AS ntp_max_walldiff, COUNT(*) as ntp_count,
        MAX(ntptime)-MIN(ntptime) AS ntp_span,
        MAX(ABS(MIN(walldiff)),ABS(MAX(walldiff)))/60000 AS ntp_minutes_of_inaccuracy
    FROM extract_ntptime
    WHERE roundtrip > 0 AND roundtrip < 10000 GROUP BY device;
```

The query calculates, for each device, the minimal and maximal reported NTP time, the time span covered by them, the minimal and maximal system clock difference (walldiff), as well as the number of NTP events, the time span between the first and last event, and the maximum difference between the system clock and NTP time in minutes (rounded down). NTP events with invalid (< 0) or too large round trip times are ignored.

Now, some message-related log entries are extracted. Only the creation of messages and the receipt of messages are considered, and the full log rows are extracted unchanged. In a second step, timing accuracy information is added:

```
CREATE TABLE extract_messaging AS
  SELECT * FROM log_readable
  WHERE (tag = 'BroadcastManager' AND c0 = 'creating␣message') OR
      (tag = 'BroadcastProtocolConnection' AND c1 = 'Received' AND c2 = 'PAYLOAD');


CREATE TABLE extract_messaging_with_timing AS
  SELECT *, (ntp_min_ntptime-60000) < wallclock AND wallclock < (ntp_max_ntptime+60000) AS within_ntp
  FROM
    extract_messaging
  LEFT JOIN
    helper_ntp_accuracy
  USING (device);
```

If the log with the “message received” event is lost, the fact that the node obtained the message will not be visible using the resulting data. By analyzing the BroadcastStates, which are periodically logged, such messages could be found. However, doing so is impractical in SQL. This demonstrates one of the limitations of SQL-based log analysis.

The logs are split further into creation and receiving (using the fact that the creation log lines do not have a value in column c3) and the relevant information is extracted:

```
CREATE TABLE tmp_messaging_receive AS
  SELECT wallclock, within_ntp, ntp_minutes_of_inaccuracy, device, c3 AS channel, c4 AS message
  FROM (SELECT * FROM extract_messaging_with_timing WHERE c3 IS NOT NULL);

CREATE TABLE tmp_messaging_create AS
  SELECT wallclock, within_ntp, ntp_minutes_of_inaccuracy, device, SUBSTR(device, 5, 16) AS channel, c1 AS message
  FROM (SELECT * FROM extract_messaging_with_timing WHERE c3 IS NULL);
```

The following query now calculates the desired statistics. The inner query is sufficient to return the number of nodes that received the message at some point in time. By joining it with information about the creation of the message, information about the creation time (and its reliability) is added:

```
CREATE TABLE results_messagespread AS
  SELECT datetime(wallclock/1000,'unixepoch') AS human_create_time, *
  FROM
    (SELECT channel, message, COUNT(DISTINCT device) AS carriers FROM tmp_messaging_receive GROUP BY channel, message)
  LEFT JOIN
    tmp_messaging_create
  USING (channel,message)
  ORDER BY human_create_time;
```

The data can now be converted to a format suitable for charting, while selecting only rows with accurate time information. Since plotting individual data points is not feasible due to their large number, the number of messages that reached a certain number of nodes is calculated for each day. This data can then be exported and plotted using gnuplot.

```
CREATE TABLE results_messagespread_chartable AS
  SELECT wallclock/1000 AS unix_wallclock, carriers
  FROM results_messagespread
  WHERE within_ntp = 1 AND ntp_minutes_of_inaccuracy < 10;

CREATE TABLE results_messagespread_daybuckets AS
  SELECT (unix_wallclock/(24*3600))*(24*3600) AS daystart, carriers AS number_of_carriers, COUNT(*) AS count
  FROM results_messagespread_chartable
  GROUP BY daystart, number_of_carriers;

CREATE TABLE results_messagespread_hourbuckets AS
  SELECT (unix_wallclock/(3600))*(3600) AS hourstart, carriers AS number_of_carriers, COUNT(*) AS count
  FROM results_messagespread_chartable
  GROUP BY hourstart, number_of_carriers;
```

In order to calculate how many nodes received the message within 24 hours of creation, the following query is used:

```
CREATE TABLE tmp_messages_receive_with_create_date AS
  SELECT *
  FROM
    (SELECT channel, message, device AS recipient, wallclock AS receive_wallclock
     FROM tmp_messaging_receive WHERE within_ntp = 1 AND ntp_minutes_of_inaccuracy < 10)
  INNER JOIN
    (SELECT channel, message, wallclock AS create_wallclock
     FROM tmp_messaging_create WHERE within_ntp = 1 AND ntp_minutes_of_inaccuracy < 10)
  USING (channel,message);

CREATE TABLE results_messagespread24_chartable AS
  SELECT create_wallclock/1000 AS unix_wallclock, COUNT(DISTINCT recipient) AS carriers
  FROM tmp_messages_receive_with_create_date
  WHERE receive_wallclock - create_wallclock < 24*3600*1000
  GROUP BY channel, message;

CREATE TABLE results_messagespread24_daybuckets AS
  SELECT (unix_wallclock/(24*3600))*(24*3600) AS daystart, carriers AS number_of_carriers, COUNT(*) AS count
  FROM results_messagespread24_chartable
  GROUP BY daystart, number_of_carriers;

