OppNet: An Energy Optimized Service Platform for Opportunistic Networking on Android

Master Thesis

MA-2014-01

Fabian Brun

14.08.2014

Advisor: **Sacha Trifunovic**
Supervisor: **Prof. Dr. Bernhard Plattner**

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

Opportunistic networking can offer connectivity even when no traditional infrastructure (e.g., cell towers or hotspots) is available. It exploits direct links between wireless devices in range and the mobility of nodes to distribute messages towards their target, using a store-carry-forward approach. There is an ever-expanding number of mobile devices capable of establishing opportunistic connections, yet no major operating system allows to automatically take advantage of it today. To change the status quo two very important aspects need to be covered: The solution must be very energy-efficient, and it must allow other developers to easily build apps upon it. Both aspects have mostly been ignored in research prototypes, so in this thesis we present a service platform for Android devices which features energy-efficient opportunistic neighbor discovery.

The platform is derived from an existing opportunistic discovery solution based on WLAN, but with a strong focus on optimizing its power consumption. We introduce duty-cycling and the use of Bluetooth to the discovery process. The effectiveness of these optimizations is measured, and policies are established to let end-users impose energy constraints on the platform. Evaluating the chosen policies shows a power consumption which allows for long-term operation of the platform.

Finally, the service layer enables developers to write rich applications without the need to handle single connections or to perform service discovery on their own. The platform is complemented by built-in elliptic curve cryptography, a graphical interface to control the platform, a small library which hides away the additional complexity of interacting with the core platform, and a proof-of-concept application demonstrating the usage of said library.

Acknowledgements

I would like to thank my advisor Sacha Trifunovic for his great support during the past six months. His extensive experience with opportunistic networks and their real-world application helped me to get into the subject quickly while keeping focus on the feasible ideas. I would also like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to conduct my Master Thesis at the Communication Systems Group.

Special thanks go to all my friends who helped to distract my mind from time to time – you are amazing!

Contents

List of Figures

List of Tables

1. Opportunistic Networking

Modern communication systems usually require some kind of infrastructure (e.g., the Internet) to deliver messages or connect devices, even if they are all in close proximity to each other. This central infrastructure represents a single point of failure – if it fails, any communication fails as well. Even though today's smartphones feature the technology needed to establish ad-hoc networks with neighboring devices, this way of peer-to-peer networking is not widely used. By exploiting the dynamic nature – people moving around with their devices – of such *opportunistic* networks, many new applications would be possible:

- In case of a natural disaster, such as an earthquake, which destroys communication infrastructure, opportunistic networking may help to uphold communication relay calls for help, and allow for the coordination for rescue operations.
- In regions where no infrastructure exists yet, opportunistic networking could connect locals nevertheless.
- When infrastructure is shut down (either deliberately, such as to oppress free-speech movements, or simply overloaded), opportunistic networking allows people to organize themselves.
- Multiplayer games or messaging apps can use opportunistic networks to find other participants without the need of an intermediary server.

Mobile operating systems such as Android have built-in support for some of the technologies one could use to build opportunistic networks, but there is no common interface to easily incorporate ideas based on opportunistic communication into third-party apps. This thesis tries to fill this gap by delivering an Android platform which enables developers to focus on the high-level principles of opportunistic networking without having to care about low-level chores.

1.1. Goals

Opportunistic networking is in a typical chicken-or-egg situation: The technology will only be adopted if a critical mass is using it (delivering sufficent connectivity everywhere). The critical mass only installs such a solution if the ecosystem proves to be *useful* to them (available apps, amongst other things). However, developers prefer to build apps for ecosystems where they can reach the most users.

All existing approaches provide basic connectivity, but this only satisfies part of the *usefulness* requirement. Unfinished products repell end-users, so this thesis seeks to solve the higher-level problem of *user acceptance* by pursuing the following three goals:

Goal 1: Optimize energy footprint.

The dynamic environment implicit to opportunistic networking (i.e., discover and connect to neighbors when they come in range) calls for always-on operation in the background. This comes at the cost of constant energy consumption. Smartphone batteries are already challenged to cope with the average daily usage, and users will just turn off features which drain their batteries above a certain threshold. It is therefore crucial to keep energy consumption as low as possible, especially in situations where opportunistic networks do not deliver benefits.

Goal 2: Provide a service layer.

A platform for opportunistic networking must also be useful to developers. While handling the discovery of neighboring nodes is a good starting point, the power of a good platform lies in giving the developer a different view on the network, i.e., the capabilities of surrounding devices. Connecting neighbors makes the most sense if they run the same applications and services. How the real connection between such neighbors is made is then no longer relevant. A service layer allows developers to take advantage of opportunistic networks on a much higher level: It allows them to offer (and interact with) services instead of handling low-level communication.

Goal 3: Support as many devices as possible.

Limiting the supported devices means limiting the potential of the whole platform, because it means less devices supporting the network. Therefore modification of the operating system must not be required. Android ships in many different versions, most of which are still in use. As of February 2014, 80% of all active devices were at least running Android 4.0, and 99% were at least running Android 2.3.3 [2]. Figure 1.1 shows that the adoption rate of Android 4.0+ is not fast enough to justify dropping support for Android 2.3.3 yet: In August 2014, still about 14% were running this older version of Android. A number of test devices (listed in Table 1.1) are used to ensure compatibility with all major Android versions.

Vendor/Device	**ROM**
Samsung Galaxy S	Android 2.3.6 (Samsung TouchWiz UI)
LG Optimus 4X HD (P880)	Android 4.0.3 (LG Optimus UI)
Samsung Galaxy Nexus	Android 4.1.1 (Stock, rooted)
Samsung Galaxy Nexus	Android 4.2 (Stock)
Samsung Galaxy Nexus	Android 4.3 (Stock, rooted)
Asus Nexus 7	Android 4.4 (Stock)

Table 1.1.: Devices used to test Android compatibility

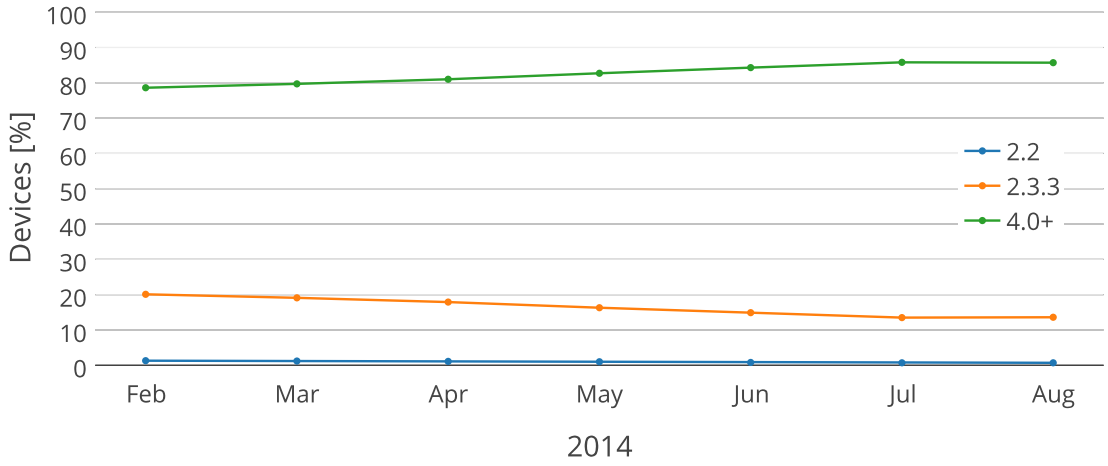


Figure 1.1.: Android distribution per month in 2014 [1]

1.2. Related Work

This thesis is mainly building upon *WLAN-Opp*, an Android platform created by the Communication Systems Group (CSG)[1] of the Computer Engineering and Networks Laboratory (TIK)[2] at ETH Zurich. The fundamental idea, as developed by Trifunovic et al. [3], has been implemented into an Android application [4]. It achieves opportunistic neighbor discovery with WLAN, using the access point mode (which is available since Android 2.3) or public hotspots when available. Based on a study conducted at Aalto University [5], which investigated ways to use link-local communication on commercial hotspots, WLAN-Opp then received connectivity improvements. Of special interest was the neighbor detection performance when using public hotspots, which was assessed with real hotspots found all across Zurich [6].

While the platform so far focused on delivering good connectivity (finding as many neighbor nodes as fast as possible), the energy consumption only got little attention. A comparison between different wireless peer-to-peer technologies by Trifunovic et al. [7] hints at areas where and how to improve the energy consumption of the platform (see Section 3.2 for details).

The service layer is losely inspired by *Multicast DNS* [8] and *DNS Service Discovery* [9] to provide a decentralized, zero-configuration service discovery mechanism. The cryptography parts are built using Dan Bernstein's crypto library NaCl [10], which features a ready-to-use cryptosystem providing authenticated public-key encryption.

[1] http://www.csg.ethz.ch
[2] http://www.tik.ee.ethz.ch/

Aside from WLAN-Opp, there are a few other projects which offer opportunistic networking platforms on mobile devices:

- The *Serval Project* emerged in response to a catastrophic earthquake in Haiti in 2010, with the goal to bring opportunistic communication to people suffering from disaster situations [11]. The project's mesh network technology covers multi-hop communication, encrypted voice calls and file transfers. It uses the WLAN client mode, access point mode, and on rooted devices additionally WLAN ad-hoc mode. The software is not specifically designed to save energy, as its primary focus are emergency scenarios. It is available as open-source, although it is not targeted at third-party app developers.
- *SCAMPI*, a EU-funded project, is a service platform with a HTML5 application development framework [12]. It runs on WLAN as well as on Bluetooth and is platform-independent (e.g., not tied to Android). The development stopped with the end of the EU project.
- *PodNet* is a platform for content dissemination in an opportunistic way. It also features a secure version [13] which allows for finer grained control over content channels (e.g., read-only access to a channel) and a reputation system to fight unsolicited electronic messages (i.e., spam). It is not under active development anymore.

There also exist some Android applications leveraging opportunistic networks without providing a full-stack platform:

- *FireChat* is a messaging app which allows users to chat "off-the-grid" with other users in the neighborhood. It is also available for (and interoperable with) iOS. It uses a proprietary mesh networking framework which is built on top of WiFi Direct and Bluetooth.
- The messaging app *TinCan* solely uses opportunistic communication, where users need to subscribe to other users before receiving their messages. It uses the WLAN access point mode and requires Android 4.0.
- *Uepaa* describes itself as "outdoor safety app". It is not a purely opportunistic app, but has a feature to contact neighboring devices in case of an accident. The proprietary discovery mechanism is based on WLAN-Opp.

2. WLAN-Opp Platform Description

The basis of this thesis is the WLAN-Opp platform, as explained in Section 1.2. The first part of WLAN-Opp is the *core*, an Android application package which contains all the code for actually performing opportunistic neighbor discovery. This package also includes a graphical interface to change operational settings of the platform. The second part of the platform is a *library*, which enables other programmers to integrate the opportunistic features provided by the core into their own apps (also called *client apps*). It encapsulates all inter-process communication between the client app and the core.

