



# An Evaluation Framework for distributed Android Applications

Master Thesis

by

Gian Perrone

born in

Düsseldorf

submitted to

Technology of Social Networks Lab  
Jun.-Prof. Dr.-Ing. Kalman Graffi  
Heinrich-Heine-Universität Düsseldorf

June 2017

Supervisor:

Andre Ippisch, M. Sc.



---

# Abstract

As smartphones and other portable devices like tablets gain popularity on a daily basis the possibility to provide solutions for use cases not feasible before becomes viable. Although those devices provide different means of wireless communication like Wi-Fi and Bluetooth, one such use case not solved yet and subject of current studies is to transfer data - like chat messages or files - from one device to another without using the internet. While Wi-Fi Direct and Bluetooth can provide such functionality for devices in close proximity to each other no solution exists to transfer data between devices which cannot establish a direct connection to each other.

One possible solution to this problem is the use of *opportunistic networks* on mobile devices which can forward packets to participating devices until they arrive at their intended destination. An implementation of this solution is the Android application *opptain* developed at our department. As there are a lot of variables to adjust to improve the performance and stability of the network a tailored test framework is desirable.

We present a powerful and easily expandable test framework for *opptain* to conduct measurements of various metrics under configurable configurations and aggregate these to a central point for further examination.

Several metrics were evaluated to cover the big picture of our opportunistic network regarding routing configurations, different topologies and different use cases.

We came to the conclusion that there is still a lot of room for improvement in several use cases while others work reasonably well in *opptains* current state.



---

# Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude, first and foremost my girlfriend which continually supported me to a ginormous extent. Also a lot of support, guidance, and constant motivation was provided by Andre Ippisch, my supervisor and author of the master's thesis mine builds upon. I cannot thank both of you enough.

My gratitude also covers my parents, my sister and all my friends for all the great support I received over the years.

Special thanks also go to Jun.-Prof. Dr.-Ing. Kalman Graffi and Prof. Dr. Martin Mauve for reviewing this thesis.

Shoutouts to SimpleFlips.



# Contents

<b>List of Figures</b>	<b>ix</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Related Work . . . . .	2
1.3. Outline . . . . .	2
<b>2. Fundamentals</b>	<b>3</b>
2.1. Android operating system . . . . .	3
2.1.1. Applications . . . . .	3
2.1.2. Intents . . . . .	4
2.1.3. Android SDK . . . . .	5
2.2. Opportunistic networks . . . . .	5
2.2.1. opptain . . . . .	6
2.2.2. Routing . . . . .	7
<b>3. Demands and Design</b>	<b>9</b>
3.1. Demands . . . . .	9
3.2. Design . . . . .	10
3.2.1. Planning, scheduling and aggregation . . . . .	10
3.2.2. Ad-hoc based . . . . .	10
3.2.3. Web-based . . . . .	10
3.3. Bundle generation . . . . .	11
3.4. Routing . . . . .	11
3.4.1. Epidemic . . . . .	11
3.4.2. Spray and Wait . . . . .	12
3.4.3. Binary Spray and Wait . . . . .	12
3.4.4. P <sup>R</sup> oPHET v2 . . . . .	12
<b>4. Implementation</b>	<b>15</b>
4.1. Development . . . . .	15
4.2. Åggregator on Android . . . . .	16

4.2.1. Ad-hoc based case . . . . .	16
4.2.2. Web-based case . . . . .	22
4.3. Åggregator-Webb . . . . .	24
<b>5. Evaluation</b>	<b>25</b>
5.1. Long-running opptain instances . . . . .	25
5.2. Topologies . . . . .	26
5.2.1. Event . . . . .	26
5.2.2. LazyOffices . . . . .	27
5.2.3. BusyOffices . . . . .	27
5.3. Scenarios . . . . .	28
5.3.1. Chat messages . . . . .	28
5.3.2. File sharing . . . . .	28
5.4. Routing . . . . .	28
5.5. Metrics . . . . .	29
5.6. Results . . . . .	30
5.6.1. Chat messages . . . . .	30
5.6.2. Conclusion . . . . .	32
5.6.3. File sharing . . . . .	33
5.6.4. Comparison . . . . .	35
<b>6. Conclusion and Future Work</b>	<b>43</b>
6.1. Conclusion . . . . .	43
6.2. Future Work . . . . .	43
<b>A. Åggregator User’s Guide</b>	<b>45</b>
A.1. Overview . . . . .	46
A.2. Client mode . . . . .	47
A.3. Ad-hoc server mode . . . . .	48
A.3.1. Manage DeviceSets . . . . .	48
A.3.2. Manage Series . . . . .	49
<b>B. Åggregator-Webb User’s Guide</b>	<b>53</b>
<b>Bibliography</b>	<b>55</b>



# List of Figures

2.1. opptain's Automation mode . . . . .	7
4.1. Register protocol . . . . .	17
4.2. DeviceSet database scheme . . . . .	17
4.3. Schedule protocol . . . . .	18
4.4. Series database scheme . . . . .	19
4.5. Creating a Record . . . . .	20
4.6. Aggregate protocol . . . . .	21
4.7. SQL Export . . . . .	21
4.8. Join protocol . . . . .	22
4.9. Devices protocol . . . . .	23
4.10. Aggregate protocol (web) . . . . .	23
5.1. Topology overview . . . . .	27
5.2. Chat messages / Event topology . . . . .	36
5.3. Chat messages / LazyOffices topology . . . . .	37
5.4. Chat messages / BusyOffices topology . . . . .	38
5.5. File sharing / Event topology . . . . .	39
5.6. File sharing / LazyOffices topology . . . . .	40
5.7. File sharing / BusyOffices topology . . . . .	41
A.1. Main activity . . . . .	45
A.2. Deleting an element . . . . .	46
A.3. Overview . . . . .	46
A.4. Client mode . . . . .	47
A.5. Manage DeviceSets . . . . .	48
A.6. Exported DeviceSet JSON . . . . .	49
A.7. Manage Series . . . . .	49
A.8. Configuration JSON . . . . .	50
B.1. Login screen . . . . .	53
B.2. Add screen . . . . .	54

B.3. Dashboard screen . . . . . 54

# Chapter 1.

## Introduction

### 1.1. Motivation

As smartphones and other portable devices like tablets gain popularity on a daily basis the possibility to provide solutions for use cases not feasible before becomes viable. Although those devices provide different means of wireless communication like Wi-Fi and Bluetooth, one such use case not solved yet and subject of current studies is to transfer data - like chat messages or files - from one device to another without using the internet. While Wi-Fi Direct and Bluetooth can provide such functionality for devices in close proximity to each other no solution exists to transfer data between devices which cannot establish a direct connection to each other.

There are several scenarios which provide use for this functionality:

- Data plans and mobile bandwidth is limited and can possibly be conserved.
- Internet is not always available on mobile devices, either due to bad network coverage or due to unintentional or even deliberate outages.

One approach to solve this issue is being developed at our department. First introduced in Andre Ippisch's master's thesis [Ipp15], it provides an implementation of a so-called *opportunistic network* which uses mobile devices to which a direct connection can be established to route data to its destination.

As in such an implementation there are many variables that interact with each other and several implementation choices in some areas, e.g. routing, it is desirable to have solid tooling to test the reliability and performance of our implementation under various scenarios.

## 1.2. Related Work

Several other implementations of opportunistic networks have been proposed which are referenced in [Ipp15]. However, as the intention of this thesis was to provide means to evaluate our network with very specific demands further explained in Section 3.1, a solution had to be developed from scratch.

## 1.3. Outline

In this chapter we gave the basic motivation for implementing a test framework for opptain. The rest of this thesis is structured in the following way:

In Chapter 2 we explain the basics of the Android operating system for mobile devices and the opportunistic networking functionality our implementation provides.

In Chapter 3 we define our demands for our test framework and explain how these demands led to the design of our software.

In Chapter 4 we further explain how the demands and design was implemented into both our Android application and a web application to support all our use cases.

In Chapter 5 we show the results of various tests conducted using our test framework to evaluate the performance of opptain under several circumstances.

In Chapter 6 we provide a conclusion of this thesis and present how our implementation and evaluation fulfilled our demands and the goal of this thesis. In addition possibilities for future work are introduced.

# Chapter 2.

## Fundamentals

### 2.1. Android operating system

*Android* is an operating system developed by Google for mobile touchscreen devices and is based on the Linux kernel. Full documentation from Google can be found at [ANDB] upon which the following summary is based. In this summary only functionality required to understand the remainder of this thesis is presented.

#### 2.1.1. Applications

Android Applications are run in a modified *Java Virtual Machine*, so although many languages are available for the JVM the language of choice for writing Android applications is Java. Each application runs as a different user and in its own VM, so strong isolation between applications is maintained.

Unlike common desktop operating systems Android applications are not simply executed by calling their main method. As Android tries to minimize resource usage on mobile devices it starts different components of applications as they are required and relies heavily on applications and their parts being able to restore their previous state upon restart by the operating system. A notorious example for this is that by default the user interface gets completely destroyed and reinstantiated when the user changes the screen orientation.

There are four different types of applications components which are Activities, Services, Broadcast receivers and Content providers. As content providers were not used in this thesis, their description is skipped.

## Activities

An *Activity* represents a single user interface. Although parts of the interface can be exchanged, the main idea is that an activity represents one task the user can conduct, e.g. taking a picture, reading a Facebook timeline or changing application settings. Activities can start different activities, also ones belonging to different applications, in some cases without even knowing which application it belongs to. As an example an application can instruct the operating system to share an image without knowing in advance if it will be shared through Facebook, WhatsApp or email.

## Service

A *Service* is a part of an application designed to run as a background task and does not provide an user interface. Services can for example be used to execute regular cleanup tasks or fetching data from the internet without user interaction. Android distinguishes between *foreground* and *background* tasks. While *background* tasks are intended for tasks happening without user awareness, a *foreground* task is intended for tasks the user is actively aware of like music playback. Android tries its best to keep foreground services running at all times and requires that a notification is displayed to the user to interact with the service and make the user aware of its existence.

## Broadcast receivers

A *Broadcast Receiver* can receive broadcasts sent by other applications or by the operating system. Such broadcasts are issued for example when the battery level changes, a charger is plugged in, the screen is turned off or other events happen that are of interest to applications. As they - like services - operate in the background, they do not provide a user interface, too.

### 2.1.2. Intents

Intents are the basis of inter-process communication on Android. They can be used to start all three of the application components discussed above, Activities, Services and Broadcast receivers. There are two kinds of intents, *implicit* and *explicit* intents. *Explicit* intents have a specified receiver while Android is responsible for choosing a suitable receiver in the case of *implicit* intents.

Typical examples for explicit intents are common tasks as sharing or sending a picture, where it is up to the user to decide if the picture is going to be shared via e-mail, WhatsApp or Facebook. Android

provides a list of standard actions to which applications can register as handlers, in this case the action would be *ACTION\_SEND*.

### 2.1.3. Android SDK

Released first in September 2008 [ANDa], Android has been under constant development while maintaining backwards compatibility through different API levels, so even long deprecated functionality works when compiling applications against older APIs. The API is provided in the form of platform images downloadable via a SDK manager.

When developing applications for Android it is important to make a careful assessment of the API level to program against: Older API versions do not provide as much features as newer ones but are supported by older and more widespread devices. Fortunately many new features are backported to older API levels in the form of *support libraries*. In addition the API level can be queried at runtime and newer functionality can be accessed if available on the device in question.

Unfortunately, not all functionality available on a platform is exposed via the API: There is no public method to open a Wi-Fi hotspot, although this functionality is existent in the API and required for this thesis. Such hidden functionality can be accessed using reflection, although there is no guarantee that this will work in future API levels. It is worth noting that this cannot be used to circumvent Android's rights management, which is not covered in this summary.

## 2.2. Opportunistic networks

A summary of *opportunistic networks* is given in the abstract of [HLT08]:

“We define an opportunistic network as one type of challenged networks where network contacts are intermittent or where link performance is highly variable or extreme. In such a network, there does not exist a complete path from source to destination for most of the time. In addition, the path can be highly unstable and may change or break quickly. Therefore, in order to make communication possible in an opportunistic network, the intermediate nodes may take custody of data during the blackout and forward it when the connectivity resumes.”

Such an opportunistic network is desirable for various reason covered in the Introduction, as it provides a solution to the mentioned problems of missing internet connectivity and missing bandwidth

when implemented on smartphones and other mobile devices.

### 2.2.1. opptain

One implementation of an opportunistic network, developed at our department, is called *opptain*. It was introduced and is covered in detail in Andre Ippisch's master's thesis [Ipp15]. It is based on Android devices and uses their Wi-Fi capabilities to exchange data packaged in *bundles* to other Android devices using the application. *opptain* serves as the transport layer while data can be supplied by other applications. Two such applications are *SpeeChat* and *FileShipping* which focus on chat messages and files respectively.

As there is no central authority in distributed networks a mode of addressing devices has to be defined. While the initial version of *opptain* used the MAC address of the device, the current version uses the public key of the integrated encryption and signature mechanism as a unique address.

#### Automation

*opptain*'s main mode of operation is the so called *Automation*. It consists of a state-machine which executes several tasks. The basic operation is depicted in Figure 2.1.

First, the *ScanTask* enables the devices Wi-Fi and scans for access points opened by other devices using *opptain*, abbreviated to *AP* in the illustration. These are identified using their SSID which consists of a constant prefix and the fingerprint of the devices unique *opptain* id.

If no such access point is found, the *MasterStartTask* opens one and proceeds to the *ServerTask* to listen for clients via TCP. After a random timeout has passed and all clients have disconnected it proceeds to the *MasterStopTask* which closes the access point and proceeds to the *IntermissionTask* which idles for a few seconds and then jumps back to the *ScanTask*.

If an access point is found, the *SlaveStartTask* connects to the access point, the following *ClientTask* connects to the TCP server and after the connection ends the *SlaveStopTask* closes the Wi-Fi connection and proceeds to the *IntermissionTask* like the *MasterStopTask* in the other case.