CREATE TABLE results_messagespread24_hourbuckets AS
  SELECT (unix_wallclock/(3600))*(3600) AS daystart, carriers AS number_of_carriers, COUNT(*) AS count
  FROM results_messagespread24_chartable
  GROUP BY daystart, number_of_carriers;
```

The intermediate data can be reused to find interesting messages and query for their information quickly (generating the data needed for figure 6.5):

```
CREATE TABLE tmp_selected_message AS
  SELECT datetime(receive_wallclock/1000,'unixepoch') AS human_receive_time,
    receive_wallclock, channel, message, recipient
  FROM tmp_messages_receive_with_create_date
  WHERE channel = 'a800ce7c9046ab8b' AND message = 7321;

CREATE TABLE result_selected_message_spread AS
  SELECT receive_wallclock/1000 as unixtime,
    (SELECT COUNT(DISTINCT recipient) FROM
      (SELECT receive_wallclock AS rw, recipient FROM tmp_selected_message)
     WHERE rw <= receive_wallclock) AS recipientcount
  FROM tmp_selected_message
  ORDER BY receive_wallclock;
```

Umfang der Datenerfassung

Die folgenden Daten werden von der App erfasst:
• Eigenschaften des Gerätes (Hersteller, Marke, …)

Während die Aufzeichnung aktiv ist, werden kontinuierlich mit Zeitstempeln erfasst:
• Sichtbare Bluetooth-Geräte von Teilnehmern (Pseudonym/ID und Signalstärke - interessant ist vor allem, wann sich welche teilnehmenden Geräte untereinander erreichen können)
• Bluetooth-Verbindungen zu anderen Teilnehmern sowie der Verlauf der Übertragung von Testdaten (diese enthalten keine Informationen über dich)
• Akku-Ladezustand
• Freier Speicherplatz

Weiterhin wird während der Aufzeichnung das Gerät in einen Zustand versetzt, in dem es für andere Geräte über Bluetooth sichtbar ist.

Die App fragt regelmäßig online ab, ob neue Nachrichten (z. B. Hinweise auf Updates oder Probleme) vorliegen. Dabei werden nur das Pseudonym und Versionsangaben übertragen und zusammen mit dem Zeitpunkt erfasst. Über dieses Verfahren können wir dir Nachrichten zukommen zu lassen, ohne deine Identität zu kennen. Personenbezogene Daten wie die IP-Adresse werden **nicht** gespeichert. (Das Fehlerprotokoll kann aus technischen Gründen kurzfristig IPs enthalten, wird aber täglich anonymisiert.) Ebenfalls wird regelmäßig die Zeit über Server von pool.ntp.org abgeglichen, um Ungenauigkeiten des Systemuhr auszugleichen.

Die gesammelten Daten werden pseudonymisiert veröffentlicht. Die Veröffentlichung erfolgt frühestens 14 Tage nachdem die Daten erfasst werden.

Durch die Datenerfassung steigt der Stromverbrauch. **Du wirst dein Gerät in der Regel jede Nacht aufladen müssen.**

Jedes Gerät erhält ein Pseudonym, unter welchem die Daten erfasst werden. Zusätzlich werden von uns Geräte mit bekanntem Pseudonym an geeigneten Orten (z. B. Universitätsgebäude) aufgestellt. Die erfassten Daten werden automatisch hochgeladen, sobald eine WLAN-Verbindung besteht. Vor der Veröffentlichung werden die Pseudonyme nochmal durch andere Pseudonyme ersetzt, um deine Privatsphäre so gut wie möglich zu schützen.

Andere Teilnehmer können möglicherweise dein Pseudonym in den veröffentlichten Daten wiedererkennen (z. B. anhand der Zeiten, zu denen sie in deiner Nähe waren). Daher können sie möglicherweise einige Rückschlüsse darauf ziehen, wann du welchen anderen Teilnehmern begegnet bist und wo du dich wann aufgehalten haben könntest (zum Beispiel, weil du in der Nähe eines der stationären Geräte mit bekanntem Standort und Pseudonym warst).

Du kannst die Teilnahme jederzeit beenden, indem du die App über die App-Verwaltung in den Einstellungen deines Telefons deinstallierst.

Einwilligung zur Datenerhebung

Ich habe die Hinweise zur Datenerhebung gelesen.

Die Erhebung und Veröffentlichung der Daten dient dazu, eine öffentlich verfügbare Datenbasis für Forschungszwecke zu schaffen, insbesondere für die Forschung an mobilen Peer-to-Peer- Systemen/MANETs (Systeme, bei denen Nachrichten ohne ein zentrales Netz von einem Gerät zum nächsten übermittelt werden). Die Daten dürfen für beliebige wissenschaftliche Zwecke verwendet werden.

Mir ist bekannt, dass:
- die Teilnahme freiwillig ist,
- das Gerät für andere Bluetooth-Geräte sichtbar gemacht wird,
- die erhobenen Daten pseudonymisiert veröffentlicht werden und
- die Pseudonymisierung keinen perfekten Schutz bietet.
- ich die Teilnahme jederzeit durch Deinstallation der App beenden kann.

Für die Datenerhebung zuständige Person:
Jan Schejbal (Weitere Infos und Kontakt)

Ich habe diese Datenschutzerklärung gelesen und akzeptiert. Mit der Erhebung, Auswertung und Veröffentlichung der Daten bin ich einverstanden.