The rest of this chapter briefly summarizes the key operation of WLAN-Opp and highlights its main limitations for reaching the goals as stated in Section 1.1. To help distinguish WLAN-Opp from the service platform developed in this thesis, the extended platform has been renamed to *OppNet*. Another reason for renaming the platform is that the extensions in this thesis break compatibility with apps developed for the original platform.

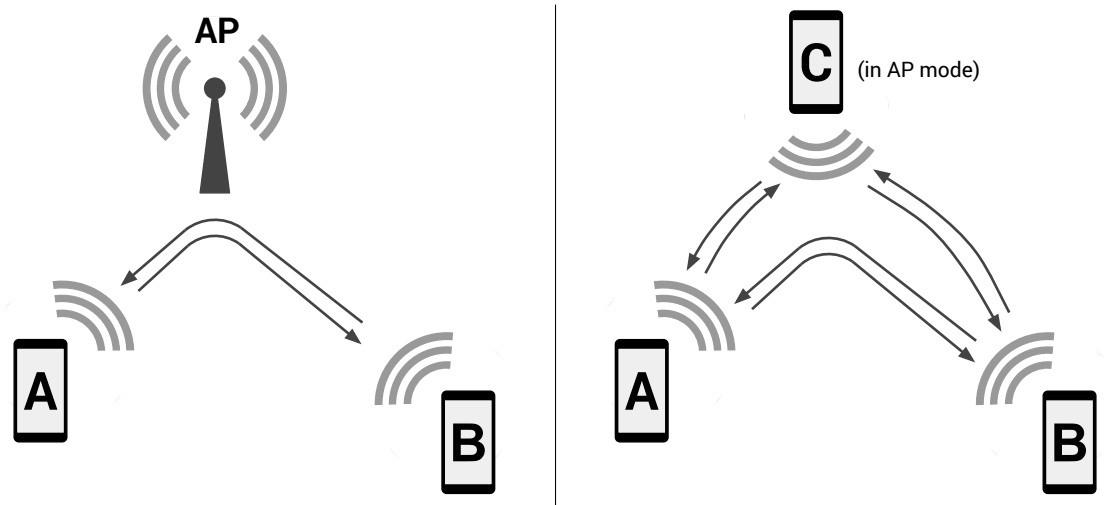
2.1. Basic Principles

To enable opportunistic discovery of neighbor devices, WLAN-Opp takes advantage of the WLAN access point mode. Discovery is performed in two operation modes, *Infrastructure mode* and *Tethering mode*, which are depicted in Figure 2.1.

The *Infrastructure Mode* is used as long as there is any connectable WLAN network available (e.g., open networks in urban areas). Note that such a network does not need to provide internet connectivity - it is only necessary that the connected devices can reach each other locally.

In case there is no pre-existing WLAN infrastructure to use (or no neighbor discovery was possible), the platform can switch into *Tethering Mode*. Here, one of the WLAN-Opp enabled devices takes over the role of the access point, using a feature called *tethering* which is supported by Android since version 2.2 [14]. Other WLAN-Opp devices will detect such an access point through its name (the platform uses a special SSID) and then connect to it eventually.

After having connected to a WLAN network (in either modes), other devices are discovered by broadcasting *beacons*. When receiving a beacon from a neighboring device (or *node*), WLAN-Opp notifies listeners (the client apps) on the device it is running



(a) Infrastructure Mode

(b) Tethering Mode

Figure 2.1.: Operation modes of WLAN-Opp

about the new neighbor. These client apps can then use this information, e.g., to send data to the neighbor.

To increase the likelihood of discovering neighbors, the platform periodically scans for other available networks or becomes an access point itself.

2.2. Core Components

The WLAN-Opp platform core consists of five main components:

- The *NetworkManager* handles all network-related management operations (e.g., turning on/off WLAN). This component is a layer on top of several regular Android networking APIs. It also allows access to some officially hidden APIs in the Android framework, such as manipulating the tethering mode of the device.
- The *BeaconingManager* handles creating and sending as well as receiving the discovery beacons. The data sent in such a beacon includes information about the current network connectivity, which is delivered by the `NetworkManager`.
- The *WlanOppCore* manages the lifecycle (starting/stopping) of the whole platform. Additionally, its API mirrors most the APIs of both the `NetworkManager` and the `BeaconingManager`, providing a single API to control every aspect of the platform.
- A *Strategy* contains the actual logic to establish an opportunistic network (e.g., when to switch networks, or in which intervals discovery beacons should be sent).

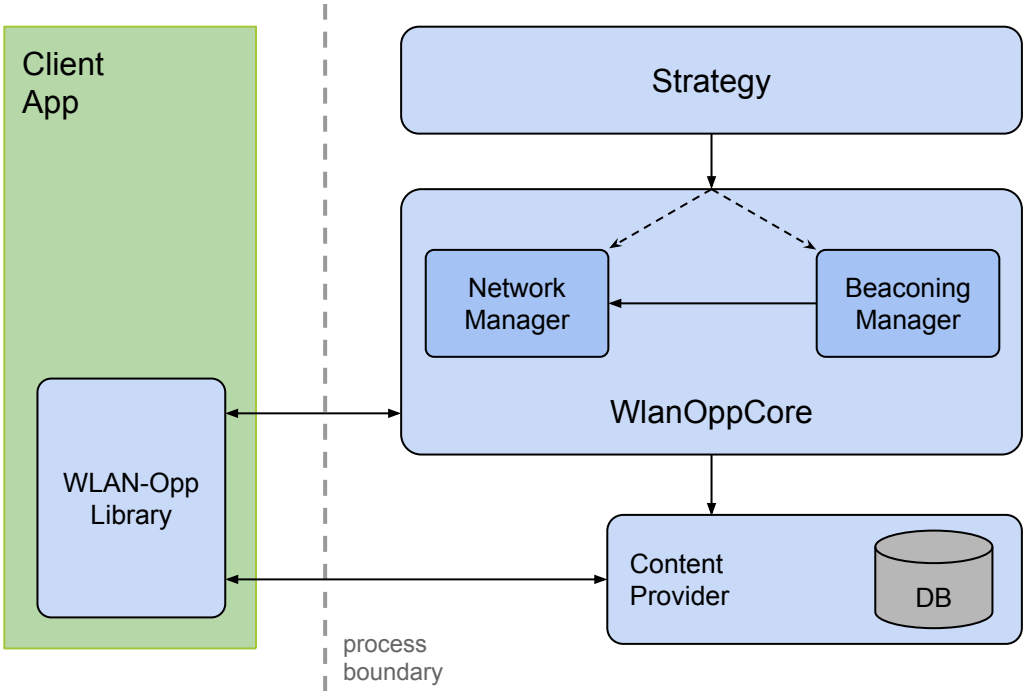


Figure 2.2.: Essential components of WLAN-Opp

It heavily uses the unified API delivered by the `WlanOppCore` to access both the `NetworkManager` and the `BeaconingManager`. There are a few strategies available to choose from, most of them disabling or forcing one certain feature of the platform (e.g., never turn on the access point mode). These usually extend a *default strategy*, which uses all possible features in the most efficient way for overall connectivity.

- The *Content Provider* controls all accesses to platform data (reading as well as writing, especially sanitizing input data). It is based on a native Android component, which also allows to automatically be notified of changes in the database. When for example a new neighbor device was found, and the library was set up to subscribe to this kind of data changes, then the client app gets an instant notification of the new neighbor.

Figure 2.2 demonstrates the dependencies between these components.

2.3. Limitations

Strategies are the most powerful piece of code in WLAN-Opp and were usually the starting point when extending WLAN-Opp. However, achieving the goals specified in Section 1.1 demand a different kind of flexibility in the overall design.

First of all, WLAN-Opp is not specifically designed to save on the device's battery. This is largely due to the way its lifecycle is managed: Switching it on needs interaction from the user (through the graphical interface) or a client app requesting it (through a specific library call). After that, the chosen strategy is constantly executed, until either the user or a client app stops WLAN-Opp again. The `WlanOppCore` has no way of *pausing* its operation or otherwise managing the energy consumption, as this is under the running strategy's control. Having a seperate strategy which just does nothing is not going to work: Switching between strategies requires the platform to be stopped and restarted again. WLAN-Opp cannot do this on its own.

Secondly, WLAN-Opp has no intrinsic concept of client apps. For the data exchange layer to be useful the platform must know about the presence and capabilities of installed client apps. For the data exchange to be efficient it additionally needs a way to learn about client apps on its neighbors. For data security purposes it is crucial that the data on the device can be protected from different client apps. A mechanism to help the platform recognize the calling client app is missing in WLAN-Opp. Instead, it considers providing security and privacy being the job of client apps.

Ultimately, WLAN-Opp's graphical interface offers good options to developers, but is too complex for an end-user. It should primarily display information about the current platform state. The end-user should not need to perform extensive configuration to get the platform running.

3. Energy Optimization

The constant energy consumption of platforms like WLAN-Opp is one of the biggest barriers for their adoption, as discussed in Section 1.1. The major contribution of this thesis is the introduction of *duty-cycling* to the discovery process. The concept refers to having a system which is only active for a portion of its complete runtime. For OppNet this is a substantial lever for the energy saving efforts. Additional savings result from carefully fine-tuning the techniques used to establish opportunistic connectivity and optimizing the overall design of the platform. This chapter explains all the energy-related modifications the OppNet platform received compared to its predecessor outlined in Chapter 2. Furthermore the final performance is evaluated and the collected data is presented. The chapter is rounded off with a discussion of the evaluation results and their implications.

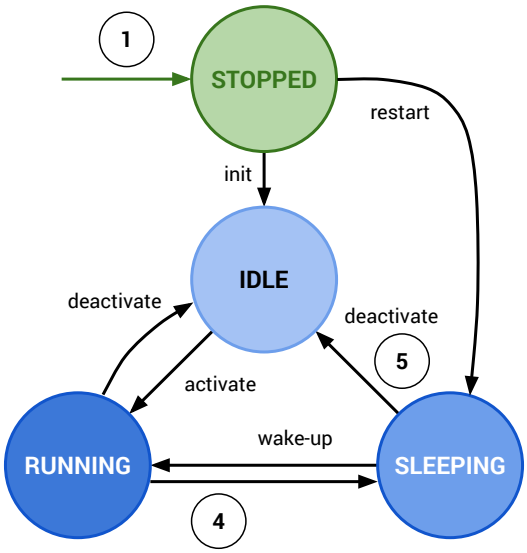
3.1. Duty-Cycling

Enabling the platform to put itself to sleep is a fundamental change in the design of OppNet opposed to WLAN-Opp. First and foremost, the platform needs to handle its lifecycle by itself. A client app may request *that* the platform should start looking for neighbors, but it must be the platform's own choice *when* this happens. In addition, the platform should pause *all* activities when sleeping to get the most out of the duty-cycling. Therefore OppNet introduces the `Supervisor`, an Android background service which ultimately controls the platform's execution.