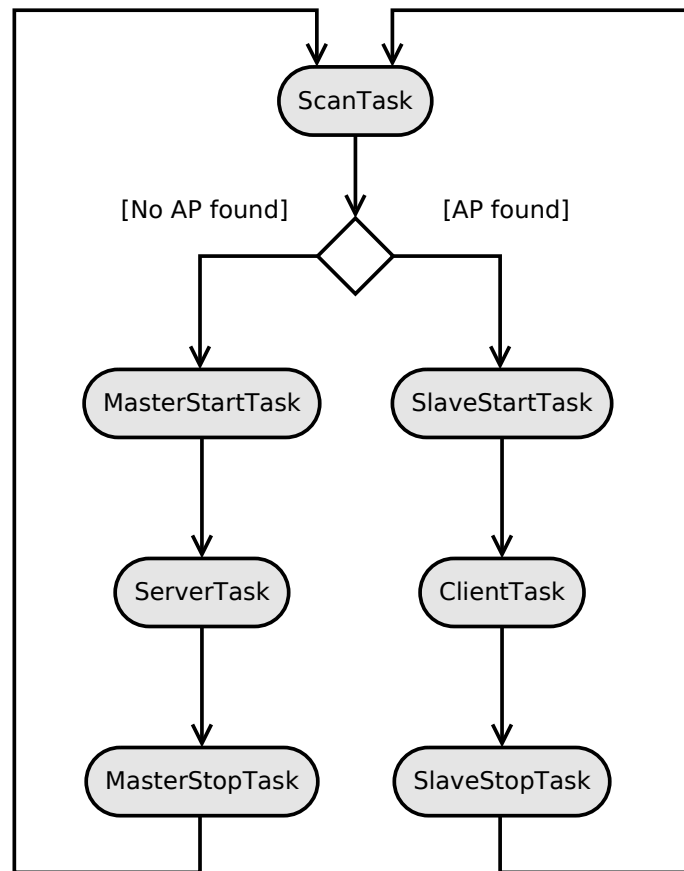


Figure 2.1.: opptain's Automation mode

### 2.2.2. Routing

Routing in an opportunistic network is non-trivial as the topology is constantly changing and there is no guarantee that a route taken earlier will be available again in the future. Several routing protocols have been proposed by different authors in the past. Some are completely unaware of the history of successful connections to other peers while others take the history and possibly other data such as location into account. These different types are described in-depth in [Ipp15]. Each routing protocol has advantages and disadvantages specific to the use-case and there is no clear best or worst protocol which fits all of them.



## **Chapter 3.**

# **Demands and Design**

### **3.1. Demands**

The goal of this master thesis was to create a framework for use with opptain to conduct fully automated and highly configurable measurement series to evaluate the Automation mode of the opptain network under various aspects. The main focus was to enable the evaluation of different routing protocols under different use-cases. To achieve this goal an application had to be developed which has to serve several demands.

It must be possible to plan and schedule measurement series on multiple participating devices so that opptain is operated in a fully automated way without required user interaction. This is important so that measurements can be conducted easily both in test environments and by real users, so that the least possible effort is required by these users.

Bundles have to be generated to send and receive via opptain to simulate the load of interest on the network. This should be highly configurable to simulate different and mixed loads as these stress the network in possibly different ways.

During a measurement series measured values must be collected from opptain and stored for later analysis. Recording measured values should be simple and opptain should still be able to run properly without a measurement series taking place. The recorded values must later be aggregated from their respective devices to a central location and exportable for analysis.

All these demands should be met both in confined test environments and on real-life devices e.g. belonging to students.

## 3.2. Design

### 3.2.1. Planning, scheduling and aggregation

One of the basic requirements was that measurement series should both be possible on test devices and on real-life devices e.g. belonging to students. These requirements are fundamentally different in many ways and will be explained in detail in the following sections.

Due to the way we implemented solutions to both these scenarios we refer to the first case as *Ad-hoc based* and to the latter case as *Web-based*.

### 3.2.2. Ad-hoc based

In this case, as participating devices are known beforehand, measurement series should be able to be planned device-dependent. To evaluate different topologies without physically moving devices it should be possible to *whitelist* devices, so that opptain can only connect to devices on its whitelist. This vastly simplifies taking measurements which do not depend on the distance of the devices.

Not all Android devices allow setting the clock via GPS, so devices do not necessarily have synchronized clocks and Android does not allow setting the clock from applications not supplied by the device vendor, so a mechanism should be implemented to synchronize the aggregated data. The data should be aggregated on a central Android device via ad-hoc Wi-Fi networking as no internet access is provided on the test devices, so the devices have to be brought back the same physical location after a test.

### 3.2.3. Web-based

In this case the participating devices are not known beforehand, so measurement series cannot be planned device dependent. Nonetheless participating devices have to know about each other as they have to generate bundles to deliver to each other via opptain.

Real-life devices usually have synchronized clocks as they obtain their time from the cellular network. As there is no guarantee that participating devices ever meet again in one physical location internet access is mandatory so that data can be aggregated to a web server.

Note that our requirements for real-life devices cannot be satisfied in all situations where testing

opptain would be desirable. For now there is no way to test opptain in real disaster-hit areas where neither internet access nor a central location is available.

### 3.3. Bundle generation

The bundle generation has to be highly configurable in order to provide a proper environment for testing. Both the interval of bundle generation and the size of the bundle payload should be variable and different distributions should be possible. Multiple generators should be able to run at the same time to mix different use cases. The distributions chosen for implementation were:

- *set*: Pick values from a set of values with equal probability.
- *uniform*: Pick values uniformly distributed between a lower and an upper bound.
- *loguniform*: Pick values whose logarithm is uniformly distributed between a lower and an upper bound.

The last distribution was chosen for implementation as it can be used to provide a more realistic distribution of file sizes than a simple uniform distribution.

### 3.4. Routing

The main focus of evaluation was put on the comparison of different routing schemes available for opportunistic networks. Several schemes have been proposed in the past and a suitable selection had to be made. The available options range from the simple flooding of each message to each connected peer to elaborate schemes which e.g. take into account the history of peer connections. For our tests, we chose to implement four different routing protocols spread across this spectrum.

#### 3.4.1. Epidemic

Defined in [VB00], this is a basic routing scheme which does not require connection history or other data. Each and every single bundle is forwarded to each and every single connected peer. While on one hand this makes sure every possible route is taken on the other hand it potentially consumes a lot of resources. In contrast to blindly flooding the network we only forward a bundle to connected peers if they do not already have a copy of the bundle.

### 3.4.2. Spray and Wait

Spray and Wait, defined in [SPR05], is a routing protocol which limits the amount of copies of a single bundle in the network to a parameter  $L$ . It achieves this using two phases: In the first phase, called Spray, the sender forwards one copy of the bundle to each of the first  $L - 1$  connected peers, so that  $L$  copies exist in the network in total. In the following Wait phase, each of the peers just waits to encounter the bundle destination without further forwarding the bundle to other peers. Obviously this limits the hops a bundle takes to two, the first hop being from the sender to an arbitrary peer, the second being from this peer to the destination.

### 3.4.3. Binary Spray and Wait

Binary Spray and Wait, also defined in [SPR05] is a variant of the Spray and Wait protocol, where not only one, but half of the remaining copies are transmitted to each encountered peer if the peer did not receive any copies yet. As an example, if  $L = 8$ , on the first encounter the sender keeps 4 copies and forwards 4 copies to the encountered peer, so both peers have 3 copies left to forward. This variant lifts the limit of two hops imposed by Spray and Wait while keeping the number of copies of a bundle in the network the same.

### 3.4.4. PRoPHET v2

The *Probabilistic Routing Protocol using History of Encounters and Transitivity v2*, hereafter shortened to *PRoPHETv2*, defined in [GDLD11] and [LDD12] is an update to the earlier *Probabilistic Routing Protocol using History of Encounters and Transitivity* defined in [LDS03]. It is based on the assumption that encounters between peers are not truly random but follow patterns so that the probability to meet the same peer multiple times is higher than the probability to meet a new peer. Several equations are defined to calculate an approximate probability  $P$  of bundle delivery based on the connection history between pairs of peers.

Our implementation of PRoPHETv2 is based on the definition in [LDD12] and takes the parameters  $P_{\text{encounter\_first}}$ ,  $P_{\text{encounter\_max}}$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ , for which the default values were taken. Additionally a *time unit* is needed which defines the resolution of the protocol. No default value for this time unit is proposed by the authors but is dependent on the properties of the network in question.

$$P_{A,B} = P_{A,B_{\text{old}}} + (1 - \delta - P_{A,B_{\text{old}}}) \quad (3.1)$$

$$P_{A,X} = P_{A,X_{\text{old}}} \cdot \gamma^K \quad (3.2)$$

$$P_{A,C} = \max(P_{A,C_{\text{old}}}, P_{A,B} \cdot P_{B,C_{\text{recv}}} \cdot \beta) \quad (3.3)$$

The first calculation two peers  $A$  and  $B$  execute upon opening a connection is to update their respective *delivery probabilities*  $P_{A,B}$  and vice versa. If they have never met before,  $P_{A,B}$  is initialized to  $P_{\text{encounter\_first}}$ . Otherwise it is updated according to equation 3.1.  $\delta = 0.01$  is a parameter intended to set an upper bound to  $P_{A,B}$ .

After this update every delivery probability  $P_{A,X}$  already in the database is aged according to equation 3.2, where  $K$  is the number of time units passed and  $\gamma = 0.999$  is the *aging constant*. This aging constant in conjunction with the choice of the time unit dictates how fast the delivery probabilities will decrease. For the default value of  $\gamma$  it takes  $0.999^K = 0.5 \Rightarrow K \approx 693$  time units to halve a delivery probability. We chose 30 seconds as time unit so that in our case the amount of time it takes to halve the delivery probability is approximately 6 hours.

The last update step that takes place is to account for transitivity. First they exchange their lists of delivery probabilities to other nodes with each other. From each delivery probability  $P_{B,C_{\text{recv}}}$  reported to peer  $A$  it updates or initializes its delivery probability  $P_{A,C}$  according to equation 3.3. This means that the new delivery probability is the maximum of the old delivery probability or the product of both the own delivery probability to peer  $B$ , the received delivery probability between  $B$  and  $C$  and  $\beta = 0.9$ .  $\beta$  defines the influence each hop has on this calculation.

After all these calculations have taken place a routing decision is to be made. The authors do not define a single best scheme to decide if a packet should be forwarded or not. We chose to forward packages if a configurable minimal delivery probability  $P_{\text{min}}$  is achieved.





## Chapter 4.

### Implementation

Most of the aforementioned demands were implemented in a single Android application called *Åggregator*. The name was chosen because the application *aggregates* data from *opptain* and the letter *å* is pronounced like *o* in most of the Scandinavian languages, so it is pronounced like *oggregator*. Solely to support the web-based case, a web application called *Åggregator-Webb* was developed, its name again derived from a Scandinavian language, in this case Swedish, where *webb* means (inter)net.

Participating devices are distinguished by their unique opptain device ID, which is a public key to opptain's underlying signature and encryption mechanism. In the following this unique identifier is referred to as *oppid* to prevent confusion with database IDs.

As both the SQL database used in the Android application and the one used in the web application use limited sets of data types not the actual data types used in the implementation but descriptive datatypes such as *ENUM*, *BOOLEAN* and *DATETIME* are used in database structure diagrams, although the implementation may differ. As an example these three data types are all implemented as *INTEGERS* on Android. The database structure and the used protocols were chosen to be described in detail as they offer a sensible description of how *Åggregator* and *Åggregator-Webb* work.

#### 4.1. Development

As mentioned before, Android applications run in a special JVM, so Java is used to develop Android applications. Google provides all the necessary APIs and tools to build Android applications together with an Integrated Develop Environment called *Android Studio* for free. *Android Studio* is based on a widespread IDE for Java called *IntelliJ IDEA* by the Czech company JetBrains. To be able to parse configuration files in the human-readable *JSON* format the *GSON* library by Google was used. Android provides an interface to use embedded *SQLite 3* databases.

To keep things consistent the web application was implemented in Java, too, using the aforementioned *IntelliJ IDEA* as an IDE. The web application is based on both the *Spring Framework* and the *Vaadin Framework*. *Spring* is an application framework for Java mainly focused on web applications. Our web application uses *Spring* to integrate a web server, an embedded SQL database (*H2*) and HTTP endpoints to communicate with the Android application. Furthermore it delegates access from a browser to *Vaadin*, which is a framework for one-page web applications with the look and feel of desktop applications.

For communication both between two Android devices and from an Android device to the web server *Protocol Buffers* by Google are used. They provide an easy mechanism to serialize data for transport via TCP and even HTTP.

## 4.2. Ågggregator on Android

A user manual to Ågggregator is provided in Appendix A and should be read before continuing to gain a basic understanding how the application is operated. To understand how Ågggregator works, first the ad-hoc based case is explained in its entirety and then only the differences in the web-based cases are explained.

### 4.2.1. Ad-hoc based case

In the ad-hoc based case the measured data is aggregated on a central Android device which we call the *server* in contrast to the other devices called *clients*. The server is responsible for Planning, Scheduling and Aggregation. To accomplish this it communicates with the clients using the built-in Android functionality of opening a Wi-Fi hotspot to which the clients can connect, much like *opptain* uses this functionality to distribute bundles.

For normal operation every action is initialized and configured by the server while the client just has to connect to fetch or push its data. Every such connection starts with the client sending a *ClientHello* message which contains its oppid and the current time. After that the server can send requests.

As in this case the devices are known beforehand, they can be stored in the server's database using an action called *Register* as depicted in the user's guide. The *Register* protocol is shown in Figure 4.1. No additional data is needed, so only an empty request and an empty confirmation have to be sent. After all devices have been registered, there are two choices. Either a so called *simple* measurement series can be planned directly or the device data can be exported to JSON to configure a *complex*

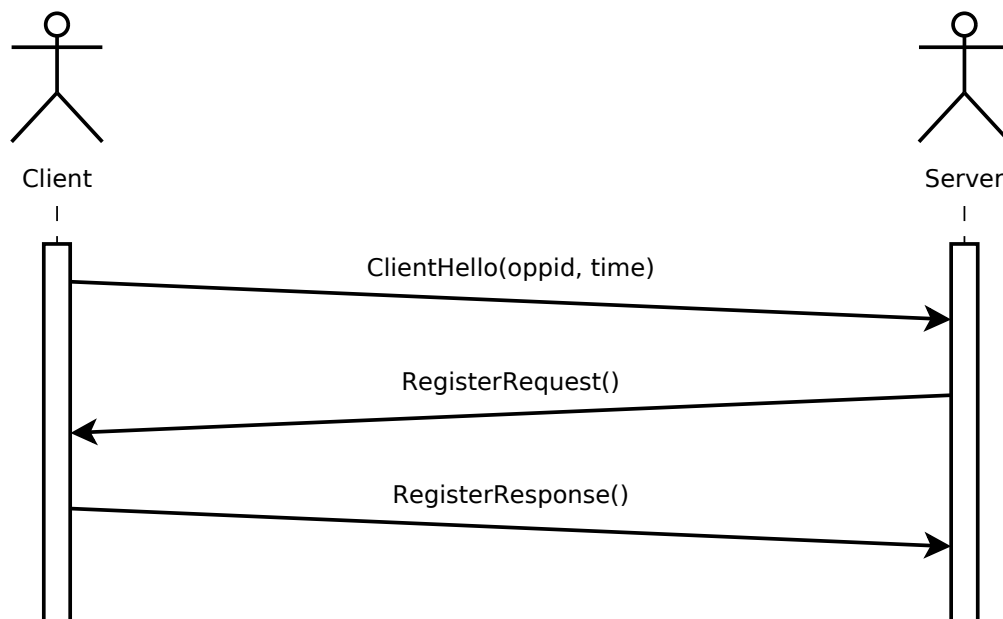


Figure 4.1.: Register protocol

measurement series using a PC.