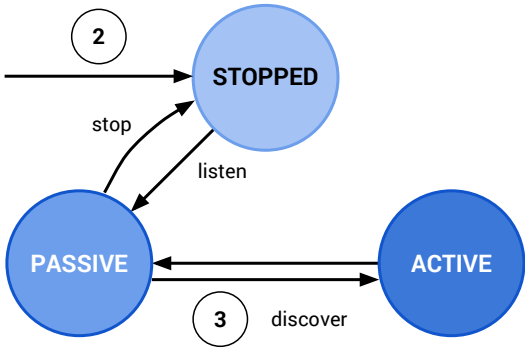
The main purpose of the `Supervisor` is to make sure that the rest of the platform is only fully awake during a 1-minute *discovery period*, which is initiated by the `Supervisor` itself in wake-up intervals of $t_{cycle} \in \{2, 5, 10\}$ minutes. This works under the assumption that all participating devices have synchronized clocks (in the order of seconds). Thanks to network synchronization protocols (NITZ [15] or NTP [16]), this is mostly true for Android devices recently connected to the internet or a cellular network. To cope with unsynchronized devices, the wake-up interval can be randomized.

The `Supervisor` is a lightweight layer below the `BeaconingManager` (which existed previously, but has been revamped as detailled in Section 3.2). Figures 3.1a and 3.1b show the state machines for each layer, while Figure 3.1c describes the dependencies between them[1].

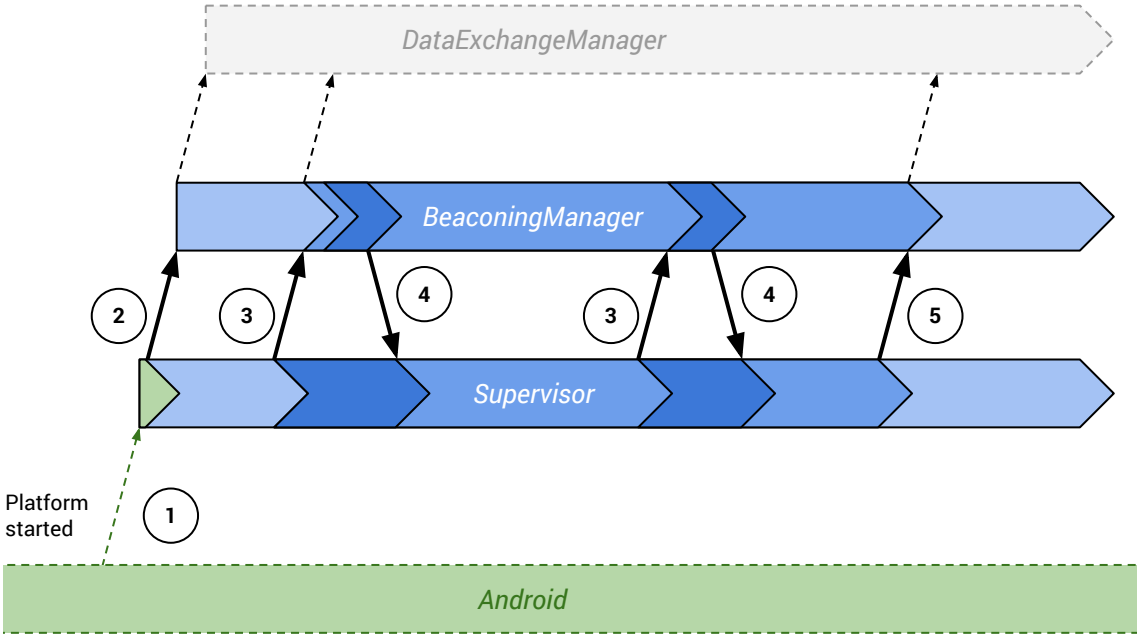
[1]The DataExchangeManager layer will be introduced in Chapter 4



(a) Supervisor state machine



(b) BeaconingManager state machine



(c) Lifecycle Management

Figure 3.1.: Overview of the components used in OppNet duty-cycling

(1) Android is instructed to automatically launch the `Supervisor` before starting any other part of the platform, putting the `Supervisor` into the `IDLE` state.

(2) The now initialized `Supervisor` starts the `BeaconingManager` in the `STOPPED` state. The platform is now ready to be activated (e.g., by the user switching it on through the admin interface).

(3) Eventually the `Supervisor` gets activated (or woken up from sleeping), which in turn puts the `BeaconingManager` into the `ACTIVE` state (via the `PASSIVE` state, if it was `STOPPED` before). This results in neighbor discovery being performed.

(4) After neighbor discovery finishes (after 1 minute the latest), the `BeaconingManager` transitions back into the `PASSIVE` state. The `Supervisor` is notified as well: A wake-up alarm is registered with Android, and everything is put to sleep.

(5) When the platform gets switched off, the `Supervisor` stops the `BeaconingManager` and returns to its own `IDLE` state. Unless the platform is reactivated later by the user, it will not do any neighbor discovery.

3.2. Optimizing Discovery

OppNet inherits the WLAN-based discovery mechanism from its predecessor, and additionally uses Bluetooth as a secondary discovery channel. While the latter already features a reasonable low energy consumption for discovery, the former should particularly profit from duty-cycling: As Trifuovic et al. pointed out [7], the WLAN access point mode consumes a lot of energy and scanning is also rather expensive, compared to Bluetooth. OppNet allows for more flexibility in chosing the energy penalty by seperating the WLAN *client mode* from the *access point mode* feature: The first allows a device to switch only between networks found through WLAN scans, the second allows a device to turn into an access point. Both features can be freely combined with the Bluetooth feature to form policies. A *policy* describes the energy budget the OppNet platform is allowed to use for discovery, and along with the usable features it defines the wakeup interval.

Note that from a connectivity point of view, both technologies have reason to co-exist: WLAN covers longer distances and achieves higher throughput, and Bluetooth promises low energy consumption for the discovery process. Using both also enhances the chance of mutual discovery, as there are situations where one technology fails to discover neighbors (e.g., between devices running different Bluetooth stacks). The `BeaconingManager` uses both technologies in different ways during the 1-minute discovery period.

3.2.1. WLAN Discovery

WLAN discovery splits the 1-minute period into four equally long slots of 15 seconds each. In the beginning of each slot the `BeaconingManager` picks the best action for this slot based on the previous slot. The general rules are:

- If neighbors have been found in the previous slot, stay on the same network. Otherwise (and if permitted by the policy in effect) connect to the next best available network (OppNet access points before other open networks, ordered by signal strength).
- If no more networks are available (or switching networks is forbidden), and if permitted by the chosen policy, turn on the access point mode. Otherwise, terminate WLAN discovery.
- When in access point mode, if no neighbors have connected after two slots, turn access point mode off again.

Connecting to a network can sometimes take a long time. The slot length of 15 seconds ensures that in most cases the WLAN adapter will have enough time to connect to a network and send a few beacons before the end of the slot.

The discovery cycle is terminated after at most four slots. Figure 3.2 shows two exemplary WLAN discovery cycles:

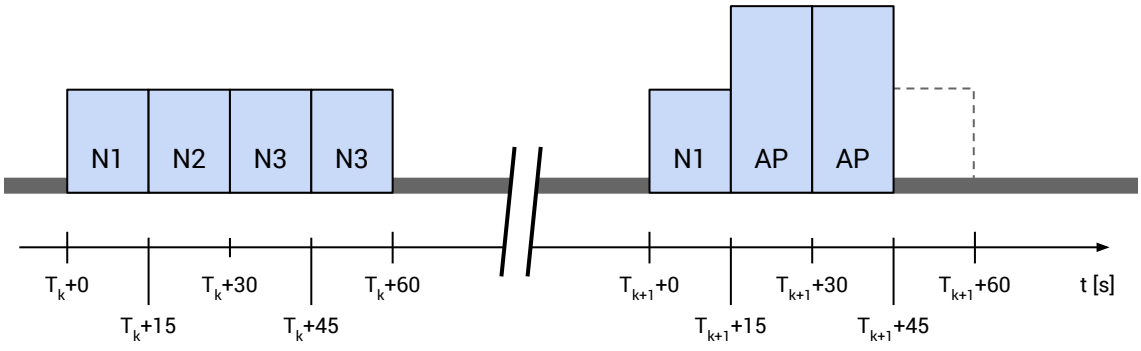


Figure 3.2.: Example of WLAN discovery process

1. At T_k, the `BeaconingManager` starts discovery by connecting to network $N1$. Because no neighbors can be found in the next 15 seconds, it switches to network $N2$, where also no neighbor can be found in the following 15 seconds. As a result network $N3$ is tried, where finally neighbors are found, so the device stays on $N3$ until the discovery cycle is terminated at $T_k + 60s$.
2. At T_{k+1}, the discovery is started again on network $N1$. Again, no neighbors could be found, but this time also no other networks are available (i.e., the device was

moved). The policy allows to switch to access point mode, which is what the `BeaconingManager` does at $T_{k+1} + 15s$. It stays in this mode, but because no neighbors have connected, it turns off access point mode at $T_{k+1} + 45s$. Since there are still no other networks around, the discovery cycle is terminated early after 45 seconds, causing the platform to go back to sleep.

Note that when using the previous rules there is a possible race condition that all devices simultaneously turn on access point mode, leading to no device discovering any neighbor. OppNet tries to circumvent such situations by only turning on access point mode with a certain probability.

3.2.2. Bluetooth Discovery

If using Bluetooth is enabled, the `BeaconingManager` performs a full scan (enabling the Bluetooth adapter first if necessary). This takes around 12-15 seconds and finds all discoverable devices nearby. After finishing the scan the `BeaconingManager` connects to each discovered device one after another and checks if it is also running OppNet. If so, both devices exchange their discovery beacons, and the connection is closed. After all discovered devices have been contacted the Bluetooth discovery is terminated. This whole process rarely takes more than 30 seconds.

3.3. Evaluation of Power Usage

Goal 1 from Section 1.1 requires OppNet to operate as energy-efficient as possible, while respecting the environment of a user. But instead of turning on (or off) single features, a typical end-user should only need to decide on a rough energy budget for the platform. The discovery features presented in the previous section should therefore be combined into reasonable policies. The user could then chose one of them based on the alleged energy use of each policy. One main factor in composing the policies is the power consumption of each single feature. This section quantifies each feature's power usage related to the different duty-cycles.

3.3.1. Power Monitoring Setup

The energy consumption is measured with the help of a power monitoring device [17] as shown in Figure 3.3. The *power monitor* (PM) bypasses the internal battery of the *device under test* (DUT). As a result, the PM supplies the DUT with a constant voltage (adjustable between 2.1 V and 4.5 V), delivering the DUT with a current of at most 4.5 A. This current flow is recorded up to 5000 times per second, and the measured data points (minimum and maximum currents) are sent to the *power monitoring tool* (PMT), a software running on a seperate computer. The software is used to control the PM's

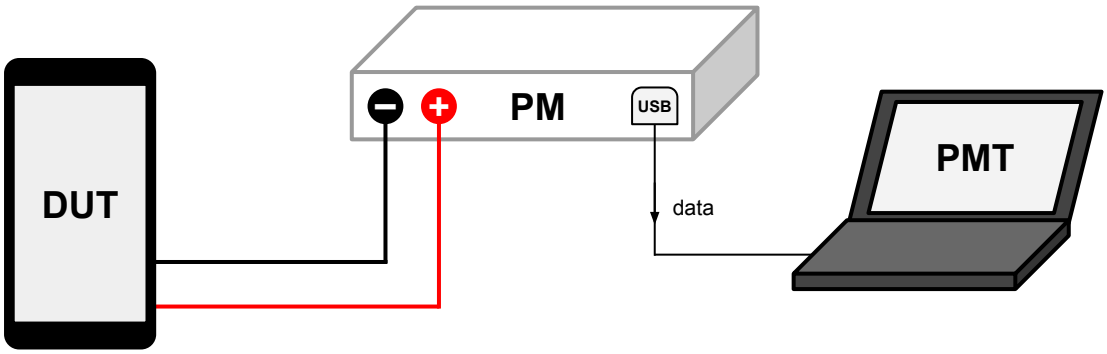


Figure 3.3.: Setup for the power measurement experiments

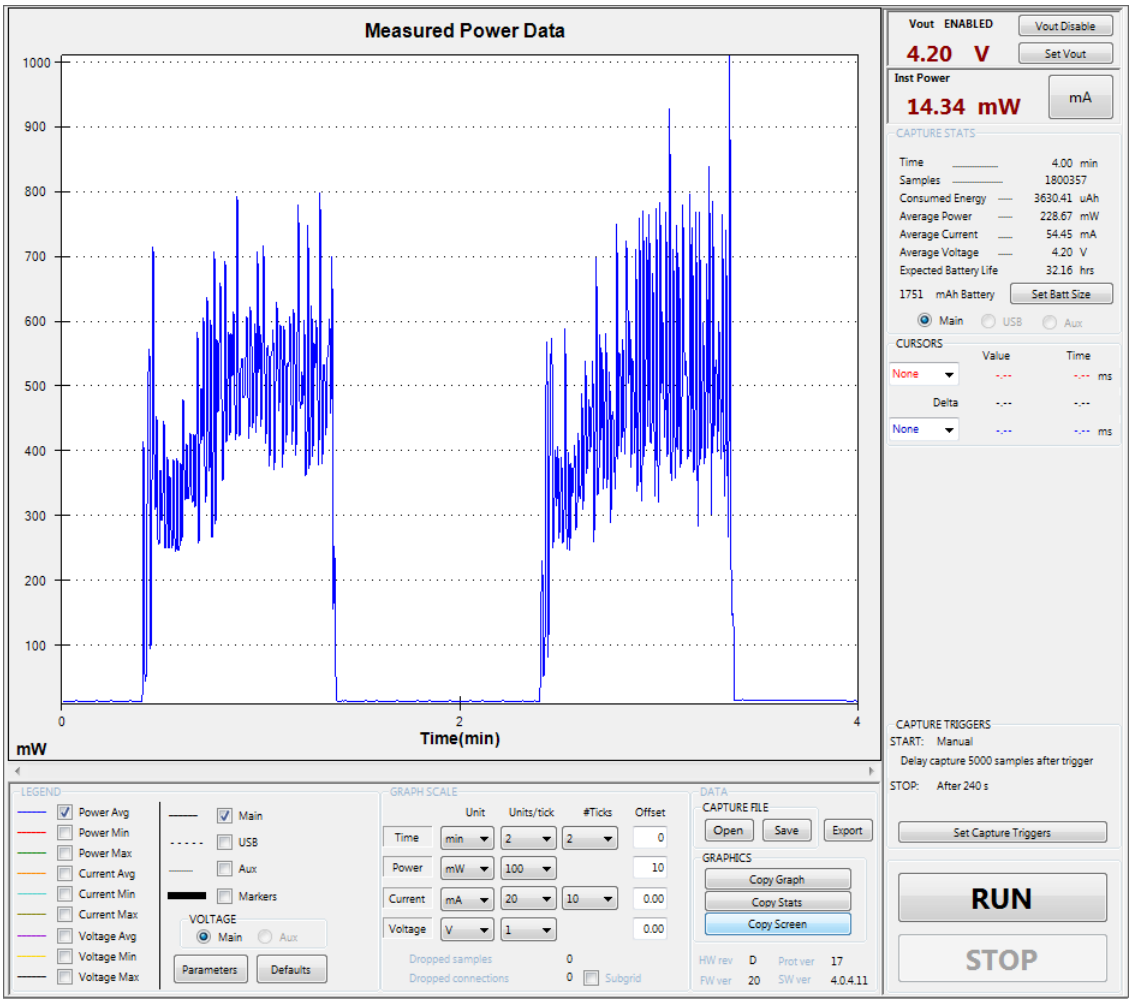


Figure 3.4.: Screenshot of the power monitoring tool

operation as well as to graph either the measured current or the electric power, derived from $P = I_{measured} * U_{const}$. Figure 3.4 shows a screenshot of the reporting software. The graph exemplarily illustrates two periods of high current demand (i.e., phases of activity by the device) seperated by periods of low current flow (i.e., sleeping phases).

3.3.2. Methodology

All measurements in this thesis were performed with the PM outputting a constant voltage of $U_{const} = 4.2\,\mathrm{V}$ (representing a fully charged smartphone battery). Since the PM measures the consumption of the whole DUT, the following steps have been taken to minimize the influence of the operating system (or other apps) on the measurement:

- The DUT is always a Samsung Galaxy Nexus running on a factory-reset Android 4.3. It contains no SIM card and is in flight mode.
- None of the pre-installed apps are running (except system apps), and no additional apps are installed.
- If possible, the OppNet platform is freshly installed on the DUT before every test run. The only exception is when a test run needs some preconditions to be met (e.g., when the device should already know some neighbors).
- In any case the platform is completely restarted for each test run (using the "Force Stop" feature in the apps settings of Android). After restarting, the system is given at least one minute to settle before a test run is started.
- After setting up the platform for the next test run (i.e., activating the proper feature), any debugging connection is torn down and the display is switched off. The test run is not started until the platform is back into its sleeping state (this can only be indirectly observed by looking at the live output of the PM).

A test run covers one full duty cycle, and at least four test runs are performed per test scenario. Some test scenarios are additionally run with two neighbor devices. These neighbor devices are set up according to the same rules as the DUT. To avoid a bias towards a certain configuration of neighbors, they are randomly selected from the pool of available devices (see Table 1.1) for every test run.

The average power consumption P_{avg} for a scenario is calculated through the average electric current $I_{avg,cycle}$ consumed per run, which in turn is the arithmetic mean of all measured currents $I_{measured}$ from n runs:

$$P_{avg} = U_{const} * I_{avg,cycle} = U_{const} * \left(\frac{1}{n} \sum I_{measured} \right)$$

Some scenarios are also evaluated in terms of *battery rundown time* $t_{rundown}$, based on the battery in the DUT, which contains $Q_{battery} = 1751\,\mathrm{mAh}$ of electric charge. It

can be calculated from the average electric charge consumed per cycle, which is derived from the average electric current $I_{avg,cycle}$ during a cycle t_{cycle}:

$$Q_{avg,cycle} = I_{avg,cycle} * t_{cycle}$$

$$t_{rundown} = \frac{Q_{battery}}{Q_{avg,cycle}} * t_{cycle} = \frac{Q_{battery}}{I_{avg,cycle}}$$

The battery rundown time is an upper limit which most likely will not be reached in real-world experiments – the OppNet platform is never the only consumer. However, it does allow for a better qualitative comparison between the scenarios.

3.3.3. Test Scenarios

The test scenarios can be grouped into three categories:

1. Each of the *reference scenarios* turn on a single basic Android function. The results are used to compare against the energy savings of the OppNet platform using duty-cycling.
2. The *feature scenarios* test the single power saving features (Bluetooth, WLAN client mode, WLAN access point mode) in isolation.
3. From the results of the feature scenarios five *policies* are formed, which are then evaluated.

The feature scenarios are measured for all three wakeup intervals $t_{cycle} \in \{2, 5, 10\}$ (see Section 3.1). They are also measured both with and without neighbors, as are the policy scenarios. All the different test scenarios will be described in more detail along with the measurement results in the next section. The raw data can be found in Appendix A.

3.4. Results

This section showcases the results for the different scenarios. Table 3.1 contains the results of the reference measurements, which help assess the measurements for the isolated features. From the results in Table 3.1 one can see that keeping Bluetooth (and WLAN to some extent) enabled all the time does not add much to the average power consumption. Note that the scenarios only measure the wireless radio being turned on constantly, without performing any scans. As expected, an always-on WLAN access point will drain the battery very fast. The values for the *Idle* and *WakeLock* scenarios will be plotted as horizontal lines in most graphs throughout this chapter.

Note that the experimental setup always leaks a constant amount of current, which can be seen by turning off the DUT: The PM will still measure a tiny amount of leakage current. It has been subtracted from all measurements done for this thesis.

3.4.1. Isolated Features

The following subsections present results for each of the three discovery features.

Bluetooth

Figure 3.5 shows the results for the feature scenarios with Bluetooth. In the first scenario on the left the platform is actively searching for neighbors via Bluetooth, but without any neighbors around – they are added in the second scenario (middle), resulting in a higher power consumption. This is expected, as connecting to neighbors means staying awake longer and involves sending actual data packets over the air.

In the third scenario (on the right) Bluetooth is still turned on, but the platform does not itself search neighbors and instead waits to be discovered by other neighbors. This saves even more energy than in the first scenario as no scan is performed. However, this is not a sustainable strategy in itself due to fairness reasons.

WLAN Client Mode

As described in Section 3.2, the WLAN discovery process is highly dependent on the environment: The amount of open networks, the order they are visited and the number of neighbors can result in completeley different discovery periods. Thus, there are a few more scenarios to analyze for WLAN client mode.