As stated in the user's guide a feature called *Whitelisting* was implemented for opptain using *DeviceSets* and *DeviceGroups*. Only devices within the same *DeviceGroup* can see each other's Wi-Fi hotspots when looking for a connection.

Of course *Devices* can belong to multiple *DeviceGroups*, so that every possible static topology is representable by this model. To further simulate temporally changing topologies, i.e. moving devices, a feature was implemented to change the active *DeviceSet* in a configurable interval. The underlying database scheme is visualized in Figure 4.2. The IDs carry no meaning and are generated by the database.

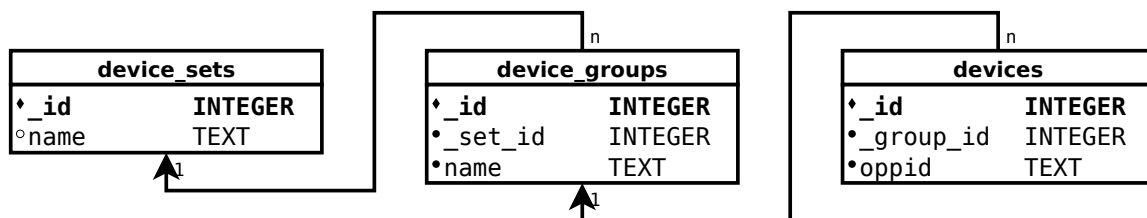


Figure 4.2.: DeviceSet database scheme

Internally both the *simple* and the *complex* measurement series case lead to a JSON configuration,

which in the next stage is distributed to clients using an action called *Schedule*, its protocol shown in Figure 4.3.

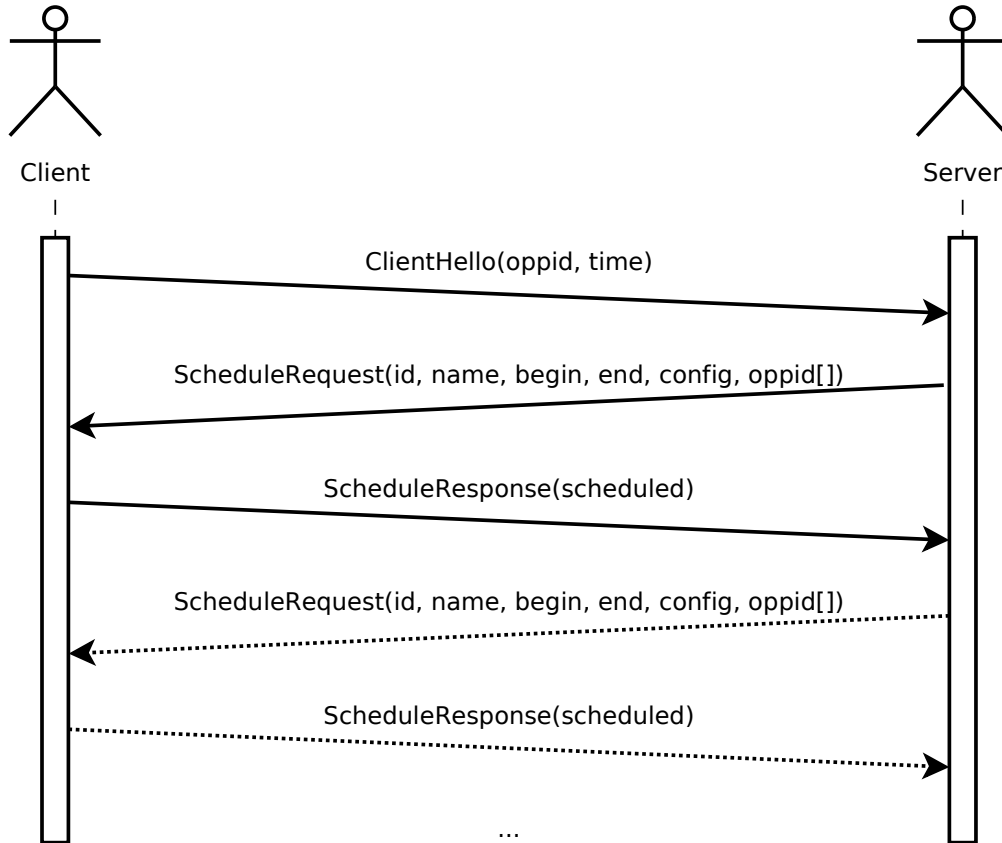


Figure 4.3.: Schedule protocol

All available information regarding the measurement series is sent from the server to the client, which includes its unique randomly generated ID, its display name, its begin and end times, its JSON configuration and a list of all participating devices. A convenience feature implemented after taking the first test measurements is that multiple series can be scheduled during the same session.

The series data is stored in the *Series* table according to the database scheme depicted in Figure 4.4. The *LocalRecord* and *AggregatedRecord* hold the collected records and will be explained later. The *close* field is not used in the ad-hoc based case and the valid values for *role* are *ADHOC\_SERVER* and *ADHOC\_CLIENT*. All participating devices have a corresponding *SeriesDevice* entry, where the time difference to the host device and the one of the states *CREATED*, *SCHEDULED*, *AGGREGATED* is saved which are self-explanatory, initialized to *CREATED* and advanced to *SCHEDULED* after scheduling.

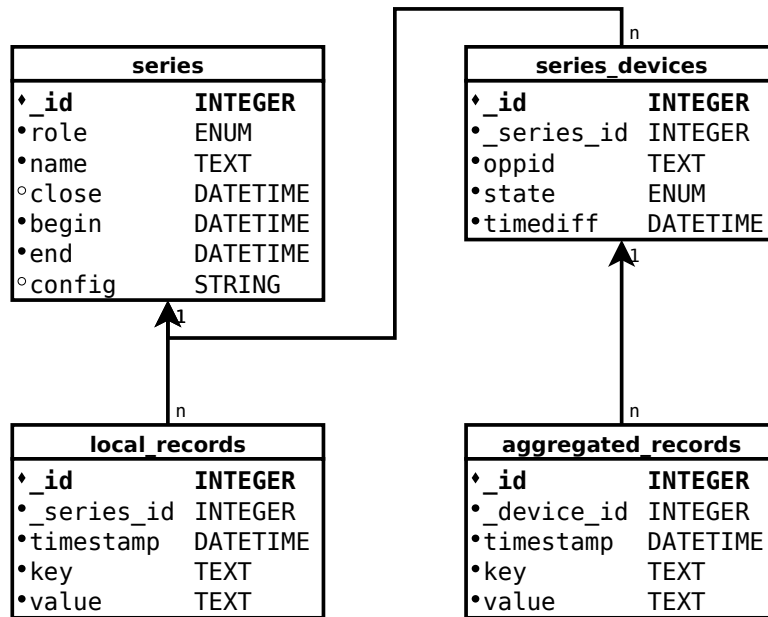


Figure 4.4.: Series database scheme

This leads to both the server (if it is participating in the series) and the clients scheduling tasks to begin and end measurement series using Androids built-in Alarm Manager which delivers Intents to Åggregator's *BeginService* and *EndService* at specified times.

The purpose of the *BeginService* is to start opptain, delete all its previous data to start with a fresh instance of the network and to configure opptain as defined in the JSON configuration. Also, the first two records are created: A meta record designating the start of the series and a record indicating the current battery level.

After all initialization is done the *CreateBundleService* is started which is a foreground Service displaying a persistent notification to the user and generates bundles for delivery through the opportunistic network as configured. opptain is constantly being sent empty Intents to keep it running in the event of a crash.

While a test is running the *RecordReceiver*, a Broadcast receiver, listens for records generated by opptain and saves them to the database. Designing this as a Broadcast receiver leads to a loose coupling from opptain to Åggregator as broadcasted Intents are simply ignored by Android when no suitable Broadcast receiver is present so opptain can be run without code changes if Åggregator is not installed on a device.

Only one class in opptain interfaces directly with `Aggregator`. Its purpose is to create and persist records, an example of its use is given in Figure 4.5.

```
new OggregatorRecord()
    .add("bundle_id", opptainBundle
        .getBundleId().toString())
    .add("bundle_state", "received")
    .add("bundle_hops", opptainBundle
        .getAdditionalInformation().get("hops", 1)
        .toString())
    .record();
```

Figure 4.5.: Creating a Record

When the *RecordReceiver* receives the generated intent it generates as many new *LocalRecord* entries (see Figure 4.4) referencing the *Series* as there are key-value-pairs while the current timestamp serves as a unique identifier for a whole record consisting of multiple such pairs.

After a measurement series has completed the *EndService* stops opptain, again creates two records to indicate the end state and the battery level of the device and deletes all generated files. The next manual step is to aggregate all records on the ad-hoc server device using the *Aggregate* protocol shown in Figure 4.6.

Again, multiple measurement series can be aggregated at once as a convenience feature. After the server asks for data regarding a specific series the client sends all its records belonging to the series in question to the server. Only after the server has successfully written these to its database and its state field has been advanced to *AGGREGATED* the client deletes all its data regarding the series.

The server saves the records of the client as *AggregatedRecords* (again, see Figure 4.4) just as the client saved them as *LocalRecords* but referencing the corresponding *SeriesDevice* instead of the *Series*.

As a final step the aggregated values can be exported to SQL suitable for SQLite 3 for further analysis. In this case for every key present in key-value-pairs a new column is created in the exported table so that key-value-pairs recorded at the same time by opptain end up in the same table row with unused fields set to *NULL*. This is illustrated in Figure 4.7 where `_device_id 3` references a *SeriesDevice* with the abbreviated oppid `654b`.

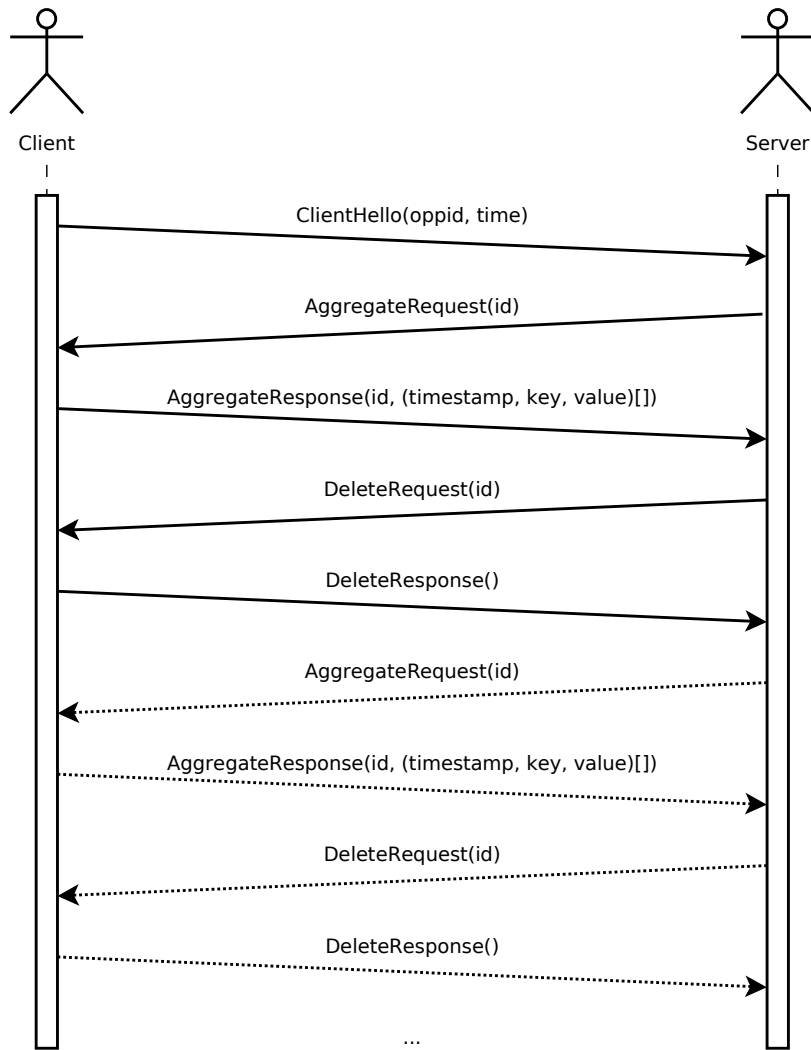


Figure 4.6.: Aggregate protocol

id	_device_id	timestamp	key	value
41	3	1498581000	_state	started
42	3	1498581090	bundle_id	34be...
43	3	1498581090	bundle_state	received
44	3	1498581090	bundle_hops	2
45	3	1498581091	bundle_id	34be..
46	3	1498581091	bundle_state	delivered
47	3	1498581091	bundle_hops	2

→

_oppid	_timestamp	_state	bundle_id	bundle_state	bundle_hops
654b...	1498581000	started	NULL	NULL	NULL
654b...	1498581090	NULL	34be...	received	2
654b...	1498581091	NULL	34be...	delivered	2

Figure 4.7.: SQL Export

### 4.2.2. Web-based case

In the web-based case the roles while communicating with the web server are reversed: The client specifies the action to take. This case skips the *Register* action but instead registers and schedules in the same step called *Join* depicted in Figure 4.8. As the client and the server communicate over a stateless protocol (ProtoBuf over HTTP) all relevant information is included in each request and response.