Figure 3.6 shows scenarios without neighbors present. In all scenarios the platform is active, but the number of available networks increases from left to right. The scenario on the left side only has one network available to connect to, which means it stays awake

Scenario	**Power**
	mW
Android Idle	12.50
Bluetooth	13.15
WLAN	14.78
WakeLock	41.38
AccessPoint	225.48

Table 3.1.: Average power consumption of reference scenarios

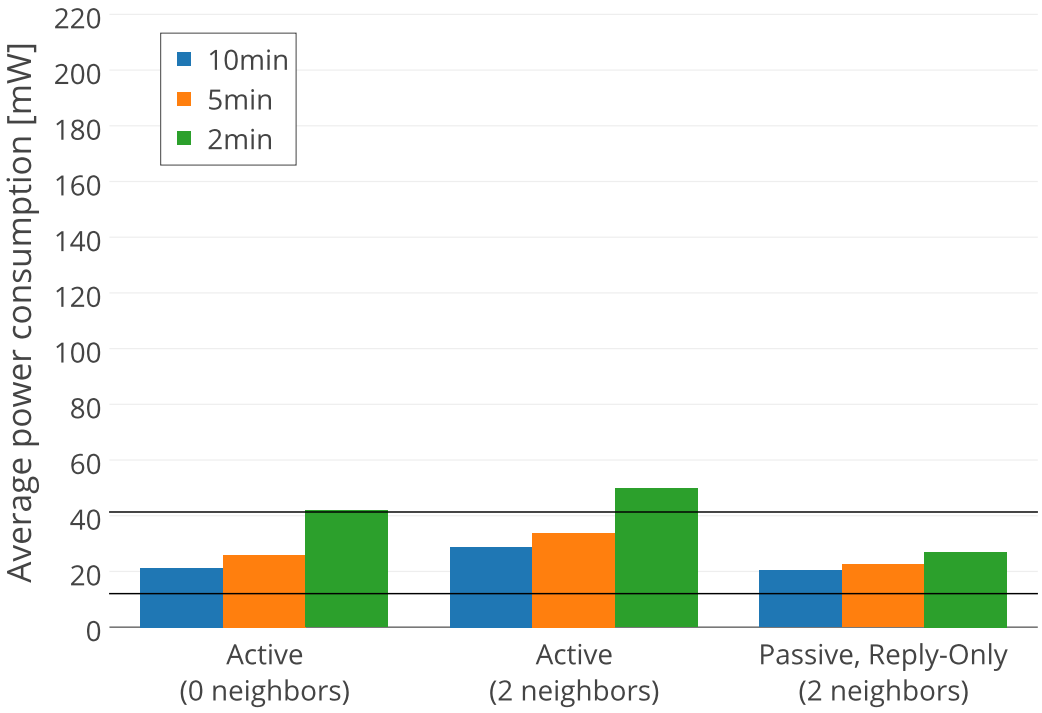


Figure 3.5.: Results for the feature scenarios with Bluetooth

		10 min	*5 min*	*2 min*
Active	P [mW]	21.26	25.93	41.78
(0 neighbors)	σ [mW]	0.68	1.69	0.80
	Runs	4	4	8
Active	P [mW]	28.79	33.77	49.87
(2 neighbors)	σ [mW]	1.97	0.84	1.35
	Runs	4	4	4
Passive	P [mW]	20.28	22.40	26.95
(2 neighbors)	σ [mW]	1.60	1.15	2.39
	Runs	6	4	7

Table 3.2.: Average power consumption of feature scenarios with Bluetooth

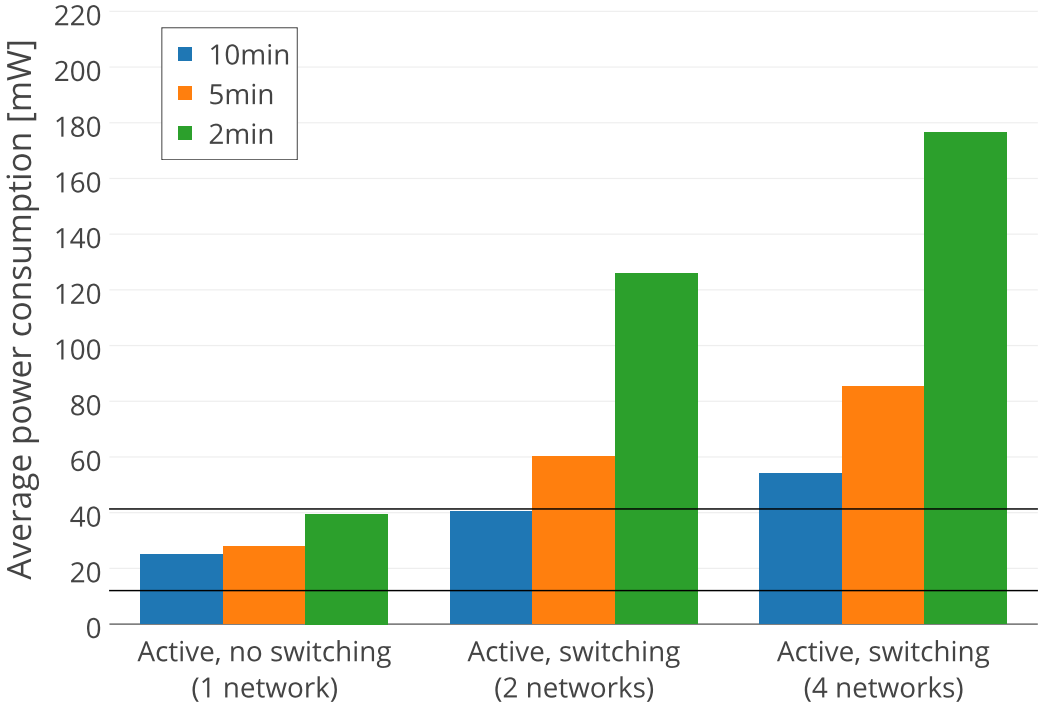


Figure 3.6.: Results for the feature scenarios with WLAN (without neighbors)

		10 min	*5 min*	*2 min*
Active,	P [mW]	24.97	27.88	39.56
no switching	σ [mW]	1.28	0.54	1.94
(1 network)	Runs	4	8	8
Active,	P [mW]	40.46	60.18	125.02
switching	σ [mW]	1.37	1.70	1.13
(2 networks)	Runs	4	4	10
Active,	P [mW]	54.20	85.20	176.56
switching	σ [mW]	0.40	1.02	0.85
(4 networks)	Runs	4	4	6

Table 3.3.: Average power consumption of feature scenarios with WLAN (without neighbors)

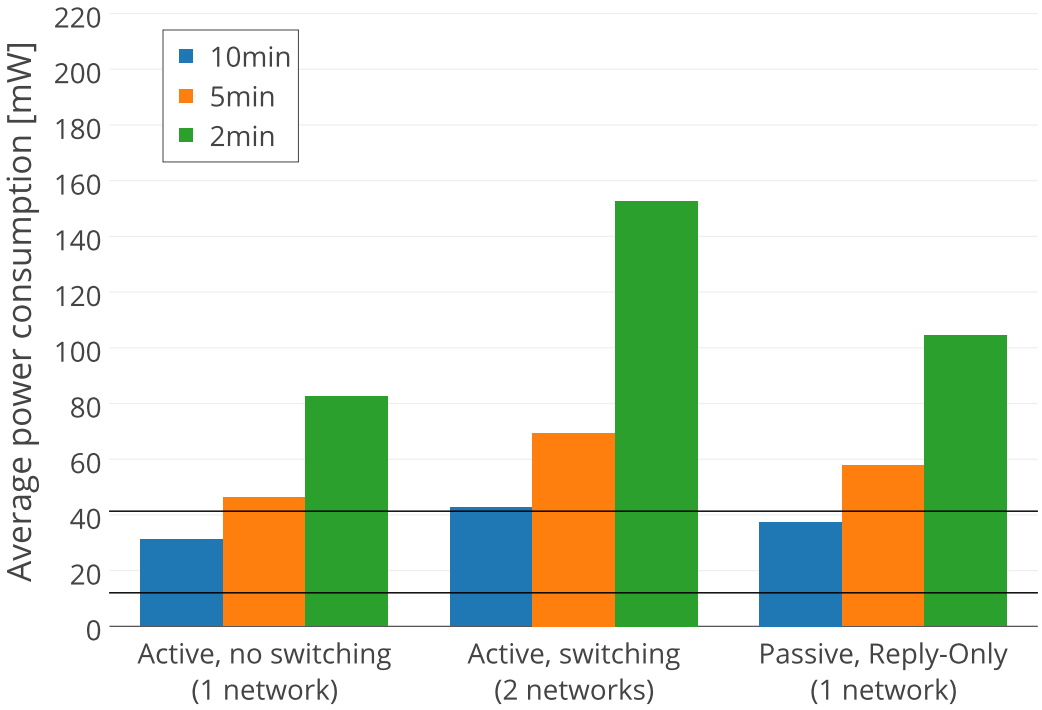


Figure 3.7.: Results for the feature scenarios with WLAN (two neighbors)

		10 min	*5 min*	*2 min*
Active,	P [mW]	31.11	46.25	82.58
no switching	σ [mW]	1.66	1.00	1.89
(1 network)	Runs	8	4	8
Active,	P [mW]	42.65	69.38	152.68
switching	σ [mW]	2.01	5.64	3.86
(2 networks)	Runs	4	4	4
Passive,	P [mW]	37.42	57.69	104.46
Reply-Only	σ [mW]	1.10	5.35	6.87
(1 network)	Runs	4	8	7

Table 3.4.: Average power consumption of feature scenarios with WLAN (two neighbors)

only for one slot (15 seconds). With two networks there will be a 30 second activity period. Energy-wise, the scenario on the right is the worst case: The platform will try to discover neighbors on all networks, staying awake for the full discovery cycle.

The graph confirms that WLAN discovery is more expensive than Bluetooth discovery. On the other hand, using WLAN client mode with a 10 minute wakeup interval still seems to be a good choice: Even in the worst case the energy consumption is only marginally higher than keeping the CPU awake all the time (the WakeLock reference scenario). Using the minimal interval of 2 minutes reveals that Android has difficulties to go to sleep in the one minute period OppNet is sleeping itself.

A second batch of scenarios is shown in Figure 3.7, this time always with two neighbors. The first two scenarios are basically the same as in the no-neighbor case. But when there are neighbors involved, switching through four networks is not possible (as the platform will stay on one network if it found any neighbors there). Therefore the third scenario with neighbors is similar to the Bluetooth case: The platform is only passively listening for discovery beacons, staying on the same network all the time. There is a small difference between this scenario and the first one with active searching and only one network as well: If the platform is passive, it replies to each and every beacon it receives. If it is active, it never directly replies, but rather sends out regular beacons every few seconds. With two neighbors, the device in passive mode will produce twice as many beacons as the one in active mode. This explains the increased power consumption of the rightmost scenario compared with the leftmost one.

All in all, using WLAN client mode in 10 minute intervals still looks promising when neighbors are involved.

WLAN Access Point Mode

Using the access point (AP) mode alone does not make a lot of sense: If all devices are only in access point mode, nobody would find anyone else. Nonetheless it is useful to measure some basic scenarios for AP mode itself: If nothing else, then at least the data can give clues when to add it to a policy (and what impact it would have) or when not.

In Figure 3.8, the scenario on the left shows an OppNet device without neighbors turning into an access point node every 2, 5 and 10 minutes, respectively. The energy consumption is more reasonable as one might think, but having no neighbors also means no traffic to handle. In this case the AP mode is turned off rather quickly. The scenario on the right adds two neighbors which connect to the tested device, so it now has to handle traffic. This scenario results in a power consumption similar to the WLAN client mode scenario without neighbors, but four available networks.

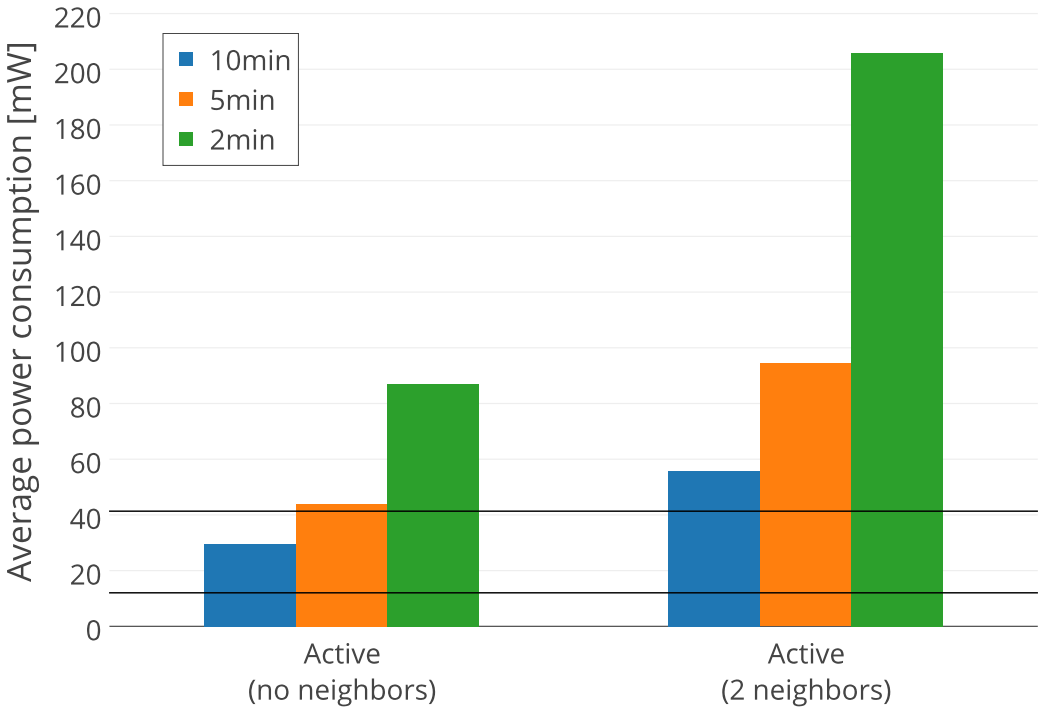


Figure 3.8.: Results for the WLAN access point mode

		10 min	*5 min*	*2 min*
Active	P [mW]	29.52	43.85	87.00
(no neighbors)	σ [mW]	1.40	1.43	1.16
	Runs	6	6	8
Active	P [mW]	55.66	94.34	205.82
(2 neighbors)	σ [mW]	0.53	1.42	2.64
	Runs	4	4	6

Table 3.5.: Average power consumption of feature scenarios with WLAN access point mode

3.4.2. Policies

Figure 3.9 compares the results of the single feature scenarios to each other. Some combinations look promising to form an energy-efficient policy which can then be used in the final OppNet platform:

- Bluetooth always seems to be a good choice. There is only one caveat with using Bluetooth in general: Since it only makes sense when the device is *discoverable*, OppNet must ask for the user's permission to activate this. Smartphone users are rather reluctant to keep Bluetooth activated all the time, so there should be an equivalent alternative to policies using Bluetooth.
- WLAN client mode in 10 minute intervals is the next best choice.
- Concerning WLAN access point mode, there is no clear answer. Long intervals seem to be quite efficient, but using client mode alone is likely still more efficient.

From these observations the five policies as seen in Table 3.6 have been formed. *Passive* is a baseline policy which does not actively scan for neighbors, but replies when detecting neighbors. It takes advantage of the fact that users sometimes keep their WLAN or Bluetooth connected when they do not use their phone (e.g., in the office).

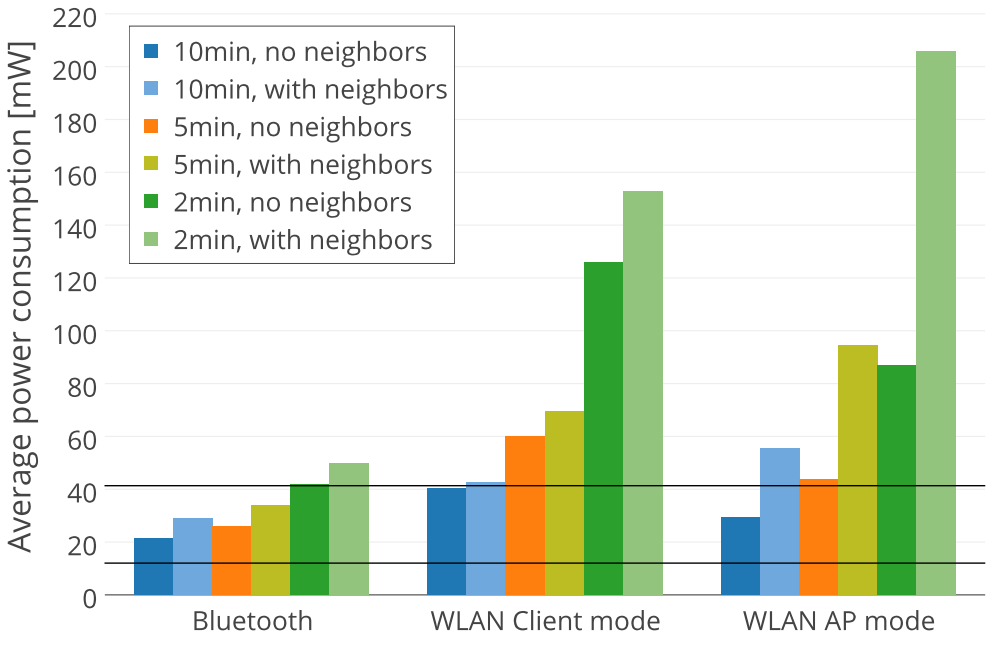


Figure 3.9.: Comparison of results for isolated feature measurements

	t_{wakeup}	WLAN Client mode	WLAN AP mode	Bluetooth
Passive	10 min	?		?
LowPower	10 min	x		
LowPower+	10 min	x		x
HighConnectivity	5 min	x	x	
Disaster	2 min	x	x	x

Table 3.6.: Policies

The *LowPower* policy actively uses WLAN client mode to find neighbors, providing a fallback for users who do not like to keep Bluetooth on (which is the only addition in the *LowPower+* policy). The *Disaster* policy tries to find as many neighbors as soon as possible and is intended for emergency situations – power efficiency is not the main objective. Finally, the *HighConnectivity* policy occupies the spot between LowPower and Disaster.

Ultimately, the power consumption of the resulting policies was measured, each with and without neighbors. The results are shown in Figure 3.10, with Table 3.7 containing the corresponding battery rundown times. The Passive, LowPower and LowPower+ policies deliver on their promise and save enough energy to consider running them constantly in the background. The expectation in real-world scenarios is that the power consumption is even less than what has been measured here: The user will be interacting with his device every now and then – if the user is using the phone, the platform will only perform discovery steps which do not disturb the user (e.g., not switching away to other WLAN networks) and therefore conserve even more energy.

In the end, the HighConnectivity policy increases the probability of connecting to neighbors by halving the duty cycle period to five minutes. This however comes at the cost of increased energy consumption. The Disaster policy primarily aims for other goals than energy efficiency. They are both not suitable to be used as always-on policies unless the device is externally powered.

Policy	**Battery Rundown Time [h]**	
	0 neighbors	2 neighbors
Passive	501.09	190.16
LowPower	213.92	148.72
LowPower+	202.86	138.20
HighConnectivity	107.90	97.58
Disaster	49.18	34.27

Table 3.7.: Battery rundown times for OppNet policies

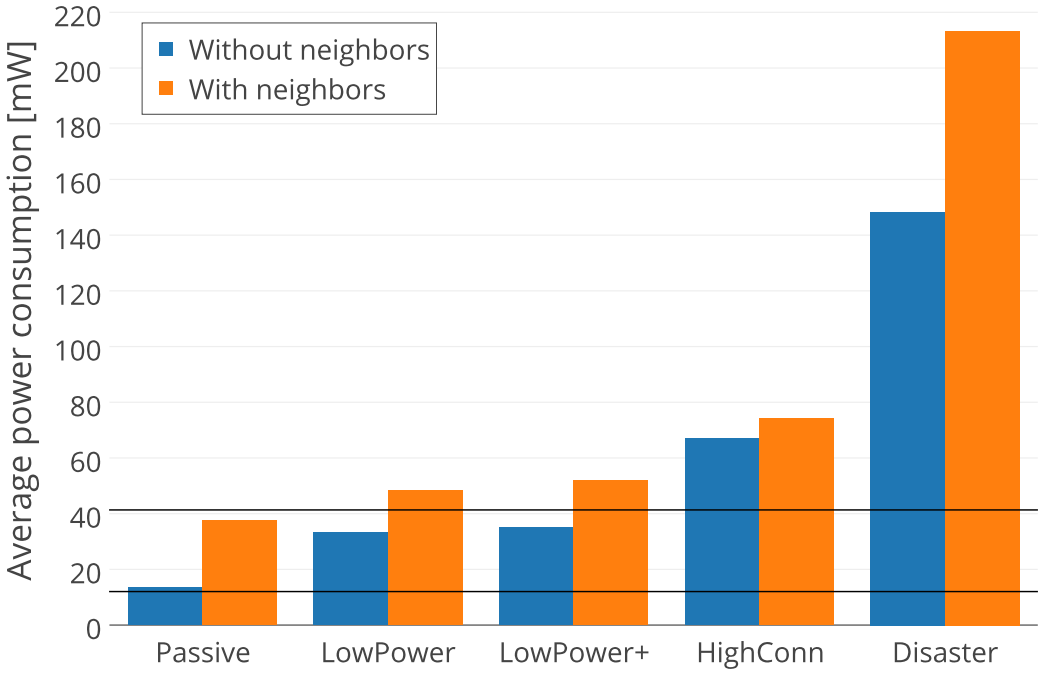


Figure 3.10.: Results for the final policies

		P	σ	BRT	Runs
		mW	mW	h	
Passive	*0 neighbors*	13.43	0.07	501.09	4
	2 neighbors	37.42	1.10	190.16	4
LowPower	*0 neighbors*	33.13	6.53	213.92	4
	2 neighbors	48.20	1.63	148.72	6
LowPower+	*0 neighbors*	35.00	0.47	202.86	4
	2 neighbors	51.96	2.66	138.20	4
HighConnectivity	*0 neighbors*	66.91	6.59	107.90	4
	2 neighbors	74.11	1.86	97.58	5
Disaster	*0 neighbors*	148.30	3.94	49.18	5
	2 neighbors	213.32	4.05	34.27	4

Table 3.8.: Average power consumption of policies

4. Service Platform

The second significant contribution of this thesis is the introduction of a complete service layer on top of the existing discovery layer. It offers a simple but powerful way for app developers to build rich apps utilizing the opportunistic communication principle. The OppNet service layer is built on three essential blocks:

1. *Service Announcement*: The component where apps can publish the services they support and subscribe for updates.
2. *Service Discovery*: The mechanism employed by the platform to get to know what neighbor understands which protocol.
3. *Data Exchange*: The component which actually transmits service-related data between neighbors.