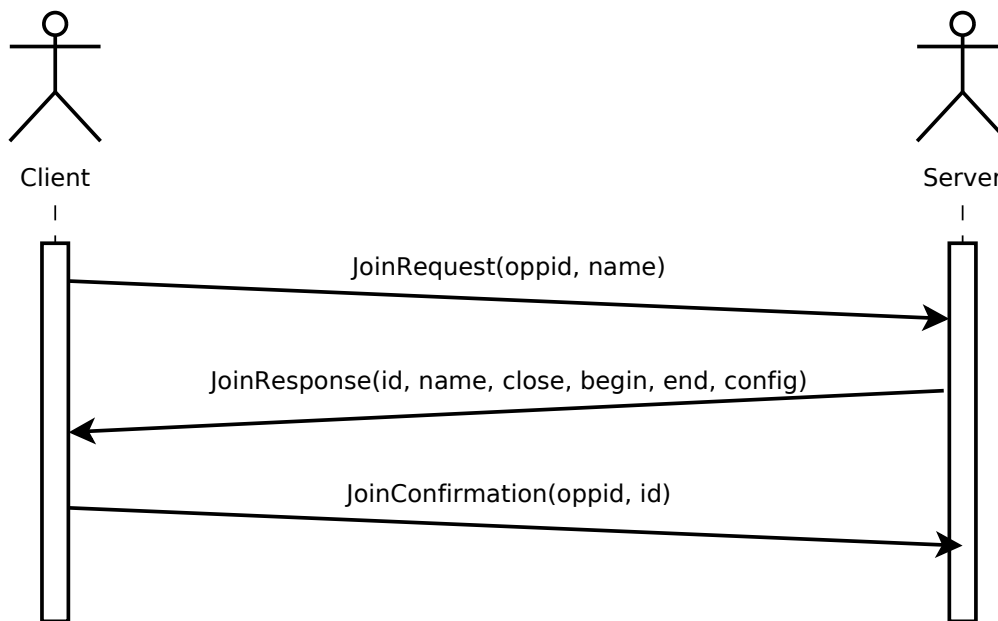


Figure 4.8.: Join protocol

The *JoinRequest* contains the name of the series to join which acts as an identifier and authorization to join the series. No information about participating devices is exchanged at *Join* time but instead a *close* time is fixed, which is also inserted into the database (see Figure 4.4). No further manual steps have to be taken to conduct a web-based measurement series on the client side.

The *role* is set to *WEB\_CLIENT*. The *close* time, which has to be before the begin time, denotes the point in time until which devices can join the measurement series. When close time passes the client tries to send a *Devices* request to the web server in intervals of one minute which can be seen in Figure 4.9.

As in the ad-hoc based case a *SeriesDevice* is created for every participating device in the series but the only whitelisting applied whitelists all participating devices so no further restrictions on visibility can be placed. If it is not possible for a client device to retrieve the list of participating devices in time



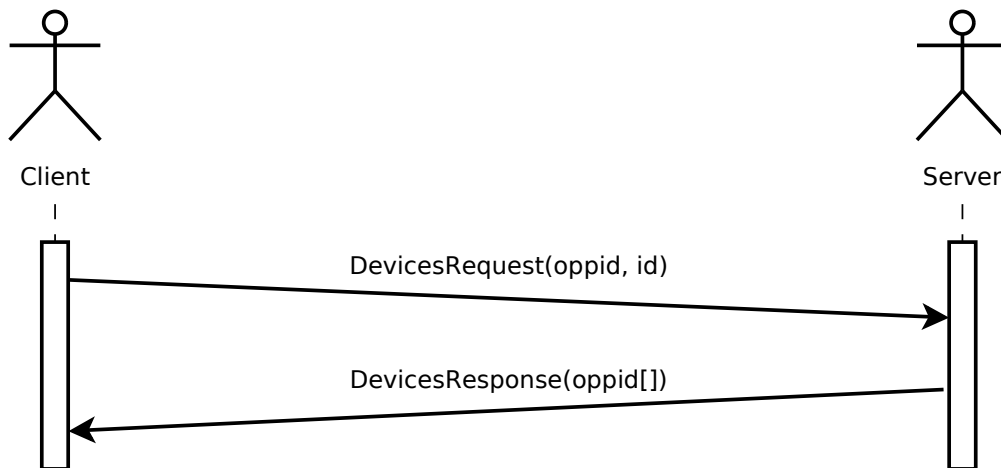


Figure 4.9.: Devices protocol

it does not participate in the measurement series and only creates a record qualifying this case.

The following steps until the aggregation of the data are equal to the ad-hoc based case. Aggregation in turn is conducted automatically, again tried in intervals of sixty seconds. For this step the protocol displayed in Figure 4.10 is used.

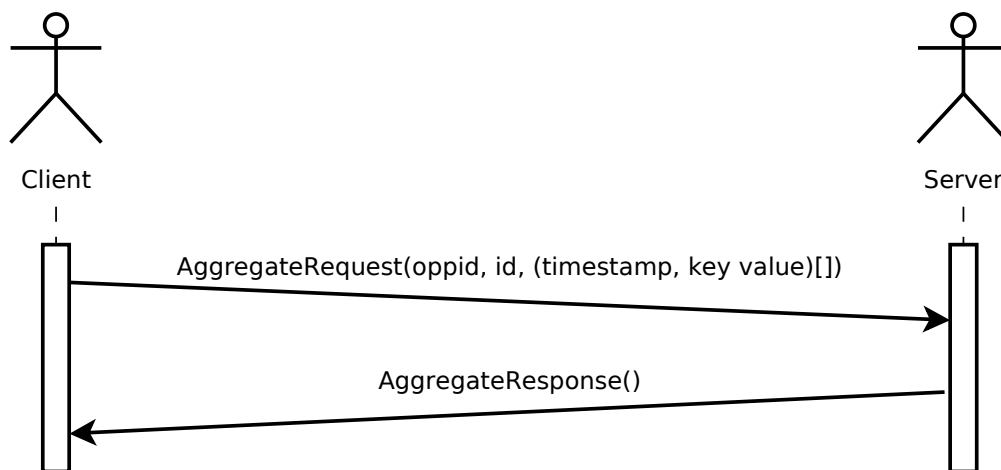


Figure 4.10.: Aggregate protocol (web)

### 4.3. Åggregator-Webb

Åggregator-Webb is the web-based server application to support the aforementioned web-based use case. It is relatively simple as it does not need to support any client features. A user manual to Åggregator-Webb is provided in Appendix B and should again be read before this section.

The whole application is self-contained in one JAR which can be executed on any server with a suitable Java Runtime Environment. The address of the server must be hardcoded into Åggregator by design.

The database scheme of Åggregator-Webb equals the one used by Åggregator (see Figure 4.4) but omits all unneeded tables: Only *Series*, *SeriesDevices* and *AggregatedRecords* need to be persisted. Instead of SQLite 3 it uses an embedded H2 database which was designed for usage with Java and is natively supported by *Spring*.

The user interface is implemented with *Vaadin* and provides a single-page web application. Communication with the Android client application is done using *ProtoBuf* over HTTP using *Spring* functionality intended for RESTful web services.

In contrast to the SQL export of the Android application the list of participating devices is also exported as it is not known beforehand. Aggregated records can be exported at any time as there is no guarantee that every real-life device can or will provide its measured data.

## Chapter 5.

### Evaluation

The evaluation was split into two parts: First opptain was subjected to critical scrutiny when running over long periods of time to find possible problems with either the application or the underlying operating system Android. After this was finished actual measurement series could be recorded.

#### 5.1. Long-running opptain instances

The first thing to test was if and how opptain was fit to run continuously for long times without user intervention. As opptain is a large project with a lot of concurrency many problems appeared while testing. The larger and hard-to-debug problems are described in this section.

opptain is largely built on the assumption that Android will not kill it while a foreground service is running. Alas, while Android makes sure the service is running at all times, the assumption did not hold, as Android restarted the opptain application after as little as 15 minutes. This was mitigated by making opptain and our measurements restart safe.

For reasons yet unknown to us the reference to the Automation gets dropped sometimes. While this rarely happens it means that we lose the ability to control the automation after a measurement series ends. This was circumvented by completely killing opptain after the end of a measurement series, as we do not need it anymore and can start the next measurement series with a fresh instance of opptain.

Several resource leaks were found and fixed, mainly database related. Also several instances were found where concurrent collection access from multiple threads could crash opptain. All instances found were fixed.

The Automation was programmed to halt if any problems appeared. This is suitable for manually

testing the application to analyze what happened but not for automated tests on an operating system like Android which does not provide many guarantees. E.g. scanning for Wi-Fi networks failed occasionally for no apparent reason and worked perfectly on the second try. This was fixed to continue running.

After that the Automation proved to leave several threads running after their respective tasks were finished. This could have been the reason for the first problem, which was already mitigated when this problem was discovered. This was fixed.

Also, the Automation would sometimes get stuck on the *ServerTask*. This task is programmed to run as long as clients are connected. It turned out that it takes a very long time on Android for connections to time out if they were not properly closed. This was mitigated by regularly killing connections after no packets have been received for some time.

As we could not rule out the possibility of other rarely occurring timing bugs or yet undiscovered concurrent collection access crashing *opptain* we make sure *opptain* is running at all times, and if not, restart it. Some crashes happened so rarely that it was not possible to discover their cause yet.

A problem that was not addressed as of yet is that Android can delay Intents arbitrarily, in rare cases up to several seconds. This is a limitation of the operating system we work on and not certainly circumventable. The problems resulting from this will be covered later.

## 5.2. Topologies

For our main tests we used three different topologies, two static ones and one with moving devices, all simulated by whitelisting. We call these *Event*, *LazyOffices*, and *BusyOffices*. Each topology consists of 21 devices to provide comparability.

### 5.2.1. Event

In the *Event* topology each of the 21 devices can see every other device; no whitelisting is applied. This could for example be the case in a lecture hall, at a public event, or in an open space office. In this case there are a lot of clients and servers that can connect to each other.

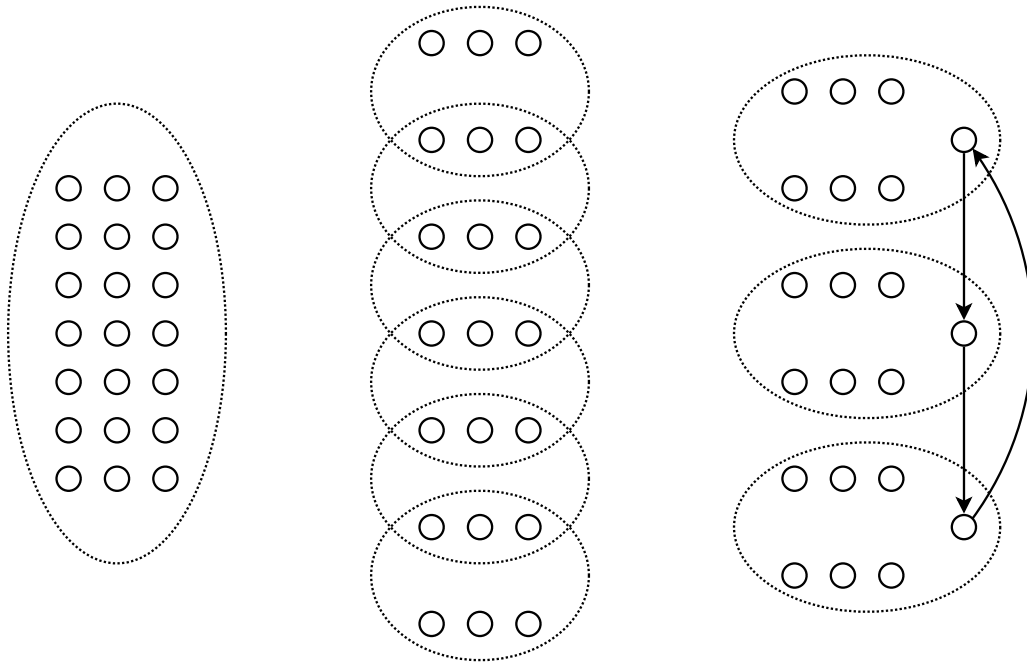


Figure 5.1.: From left to right: The *Event* topology, the *LazyOffices* topology and the *BusyOffices* topology. Small circles denote devices, the dotted ellipses denote *DeviceSets*, i.e. devices that can see each other.

### 5.2.2. LazyOffices

The *LazyOffices* topology simulates an office building with offices of three people each where the Wi-Fi only reaches the neighbouring offices. This means a lot of hops are needed to reach from one end to the other in some cases, the maximum number of hops needed being 6. The average number of nodes reachable in two hops, the maximum for the Spray and Wait routing protocol, is  $3 \cdot (8 + 11 + 14 + 14 + 14 + 11 + 8) / 21 \approx 11.4$ .

### 5.2.3. BusyOffices

This topology also simulates an office building but this time with three sets of offices of six people each. These sets are not able to see each other but three rather busy employees walk around the building. In each of its three temporally changing configurations each one of the busy employees is whitelisted in one of the sets. While none of the busy employees can ever see each other, each bundle can reach any destination in at most two hops via one of the busy employees. The main intent of this

topology was to test how a context-aware routing protocol such as P<sub>RO</sub>PHETv2 will behave.

### 5.3. Scenarios

Each of the three topologies was tested in two different scenarios: The *Chat messages* and the *File sharing* scenario. Both of these were derived from existing Android applications for opptain. Emphasis was put on the first case as the latter one has certain limitations in opptain's current version.

#### 5.3.1. Chat messages

The first scenario takes the idea from the *SpeeChat* application which is intended to deliver chat messages to peers. Chat messages are typically very small in comparison to files and happen rather often. In this configuration each node created a bundle with a fixed payload size of 1 KiB every 30 to 300 seconds using the *loguniform* distribution. The TTL was set to a constant 30 minutes and each test ran for 90 minutes.

#### 5.3.2. File sharing

The second scenario is inspired by the *FileSharing* application for opptain intended to deliver files to peers. Unfortunately the current version of opptain has no limit on how many storage space it uses for bundles and storage space on mobile devices is still rather limited. There is ongoing research to limit the storage space used to an acceptable amount and delete bundles to make room for other ones, but for our evaluation we had to make sure there was a strict size limit. This resulted in a rather small and simple case of generating a 1 MiB file each 5 minutes to stay under 4 GiB, a safe limit for our test devices. The file sharing scenario will become much more interesting in the future but was not emphasized here. The TTL was set to 100 minutes which is longer than the test duration as files do not get uninteresting as fast as chat messages.

### 5.4. Routing

All of the routing schemes described in Section 3.4 were put to test in different configurations with the exception of the *Epidemic* routing scheme as there are no configuration parameters. Both the *Spray and Wait* and the *Binary Spray and Wait* protocols were tested with their maximum number of

copies in the network  $L$  set to  $L \in \{4, 8, 16\}$ , where 16 is the highest power of 2 less than the number of participating devices. The threshold delivery probabilities for the *PRoPHETv2* protocol were set to  $P_{\min} \in \{0.4, 0.5, 0.6\}$ . These seemed to be reasonable thresholds as we had no prior empirical knowledge about this protocol. The total amount of routing configurations tested adds up ten.

## 5.5. Metrics

This evaluation focuses mainly on the big picture of an opptain network. Nonetheless some measurements were taken to make it possible to examine opptain on a smaller scale. Due to the nature of whitelisting these were limited to metrics which are not significantly influenced by physical distance between devices.

The events recorded to examine the big picture are:

- Bundle generation at the source including the destination
- Bundle receipt at a relay or the destination including the amount of hops taken so far
- Bundle delivery to the destination including the amount of hops taken

From these records the following metrics can be read or calculated using SQL only:

- Delivery rate, i.e. the amount of bundles successfully delivered before its TTL runs out
- Distribution of hops taken
- Distribution of latency, i.e. the time a bundle takes from source to destination

To enable measurements on a smaller scale the time spent in each of the *Automation* tasks is logged, too. Additionally the battery state at the start and at the end of a measurement series is logged but neglected in this evaluation as all test devices were connected to a charger at all times during the tests.

The aforementioned problem of Android arbitrarily delaying Intents surfaced when calculating latencies. In rare cases a small negative value of up to a few was computed although all other measured data was sound and synchronized. As real latencies were generally much larger than the impact of the delay introduced by Android negative values are not shown in the diagrams and introduce no significant error to the measurements.

## 5.6. Results

As stated before, for each of the combination of the two scenarios and the three topologies we compared ten different routing configurations. For each of these 60 cases we analyzed the delivery rate, the hops taken, the latency for the big picture and further looked at how long the Automation remained in server and client mode. The time to scan for networks and the time to open and close connections was not considered, as these times are device dependent and not configurable.

For both scenarios, first the results of the measurement series for each topology will be presented independently followed by a comparison of the topology results. Finally a comparison between both scenarios will be shown.

The diagrams shown for each case are from left to right, top to bottom:

1. Delivery rate
2. Distribution of hops taken
3. Distribution of latency
4. Total time spent in *ServerTask* vs. Total time spent in *ClientTask*
5. Distribution of *ClientTask* duration
6. Distribution of *ServerTask* duration

As there are few but severe outliers for the duration diagrams both a zoom to the main area of interest and the whole view is provided. To be able to compare the diagrams between the different topologies related diagrams are scaled equally.

### 5.6.1. Chat messages

#### Event topology

The results are shown in Figure 5.2.

Depending on the chosen routing protocol high delivery rates can be achieved. Comparing delivery rates and latency distribution Spray and Wait is in clear lead for  $L = 4$  and  $L = 16$  but not for  $L = 8$  where it performs equally bad as Epidemic. Binary Spray and Wait and PRoPHETv2 perform similarly although PRoPHETv2 presents more outliers with higher latency. Looking at the hops diagram



one can see that although the mean amount of hops taken by Binary Spray and Wait is only slightly less than that of PRoPHETv2, there are a few outliers with higher hop count for PRoPHETv2.

In all cases most client connections took less than 40 seconds with the best performing Spray and Wait configurations having mean client times lower than 10 seconds. For these configurations the time spent in server mode is comparatively long, so few hotspots serve as hubs for many bundles.

A possible conclusion is that in the case of the open *Event* topology it is desirable to stay in server mode for large amounts of time to benefit the whole network. Also a small amount of hops is desirable as can be seen by the good performance of Spray and Wait which by design limits the maximum number of hops for each bundle to two.

### **LazyOffices topology**

The results are shown in Figure 5.3.

Taking into account that only  $20/11.4 = 57\%$  - the number of possible target devices divided by the number of devices reachable in at most two hops - of bundles can reach its destination using Spray and Wait it performs remarkably good. Although again Spray and Wait provides low latencies the overall delivery rate of PRoPHETv2 is the highest. It achieves this by being able to use more hops, up to 11 in the most extreme cases, although every bundle can possibly reach its destination in 6 hops. This comes with the drawback of higher latencies.

Both Epidemic and Binary Spray and Wait perform poorly in comparison to PRoPHETv2 and Spray and Wait, although Binary Spray and Wait provides low latencies for the bundles it delivers.

In this topology the distribution of client and server time is much closer than in the *Event* topology. The shortest client times are provided by Binary Spray and Wait with  $L = 8$  which is also the routing protocol with the least amount of successfully delivered bundles.

In conclusion a tradeoff has to be made between low latencies and high delivery rate when choosing the routing protocol. As the distribution of client and server time does not vary much no obvious conclusions are possible here.

### **BusyOffices topology**

The results are shown in Figure 5.4.

Although PRoPHETv2 is specifically designed with changing topologies in mind both Spray and Wait and Binary Spray and Wait perform better regarding the delivery rate, although this is strongly dependent on the used configuration. As in the *Event* topology, Spray and Wait performs poorly for  $L = 8$  while PRoPHETv2 performs better the lower  $P_{\min}$  is set.

The latencies of each protocol vary far less than in the other two topologies. Fortunately, the routing protocols with the highest delivery rate also provide the least mean latency.

The comparison of client and server time favours the server mode slightly more than in the *LazyOffices* case although this is not directly evident when looking at the distribution of client time and the distribution server time.

Again, no obvious conclusions can be drawn from the distribution of client and server times. The Spray and Wait protocol performs best for certain configuration, Binary Spray and Wait being a close competitor.

### 5.6.2. Conclusion

Spray and Wait, although one of the most simple routing protocols, performs surprisingly good in comparison to our other tested routing protocols. For this to be achieved either a small or a large number of maximum bundle copies need to be available with the intermediate case of  $L = 8$  having a significantly worse performance than  $L = 4$  and  $L = 16$ .

Binary Spray and Wait is a close contender in some cases but slightly worse overall. Epidemic only provides competitive delivery rates in the case of the *LazyOffices* topology but overall performs worse than Spray and Wait.

PRoPHETv2 provides the highest delivery rate in the case of the *LazyOffices* topology, a static topology where many hops can be needed but does not perform as good with a changing topology, although this might change with longer test durations.

In all cases the distribution of latencies has outliers with high latencies but can generally be lowered by choosing the right protocol for the use case. While Spray and Wait is a favorite for both the *Event* and the *BusyOffices* topology, there is a tradeoff to be made between delivery rate and latency in the case of the *LazyOffices* topology.

The distributions of client and server time show a wide range of possible times which makes it impossible to draw conclusions what distribution would be desirable.

### 5.6.3. File sharing

Due to a bug in the software all measurement data for one of the 21 devices got accidentally discarded. As in all of the cases this was a stationary device the results are accurate nonetheless as the amount of bundles sent by and delivered to this device evens out due to the nature of our bundle generator.

#### Event topology

The results are shown in Figure 5.5.

Compared to the *Chat messages* case the overall delivery rates are much lower. The best performing protocol regarding both the delivery rate and the latency distribution is Spray and Wait with  $L = 16$ . It provides a mean latency of less than 20 minutes. The Epidemic routing protocol performs worst regarding both of these metrics.

PRoPHETv2 with  $P_{\min} = 0.5$  has comparable latencies to the best Spray and Wait case but a lower delivery rate. All other configurations fall in the range between the best Spray and Wait case and Epidemic with low variation.

Looking at the distribution of server times it becomes evident that large amounts of time were spent in server mode. As every device could connect while a long file transfer was ongoing devices rarely had the chance to leave server mode and connect to other servers themselves. Comparing server and client times it is obvious that the server mode was only entered few times but lasted for long times.

#### LazyOffices topology

The results are shown in Figure 5.6.

In contrast to the *Chat messages* case where the overall delivery rate was higher in the *Event* topology than in the *LazyOffices* topology it is the other way around in the *File sharing* case. Although Spray and Wait with  $L = 16$  provides comparable delivery rates to the *Event* topology it comes with higher latencies than the almost equally well performing other cases of Spray and Wait with the best latency provided with  $L = 4$ .

PRoPHETv2 performs slightly better than Epidemic and Binary Spray and Wait but all three perform worse in general than Spray and Wait. The mean latencies of PRoPHETv2 are close to the mean latencies of Spray and Wait but the delivery rate is lower than in all three Spray and Wait configurations.

The amount spent in server time in comparison to client time is much lower than in the *Event* case but the mean time a server was open leads to the conclusion that most servers were never connected to by clients. The ones that received connections were open for vastly varying times. Client times vary heavily with the routing protocol but offer no correlation to delivery rate or latency.

### **BusyOffices topology**

The results are shown in Figure 5.7.

The variance in the delivery rate between the routing protocols is significantly lower in this moving topology than in all other cases before. On the other hand, the latency distribution varies heavily with Epidemic having the worst latencies by far. Surprisingly the best latencies are provided Binary Spray and Wait together with PRoPHETv2 for  $P_{\min} = 0.4$  with only a slightly smaller delivery rate than Spray and Wait.

Although again a tradeoff has to be made between delivery rate and latency the overall effect on latency outweighs the effect of the routing protocol on delivery rate.

Again, as in the *LazyOffices* topology, server time varies heavily with most servers never being connected to. In this case, client time correlates with latency, so it is possible that latencies for other routing protocols might be improved by limiting client time.

### **Conclusion**

Spray and Wait again performs best overall but the delivery rates are low in all tested cases. It generally performs better as the number of maximum copies  $L$  increases.

In the restricted topologies most servers were never connected to and connections, once established, could lead to long connection times which can possibly hinder the distribution of bundles to other peers.

As already stated the *File sharing* case will become much more interesting in the future once a strategy for optimum is developed to limit the storage space in use.

#### 5.6.4. Comparison

The *Chat messages* and the *File sharing* case show vastly different behaviour in regard to all measured metrics. Although in both cases Spray and Wait performed best regarding delivery rate, in the former case both  $L = 4$  and  $L = 16$  performed significantly better than  $L = 8$  while in the latter case performance generally increased with the number of available maximum copies.

In our choice of topologies and scenarios, the choice of routing protocol depends much more on the topology than on the scenario. Although opptain makes it possible for bundles to choose which routing algorithm they want to use there is no clear choice to be made depending on the data to be sent.

Distribution of client and server time varied vastly as did their ratio. While in some cases a correlation could be deduced between these values and the delivery rate or latency in other cases no correlation was evident.