Easing developer life is the main motivation behind the integration of the service layer into the platform. Using it is completely optional, but it offers quite a few benefits: The platform automatically handles all the communication between neighbors, so the third-party app does not need to care about network sockets. This is especially handy when neighbors suddenly disappear and later reappear. The platform also already knows best how to communicate with a certain neighbor, and abstracts away the wireless technology used to do so. The final bonus is that the platform can transparently encrypt and authenticate data during exchange.

Easy access to the service layer is provided to third-party apps through a small Android library, which is covered in detail in Section 5.1. The rest of this chapter explains how each part of the service layer works.

4.1. Client Apps

One of the limitations described in Section 2.3 was the missing concept of client apps. In OppNet, a client app must supply a receiver for an OppNet-specific broadcast message, and expose it to other applications, to be considered compatible by the platform.

Client apps are automatically discovered by the platform. Figure 4.1 shows the process when the platform has already been installed prior to the installation of the client app: The platform is notified by Android when a new app is installed. The *Client App Registration Service* then checks if the new app fulfills the contract of an OppNet

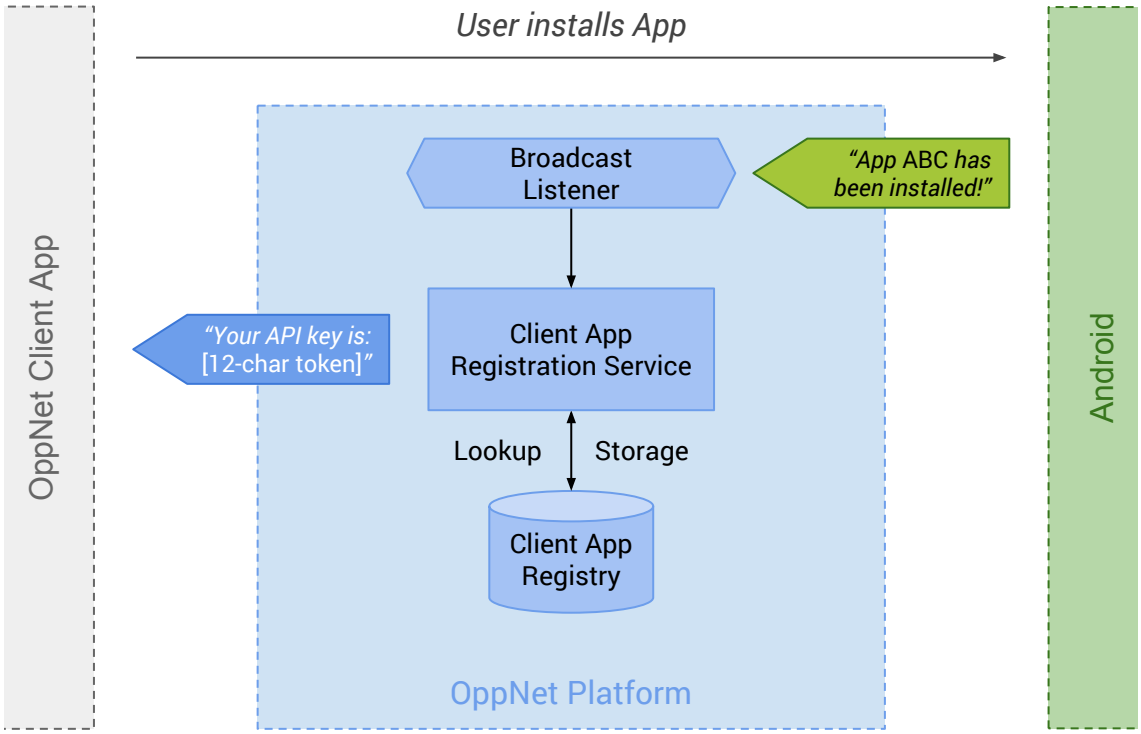


Figure 4.1.: Registering an OppNet client app

client app. This is done by querying the Android package manager for the new app and inspecting the exported broadcast receivers. If there is such a receiver, a new *API key* is sent. The API key must be stored by the client app and is required whenever communicating with the platform later. The platform keeps track of the mapping from issued API keys to client apps.

To issue such API keys even to client apps which have been installed prior to the platform, there is a similar process: After installation the platform queries the apps installed on the device and issues API keys to those who appear to be OppNet client apps, again by inspecting the exported broadcast receivers.

4.2. Service Announcement

As soon as a client app possesses an API key, it can publish its services to the platform. To do so the client app hands over *protocol descriptions* to the platform. A protocol describes a service and how its data should be handled by the platform. The only required field in the description is a unique name – service discovery will be based on that name. Additional fields are interpreted as metadata about the service: Whether data should be encrypted and/or authenticated when being exchanged, or an expiry time

(in seconds) after which data is considered outdated and not exchanged anymore. An example protocol description in XML could look like the following:

```
<protocol>
    <!-- required, should use a unique namespace -->
    <name>ch.ethz.csg.oppnet.example.chat</name>

    <!-- defaults to 'false' if omitted -->
    <encrypted>false</encrypted>

    <!-- defaults to 'false' if omitted -->
    <authenticated>true</authenticated>

    <!-- no default value -->
    <defaultTTL>3600</defaultTTL>
</protocol>
```

Figure 4.2 illustrates how a client app typically publishes services with the platform. After the app is started and connected to the *Client Connector Service* of the platform

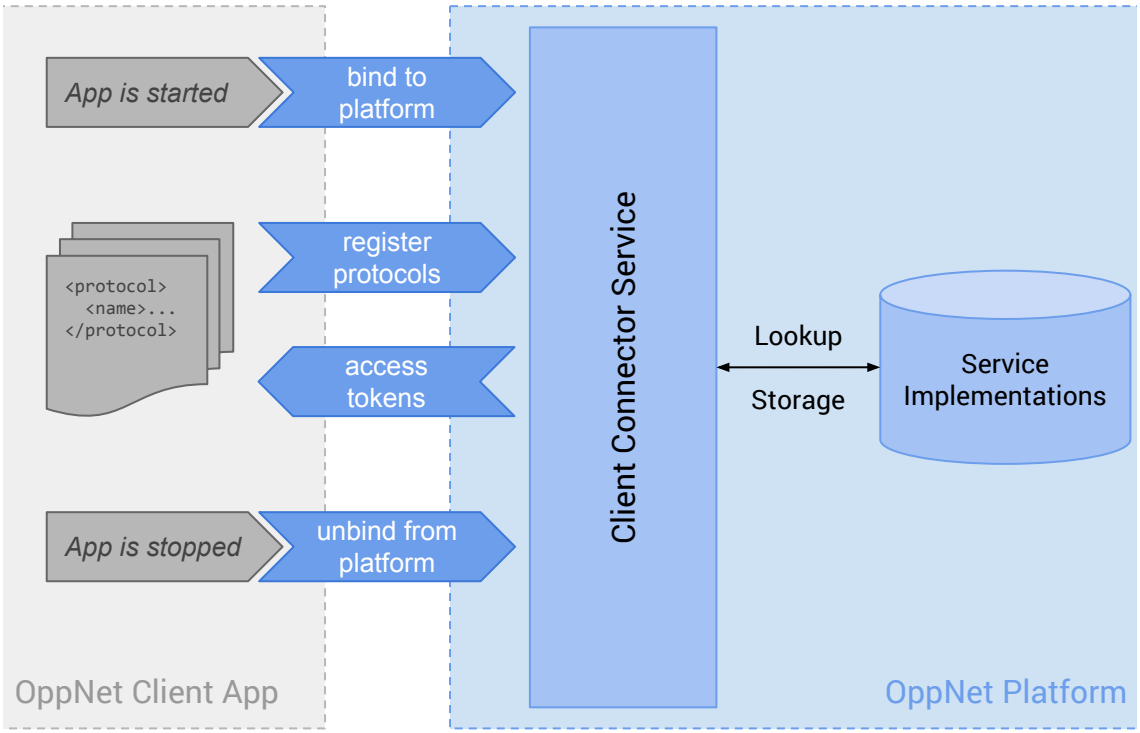


Figure 4.2.: Publishing services to the OppNet platform

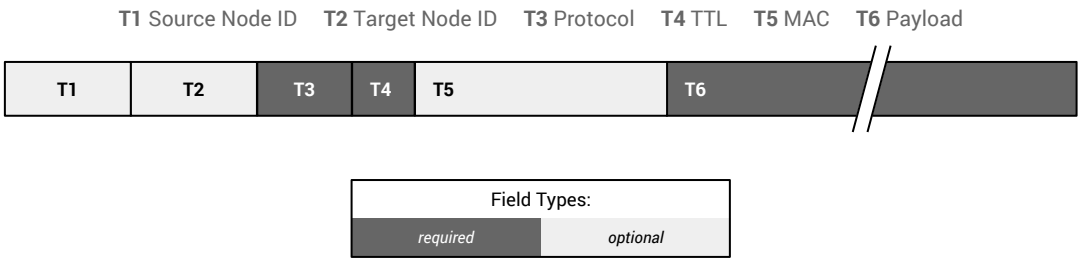


Figure 4.3.: Wire format of the TransportData packet

it can start registering protocols. OppNet records each combination of client app and protocol (internally called *service implementation*) with a unique *access token*. This token is sent back to the client app, and is required later to publish data and subscribe to data updates. Client apps can register as many services as they want.

4.3. Service Discovery

After the platform learned about apps and services present on the same device, it needs a way to learn about services on other devices. OppNet currently uses a very simple approach: It directly combines service discovery with neighbor discovery. To keep the *discovery beacon* small in size, only SHA-1 [18] hashes of the protocol names are transmitted – this only adds 20 bytes per supported protocol.

The service layer always keeps a registry of currently connected neighbors with their announced services in memory. Client apps can subscribe to this registry to be notified when a new neighbor for a specific protocol connected (or disconnected).

4.4. Data Exchange

The service layer only has one requirement when it comes to how client apps must deliver data to be exchanged between devices: It must be a serialized binary blob of data. How the serialization is done is completely up to the client app – the platform does not need to know anything about the data structure. To actually transmit this raw data to other devices it is wrapped by the `TransportData` packet shown in Figure 4.3. The platform has an *incoming*, *outgoing* and *forwarding* queue where it stores data as complete `TransportData` packets (they can be in more than one queue simultaneously). The wrapper packet adds the following metadata:

T1: The ID of the node the data originates from. This is not necessarily the ID of the node the data was created. The field is only set when the data transmission is authenticated.