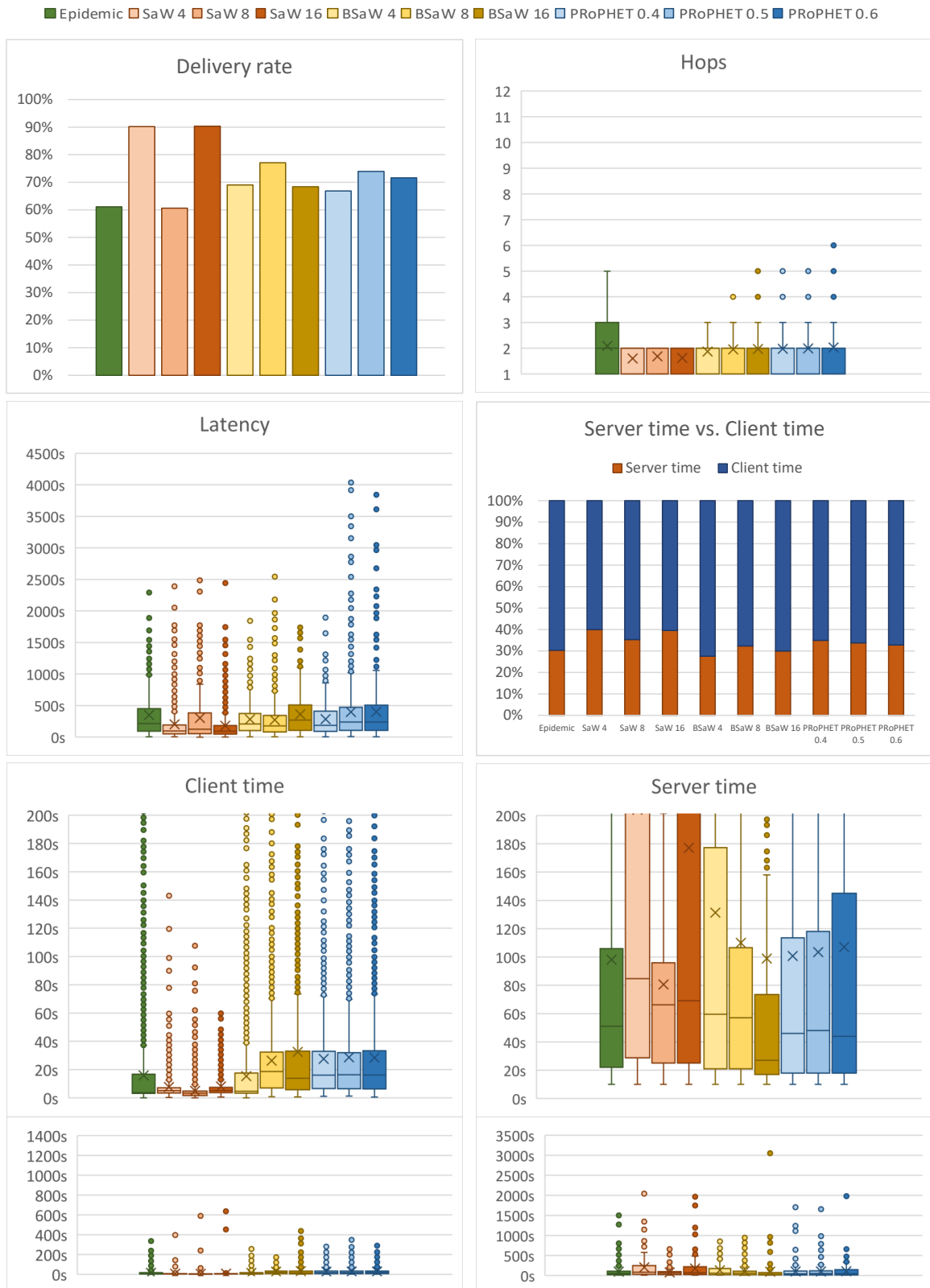


Figure 5.2.: Chat messages / Event topology

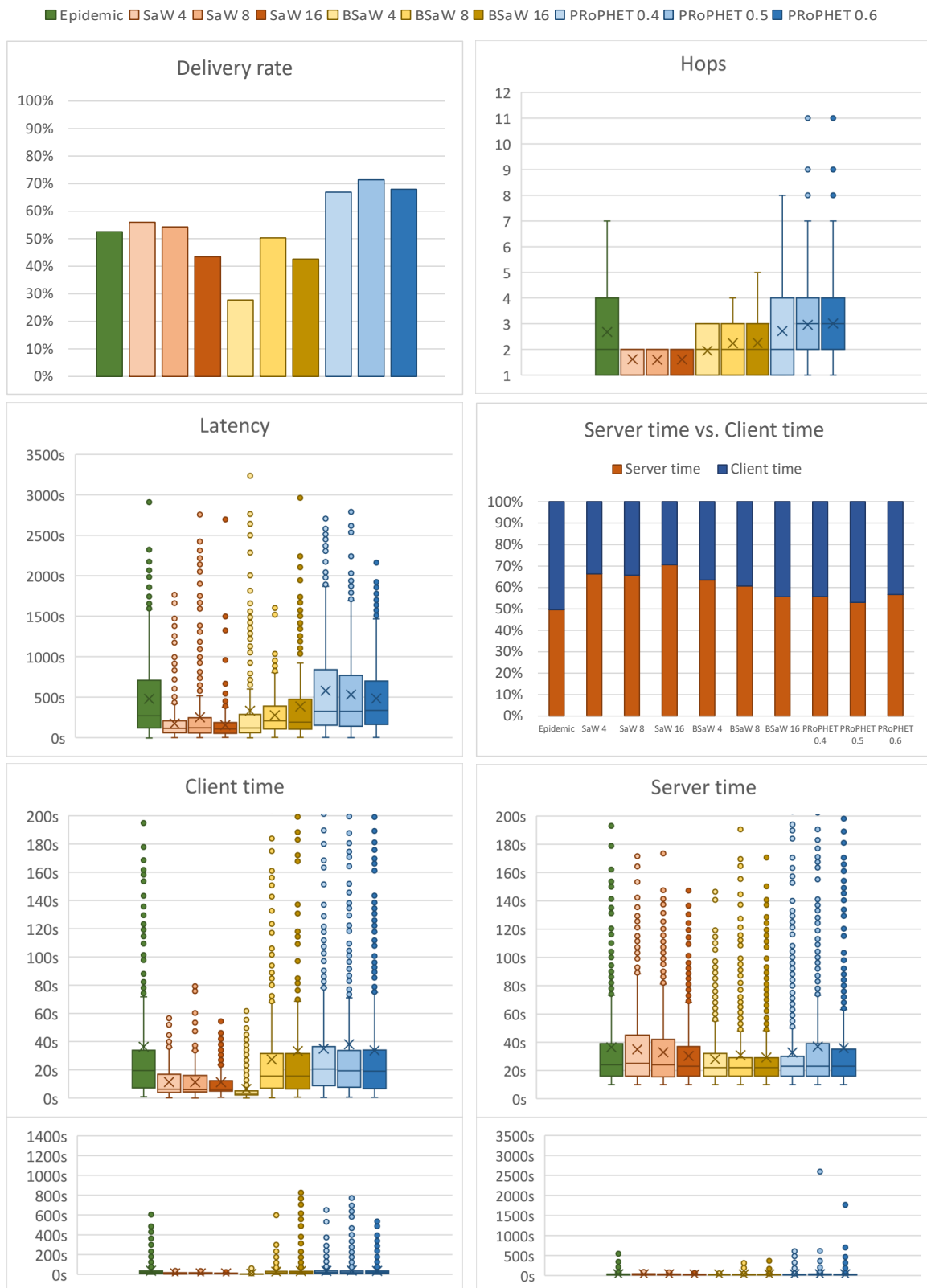


Figure 5.3.: Chat messages / LazyOffices topology

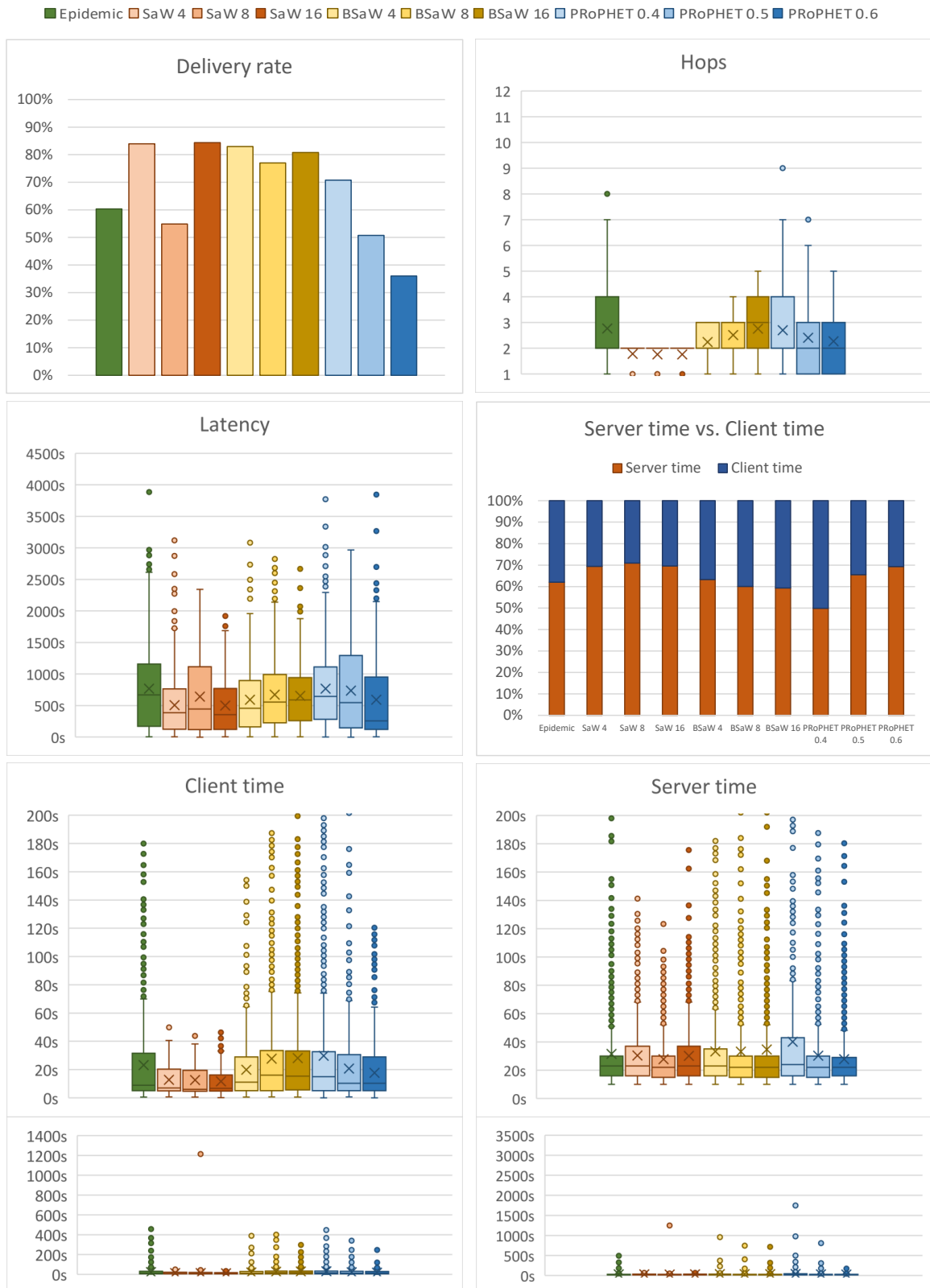


Figure 5.4.: Chat messages / BusyOffices topology



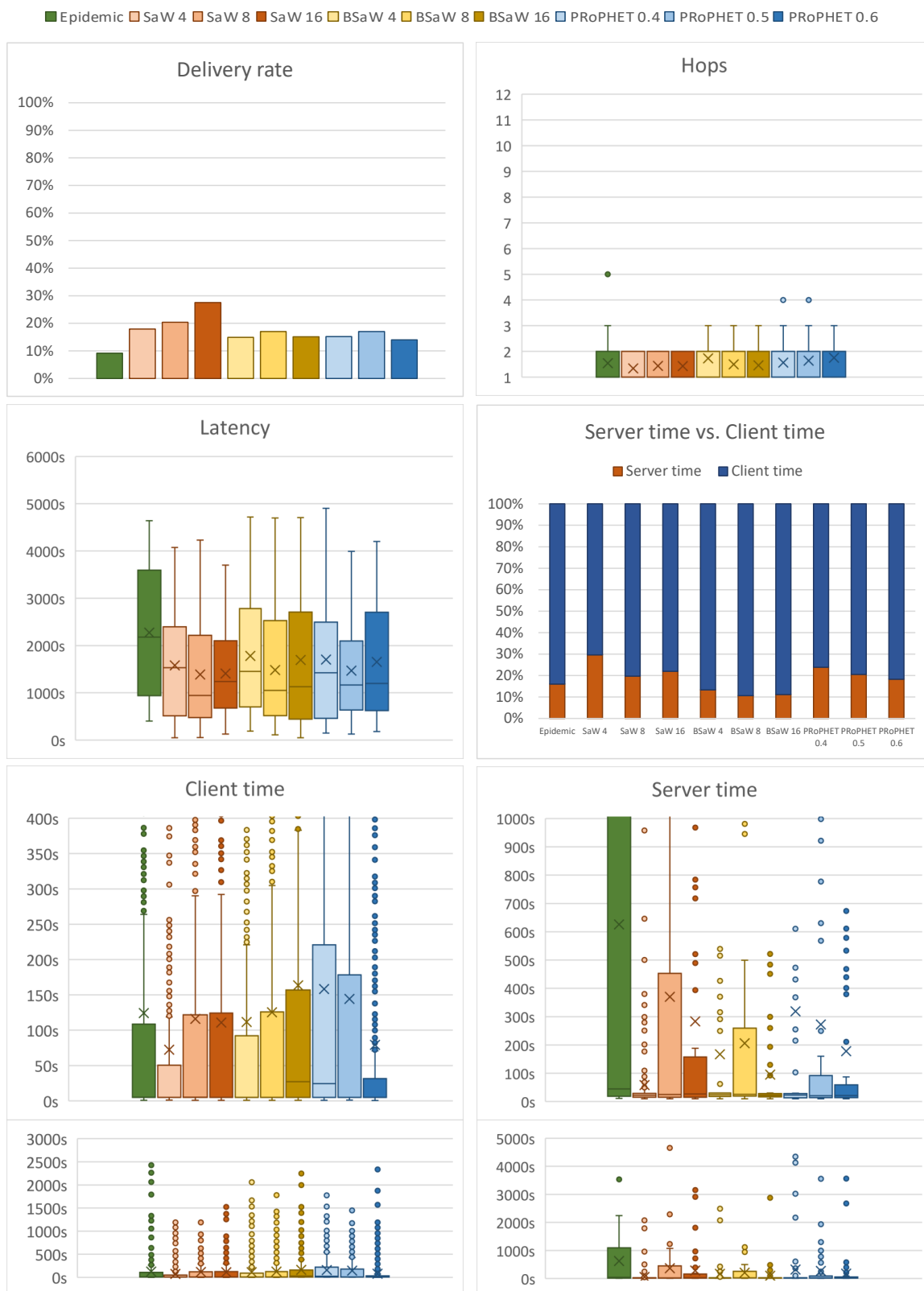


Figure 5.5.: File sharing / Event topology

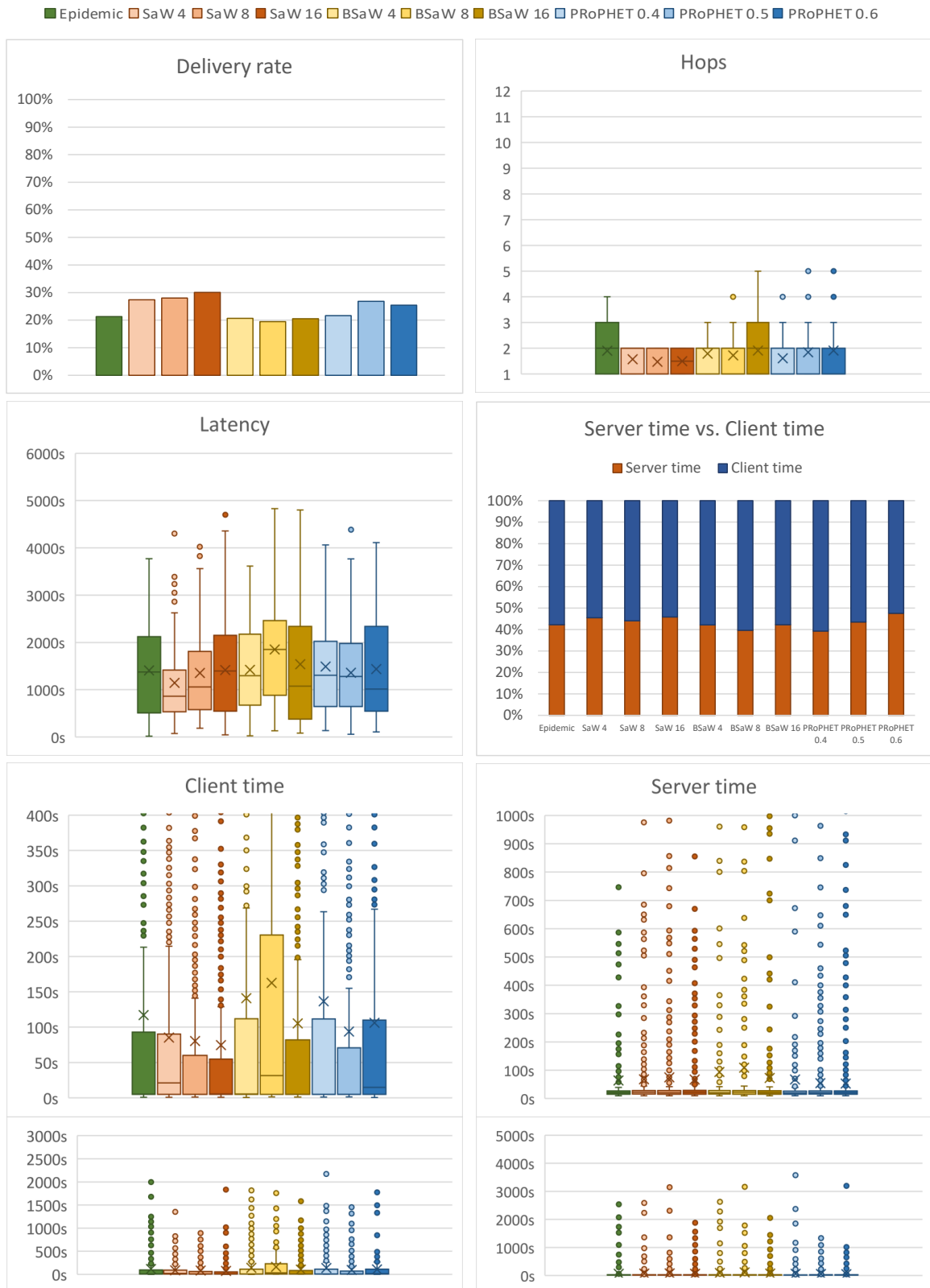


Figure 5.6.: File sharing / LazyOffices topology

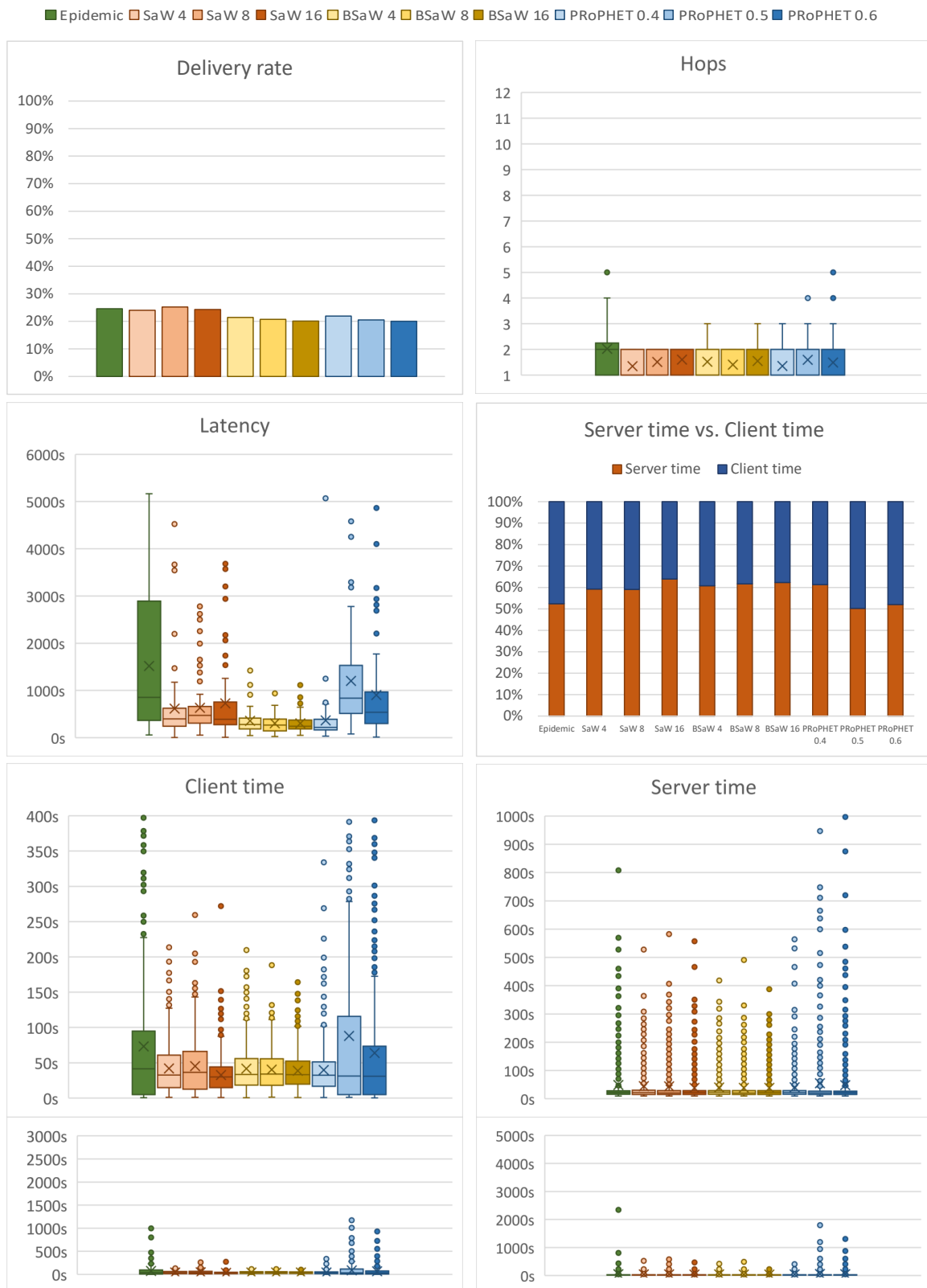


Figure 5.7.: File sharing / BusyOffices topology



## Chapter 6.

# Conclusion and Future Work

### 6.1. Conclusion

In this thesis we both developed the necessary tools for conducting fully automated tests on our opportunistic network and conducted such tests to a certain extent.

We developed applications to support both the ad-hoc based case and the web-based case which includes the bundle generator and its configurability provide a solid implementation of the demanded features. It is easily extensible to record more and different metrics and to be used in other topologies and scenarios. The applications were completely overhauled several times to provide stable and performant software and code of high quality.

The opportunistic network implemented in opptain was improved to be able to run continuously in Automation mode and many bugs were fixed which hindered correct operation. Several routing schemes were put to test and compared to each other in multiple configurations, topologies and scenarios. Existing routing implementations were amended and a completely new one, PRoPHETv2 was implemented.

As emphasis was put on the big picture of opptain not many metrics of interest on the small scale were measured and there are still many use cases to explore. Nevertheless the measurement series conducted provide a solid base for improving the Automation mode of opptain.

### 6.2. Future Work

As already stated in Section 5.1 many problems with opptain and Android persist. There is still a lot to do to improve the stability especially of the opptain's Automation mode.

Based on the conducted tests some possibilities to improve the network performance surfaced. Features such as the limitation of time spent in client and server mode or the limitation of storage space can be implemented in several ways and put to test using *Åggregator*. Another possibility is to randomize the chance of opening a Wi-Fi hotspot instead of always connecting to one if one is available as this could possibly hinder other devices out of reach of the open hotspot to distribute their bundles.

Another field of interest in the future is the comparison of performance on several devices. Our tests devices used from Samsung and Huawei offer varying degrees of adherence to Google's Android documentation.

As all tests conducted in this thesis were conducted using whitelisting no metrics have been recorded which vary depending on the distance between devices. On the one hand devices further apart have a weaker Wi-Fi connection to each other but on the other hand there are less competing Wi-Fi networks when the devices are spread out. It is possible that the delivery rate in the *File sharing* case is much better in this case.

## Appendix A.

# Ågggregator User's Guide

Ågggregator is the main application to provide means of systematical and fully automated testing of the *opptain* opportunistic network application. Once a measurement series is scheduled, opptain is remote controlled to initialize it to a known state and configure it before and stopping it after a test is completed. During a test bundles a generated according to the measurement series configuration.

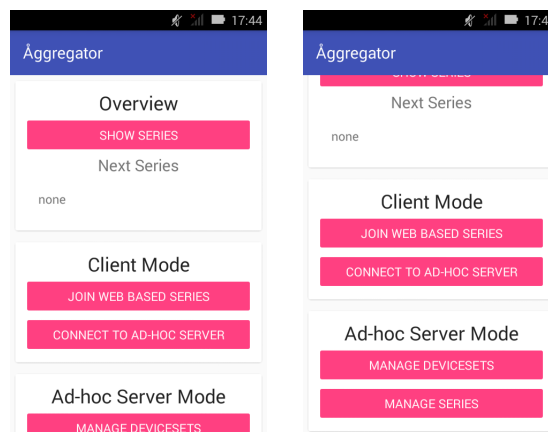


Figure A.1.: Main activity

Once Ågggregator is started it provides the user with an activity split into three parts as seen in Figure A.1. The first part, *Overview*, provides a general overview over measurement series on this device regardless if the device acts as a client or as a server. In contrast the second part, *Client mode*, provides features to use the device as an ad-hoc or web-based client while the last part, *Ad-hoc Server Mode* provides access to functions for using this device in ad-hoc server mode.

In general, whenever a list of items is displayed an action can be triggered or a subview can be entered by clicking an item. Items can be deleted by long-clicking, as seen in Figure A.2. Views can always be left to their hierarchically higher counterpart by pressing the back button of the device.

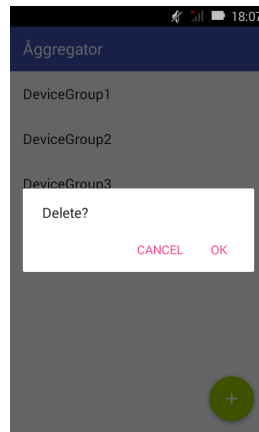


Figure A.2.: Deleting an element

## A.1. Overview

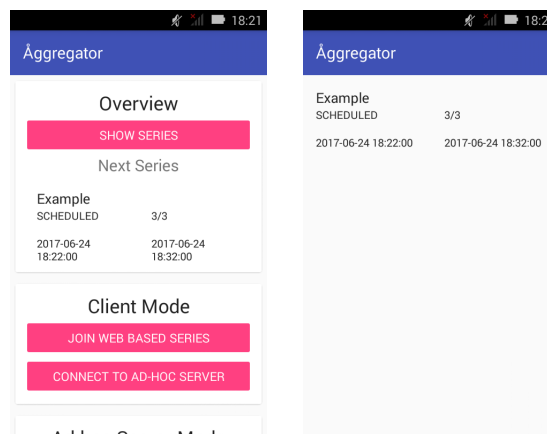


Figure A.3.: Overview

The overview part, as the name implies, provides an overview over all measurement series on this device, whether the device is an ad-hoc server, an ad-hoc client or a web-based client for each specific measurement series as can be seen in Figure A.3. If a series is running or upcoming it is immediately displayed. By clicking on “Show Series” the view switches to a list of all series ordered by begin.

The displayed information for the server mode will be covered later. In client mode the displayed information for each series below its name are from left to right and top to bottom:

- The state of the series, which can be SCHEDULED, RUNNING and FINISHED depending on the current time in relation to the begin and end times,
- the number of devices participating in the series or in web-based mode the time when this information will be pulled from the web server, called the *close* time,



- the time when the series begins and
- the time when the series ends.

## A.2. Client mode

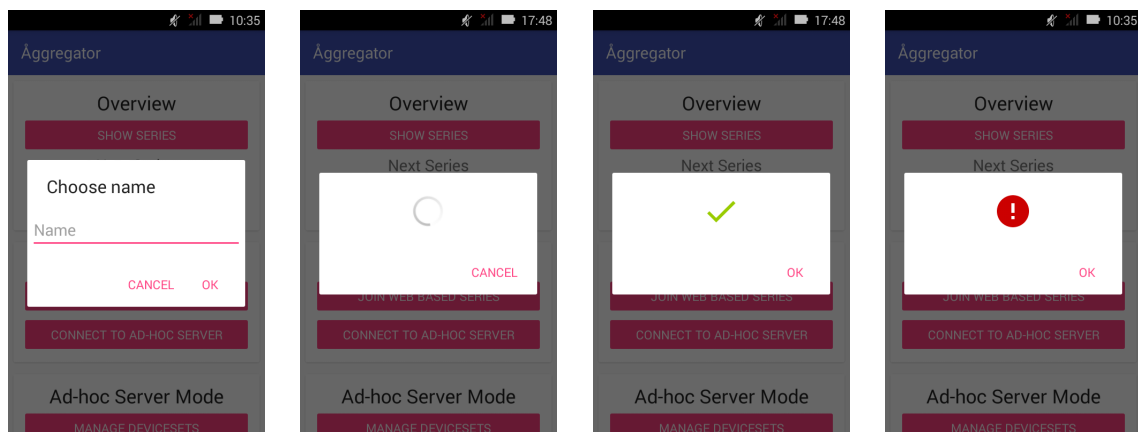


Figure A.4.: Client mode

The client mode only has two actions: “Join Web-Based Series” and “Connect To Ad-Hoc Server”.

If the user chooses to join a web-based series the series name has to be entered as displayed in Figure A.4. If the join succeeds, the measurement series gets scheduled and the user does not have to take any more steps; fetching the list of participating devices, starting and stopping the series and uploading the results to the server will be handled by *Aggregator*.

The option to connect to an ad-hoc server has to be chosen whenever an ad-hoc server waits for connections, as is the case when registering the device, scheduling a measurement series and uploading the results of such a series. If a measurement series is scheduled starting and stopping the series will be handled by *Aggregator* as in the web-based case but uploading of the results is a manual step as the devices have to be in the same physical location to connect directly to each other via Wi-Fi.

In both cases success will be indicated by a green checkmark while failure such as missing network connectivity or non-existence of a series on the server will be indicated by a red exclamation mark.

## A.3. Ad-hoc server mode

### A.3.1. Manage DeviceSets

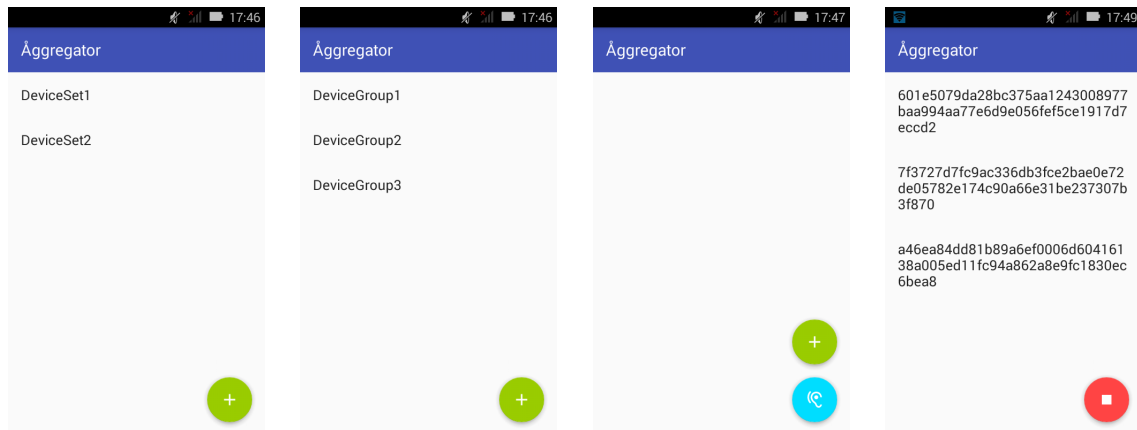


Figure A.5.: Manage DeviceSets

The first step to plan an ad-hoc based measurement series is to determine which devices will be participating in it. This task is achieved by the “Manage DeviceSets” option, its subscreens shown in Figure A.5. First the user gets to a list of DeviceSets where new ones can be created by clicking the green plus button and choosing a name. When clicking on a DeviceSet its contained DeviceGroups are shown, which again can be created by clicking the green plus button and choosing a name.

The general idea behind DeviceSets and DeviceGroups is that every ad-hoc based measurement series references one or more DeviceSets in which only devices within the same DeviceGroup are able to see each others Wi-Fi networks during a test to simulate different topologies without physically moving the devices. We call the entirety of participating devices in a series a *DeviceSet* which is divided into different *DeviceGroups* which in turn contain *Devices*.

Clicking on a DeviceSet leads to its list of devices denoted by their unique opptain device ID. The current device can be added by clicking the green plus button. The teal listen button opens a Wi-Fi hotspot and listens for connections from other devices, which can enter the DeviceSet by using the “Connect to Ad-Hoc Server” discussed earlier. The hotspot can be closed by clicking the red stop button.

A DeviceSet can either be used directly in *simple* measurement series or be exported to JSON to be used in *complex* measurement series configuration files by long clicking on it. An example of an exported DeviceSet is shown in Figure A.6 albeit with abbreviated device IDs.

```

{
  "DeviceGroup1": [
    "601e5079da28bc37...",
    "7f3727d7fc9ac336...",
    "a46ea84dd81b89a6..."
  ]
}

```

Figure A.6.: Exported DeviceSet JSON

### A.3.2. Manage Series

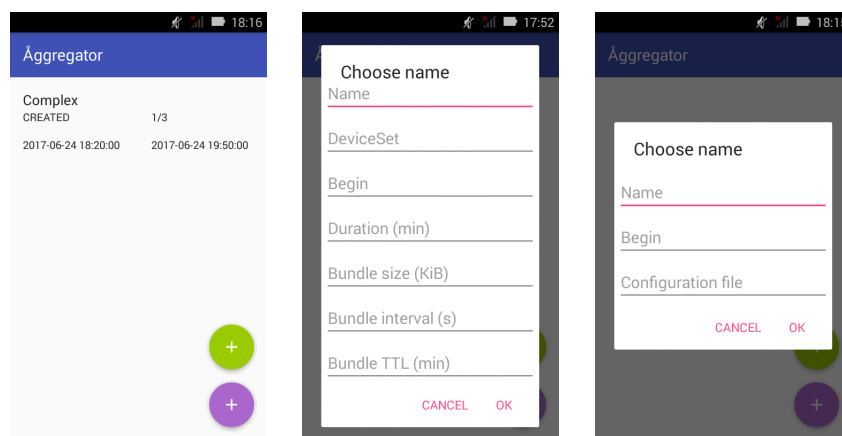


Figure A.7.: Manage Series

By clicking on “Manage Series” an overview over all measurement series is shown where the current device is the ad-hoc server. Such measurement series have two more states in addition to the aforementioned SCHEDULED, RUNNING and FINISHED: CREATED and AGGREGATED. A series is in the CREATED state immediately after creation and will advance to the SCHEDULED state once it is scheduled on every participating device. After a series is finished and its results have been collected the series advances from FINISHED to AGGREGATED.

In contrast to client mode the number of devices displayed is in the form  $x/y$  meaning that a measurement series has been scheduled on or - after the end of a series - data has been aggregated from  $x$  of all participating  $y$  devices.

Clicking on the upper green plus button creates a *simple* measurement series where all necessary configuration can be supplied through the application. This restricts the mode to only one DeviceSet, one bundle interval, one bundle size and the epidemic routing protocol and is intended for quick tests. To plan *complex* series which takes a JSON configuration and does not have these restrictions the user has to click on the lower purple plus button - in this case only the begin and the name of the series are

configured in the menu.

```
{
  "duration": 90,
  "deviceSets": [
    {
      "DeviceGroup1": [
        "601e5079da28bc37...",
        "7f3727d7fc9ac336...",
        "a46ea84dd81b89a6..."
      ]
    },
    {
      "DeviceGroup1": [
        "601e5079da28bc37...",
        "7f3727d7fc9ac336..."
      ],
      "DeviceGroup2": [
        "7f3727d7fc9ac336...",
        "a46ea84dd81b89a6..."
      ]
    }
  ],
  "deviceSetInterval": 20,
  "generators": [{
    "seed": 123,
    "intervalVals": [30,300],
    "intervalType": "loguniform",
    "sizeVals": [1],
    "sizeType": "set",
    "ttl": 30
  }],
  "routingProtocol": "binarysprayandwait",
  "routingL": 8
}
```

Figure A.8.: Configuration JSON

An example for a JSON configuration is provided in Figure A.8, again with abbreviated device IDs. In this case the two DeviceSets will alternate in a 20 minute interval during the 90 minutes long measurement. The packet generator generates packages using the seed 123 with the *loguniform* distribution every 30 to 300 seconds. The size of the generated packages will be a constant 1 KiB and their TTL will be 30 minutes. Note that all durations are given in minutes whereas the interval is given in seconds.

As a routing protocol *binariesprayandwait* with  $L = 8$  will be used. To configure  $P_{\min}$  for the *prophet* (PROPHETv2) routing protocol the parameter *routingMinP* can be used.

After a series has been created it can be scheduled on client devices by clicking on the series to enter its subscreen and following the exact same steps as before when registering devices for a DeviceSet. As a convenience feature not only the series currently selected is scheduled on connecting devices but rather all applicable series.

Scheduled series are conducted in a fully automated way without the need for user operation. To cancel a measurement series and all future ones opptain has to be stopped via its permanent notification and the device has to be restarted - otherwise opptain will automatically be restarted.

Following the end of a measurement series the data can be collected by again entering the series subscreen and again following the same listening steps. Only after data from a device has been successfully transferred to the ad-hoc server device it will automatically be deleted on the client device.

Once all data has been aggregated it can be exported to an easy to examine SQL format by clicking on it which is readable by SQLite 3. All measured data is available as recorded by opptain. The fields which are always present and provided by Åggregator are *\_timestamp* and *\_oppid* which together form a unique identifier and index for each record.



## Appendix B.

# Åggregator-Webb User's Guide

Åggregator-Webb is the web application acting as the server for web-based Åggregator measurement series.

On connection to the web server with a browser the administrator is greeted with a login screen shown in Figure B.1. The credentials are configurable using the *application.properties* file as is common in *Spring* applications. By default user and password are set to *test*.

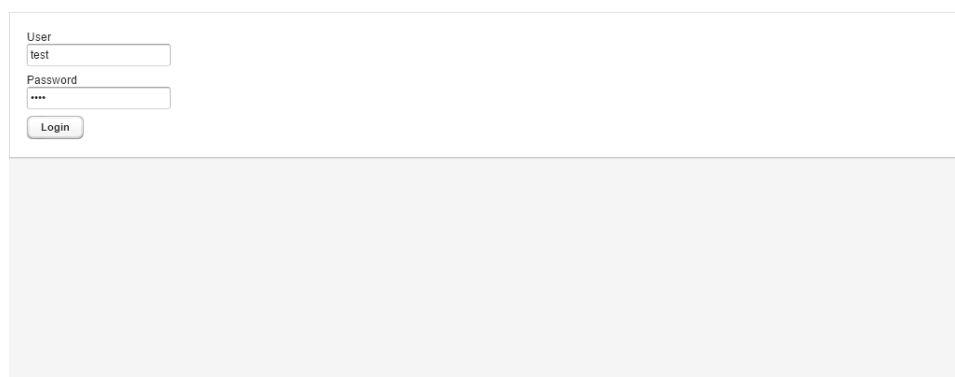
The image shows a web browser window with a login form. The form has two input fields: 'User' with the text 'test' and 'Password' with four asterisks '\*\*\*\*'. Below the fields is a 'Login' button. The background of the page is a light gray color.

Figure B.1.: Login screen

After successful login there is the option to create new measurement series by clicking the “Add” button which opens the dialog shown in Figure B.2. The configuration is like the configuration of *complex* series in the ad-hoc based case in Åggregator with two differences: Any configured Device-Sets are ignored and later overwritten and a *close* time has to be set as a deadline until which devices can join the measurement series.

The main screen is depicted in Figure B.3. Every measurement series is represented by an entry in the table view. In addition to the name, the close time, the begin time, the end time, the number of registered devices and the number of devices which have uploaded their measurement values are

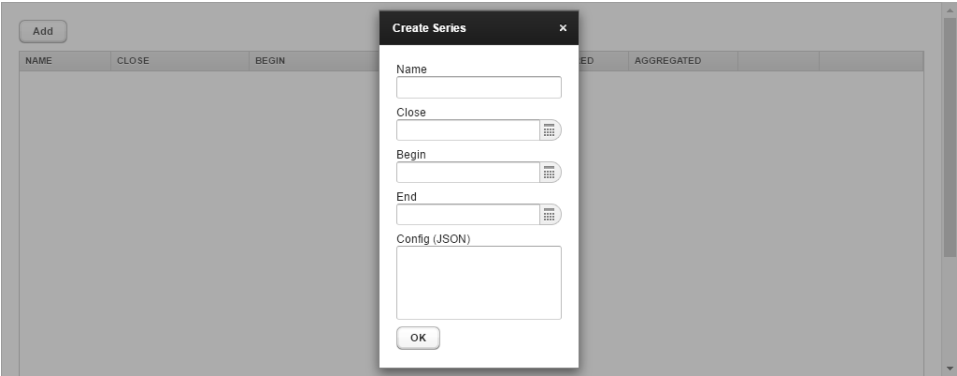


Figure B.2.: Add screen

shown. A series can be deleted by clicking the *Delete* button and exported to SQL by clicking the *Export* button.

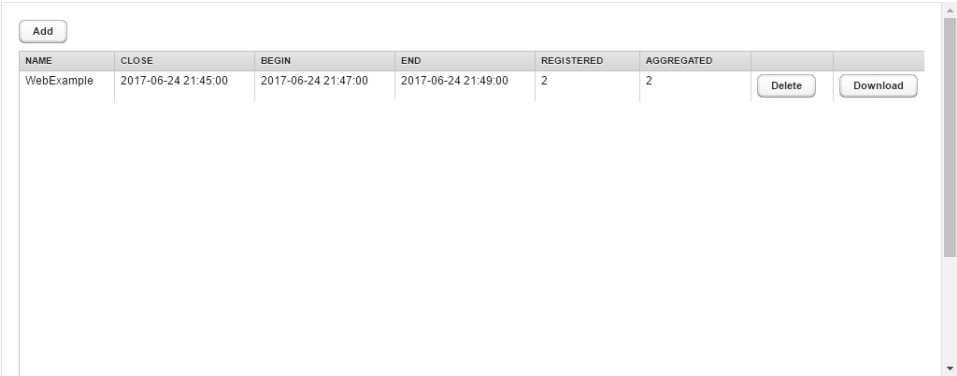


Figure B.3.: Dashboard screen



## Bibliography

- [ANDa] *Android Developers Blog: Announcing the Android 1.0 SDK, release 1.* <https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html>.
- [ANDb] *Introduction to Android | Android Developers.* <https://developer.android.com/guide/index.html>.
- [GDL11] GRASIC, Samo; DAVIES, Elwyn; LINDGREN, Anders; DORIA, Avri: The Evolution of a DTN Routing Protocol - PROPHETv2. In: *Proceedings of the 6th ACM Workshop on Challenged Networks*, 2011, S. 27–30.
- [HLT08] HUANG, C. M.; LAN, K. c.; TSAI, C. Z.: A Survey of Opportunistic Networks. In: *22nd International Conference on Advanced Information Networking and Applications - Workshops (aina workshops 2008)*, 2008, S. 1672–1677.
- [Ipp15] IPPISCH, Andre: *A fully distributed Multilayer Framework for Opportunistic Networks as an Android Application*, Department of Computer Science, Heinrich Heine University Düsseldorf, Diplomarbeit, März 2015
- [LDD12] LINDGREN, Anders; DORIA, Avri; DAVIES, Elwyn: *Probabilistic Routing Protocol for Intermittently Connected Networks*. RFC 6693 (Experimental). <http://www.ietf.org/rfc/rfc6693.txt>. Version: August 2012 (Request for Comments).
- [LDS03] LINDGREN, Anders; DORIA, Avri; SCHELÉN, Olov: Probabilistic Routing in Intermittently Connected Networks. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 7 (2003), Juli, Nr. 3, S. 19–20.
- [SPR05] SPYROPOULOS, Thrasyvoulos; PSOUNIS, Konstantinos; RAGHAVENDRA, Cauligi S.: An efficient routing scheme for intermittently connected mobile networks. In: *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, 2005, S. 252–259.

- [VB00] VAHDAT, Amin; BECKER, David: Epidemic routing for partially connected ad hoc networks / Department of Computer Science, Duke University. 2000 (CS-2000-06). Forschungsbericht.

# **Ehrenwörtliche Erklärung**

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 28.June 2017

Gian Perrone



Hier die Hülle  
mit der CD/DVD einkleben

**Diese CD enthält:**

- eine *pdf*-Version der vorliegenden Masterarbeit
- die  $\text{\LaTeX}$ - und Grafik-Quelldateien der vorliegenden Masterarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Masterarbeit erstellten Software  $\text{\AA}$ ggregator und  $\text{\AA}$ ggregator-Webb
- die zur Auswertung verwendeten Konfigurationen und Datensätze
- die Websites der verwendeten Internetquellen