T2: The ID of the node the data is targeted at, if the data has a final destination.

T3: The SHA-1 hash of the protocol the data belongs to.

T4: The time until when the data is valid.

T5: A message authentication code, when the protocol requires data transmission to be authenticated.

When receiving `TransportData` packets, malformed and outdated packets are directly discarded. Other reasons to discard packets are *a*) authenticated packets where verification fails; *b*) encrypted packets which can not be decrypted; and *c*) when no client app is registered with the platform for the protocol of a targeted packet.

Finally, the decision of what data is sent to which neighbor is the duty of the `DataExchangeManager`. The process gets triggered in two situations:

1. When a client app pushes new data to be exchanged and neighbors are currently connected. In this case the `DataExchangeManager` checks whether any neighbor supports the data's protocol. If so, the data is wrapped and sent.
2. The platform discovered a new neighbor. The `DataExchangeManager` then inspects the new neighbor's supported protocols and checks the outgoing queue if it contains matching data. It also checks the forwarding queue for suitable packets.

The `DataExchangeManager` transports data over UDP/IP, operating on port 3109.

4.5. Security

Together with the service layer, the OppNet platform got equipped with a new *identity* mechanism. What has been randomly chosen 8 bytes is now a cryptographically usable public key. It is based on elliptic curve cryptography (ECC), which has one major advantage over the classic RSA-based cryptography: A 256-bit (32-byte) ECC public key provides the same level of security as a 3072-bit (384-byte) RSA public key. The small size of such a public key allows to include it as the node identity into each discovery beacon.

The underlying cryptosystem is powered by libsodium[1], a portable version of Dan Bernstein's original NaCl library [10]. OppNet uses libsodium's `cryptobox` for authenticated public-key encryption, which is built around the Curve25519[2] elliptic curve. Signatures are created with Ed25519[3], which uses the same elliptic curve. This enables OppNet to use a single public key for both operations (encryption and signatures).

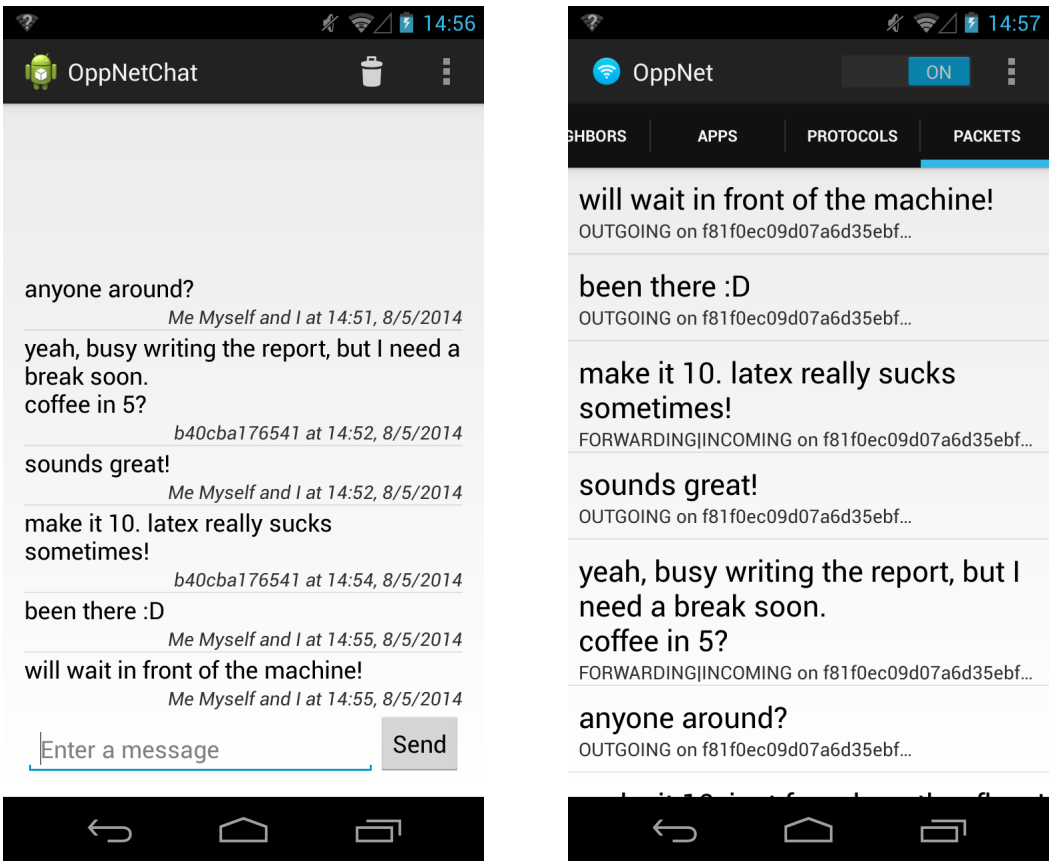
[1] `http://doc.libsodium.org/`
[2] `http://cr.yp.to/ecdh.html`
[3] `http://ed25519.cr.yp.to/`

4.6. Proof of Concept Client App

To demonstrate how to use the new service layer a proof of concept app has been built. *OppNetChat* is a simple chat app which broadcasts messages to neighboring devices. There is only one single chat room with no moderation. The app does not include an own identity layer, but reuses the one from OppNet: Usernames are the hexadecimal representation of the sender's public key (i.e., the node ID).

OppNetChat registers one protocol with the platform, which asks for authenticated but unencrypted data exchange. To serialize the actual message it is treated as a sequence of bytes (instead of characters). Deserializing converts this byte string back into a real string.

Figure 4.4 shows OppNetChat in action: The user interface (screenshot on the left) displays received messages and allows the user to send a new message. In the screenshot on the right the list of generated packets to be sent by OppNet can be seen.



(a) OppNetChat screenshot

(b) Packets registered with OppNet

Figure 4.4.: Screenshots of example client app

5. OppNet Platform Interface

There are two main interfaces to interact with all the features of the platform: An Android library which allows developers to build apps on top of the platform, and a graphical user interface for the device owner running the platform. This chapter gives an overview over the capabilities of both interfaces, and additionally describes the structure of the discovery beacon. The latter is an external interface which allows clients on other platforms than Android to understand these (otherwise useless) packets.

5.1. Android Library

The OppNet library is a collection of convenience objects to use the platform from third-party client apps. Its first feature is the broadcast receiver required by OppNet to qualify as a client app (see Section 4.1), and a `TokenStore` which handles all the tokens involved in communicating with the platform. The latter only needs to be touched when the client app registers more than a single protocol with the platform.

The starting point for including the library into a client app is the `OppNetConnector`. It handles all inter-process communication with the platform (after explicitely binding to it). If the platform is not installed, the `bind` operation will return an error. In this case the client app could notify the user to download the platform to use all features of the app.

The library features a parser for the XML format of the service descriptions. If they are available as XML resources from the client app[1], the `OppNetConnector` has a shortcut to register them directly with the platform. Registering a protocol through the library also registers a callback for notifications when new data is available for the registered protocols.

The library additionally offers convenience functionality when not using the service layer: With a `NeighborObserver` a client app can get notified when new neighbors connect.

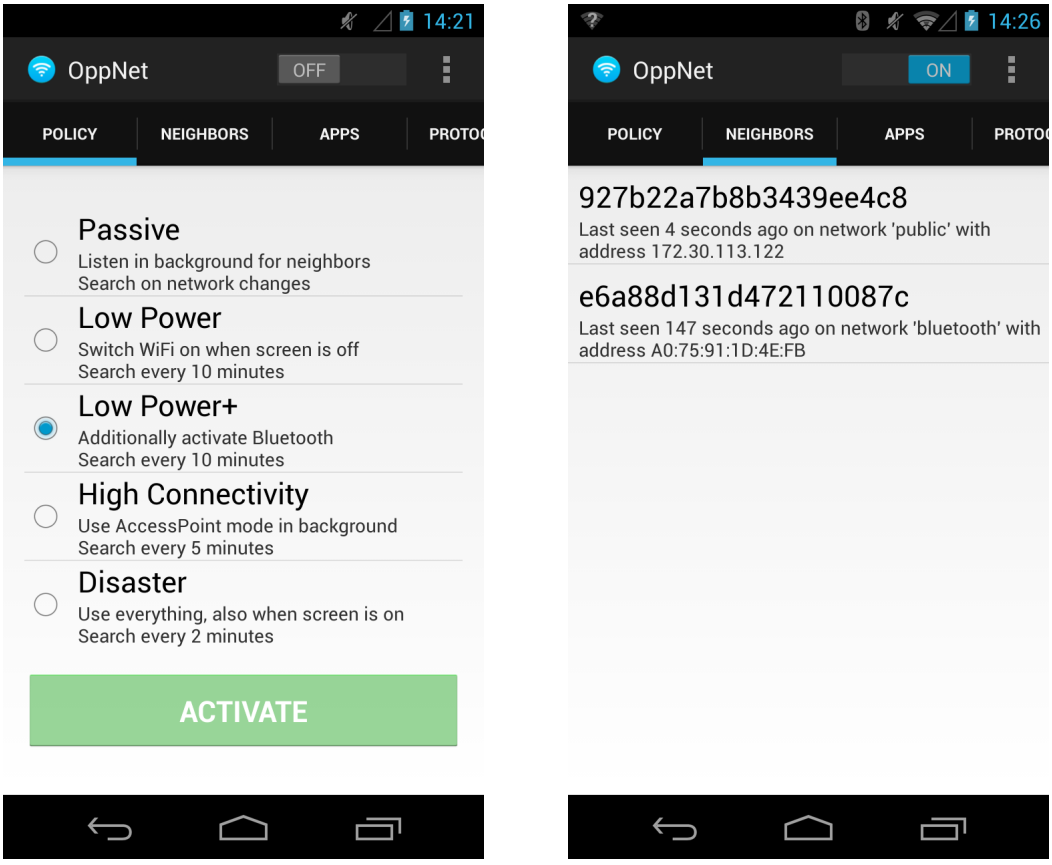
[1] https://developer.android.com/guide/topics/resources/providing-resources.html

5.2. Management GUI

Most of OppNet happens in the background. The only part visible to the user is the graphical management interface. OppNet's predecessor already had such an interface, but it was targeted at platform developers. In a complete overhaul the graphical user interface (GUI) now puts the regular end-user in control. Figure 5.1 shows screenshots of the two main screens:

(a) The start screen lets the user control the operation of the platform. It presents the list of available policies to chose from together with a short description of each policy, and a button to activate the platform. Policies can be switched on the fly by selecting a different one and hitting the green button again.

(b) Swiping to the right from the start screen reveals four (purely informational) list views: the currently connected neighbors (visible in the screenshot), all client apps



(a) Policy selection (b) Current neighbors

Figure 5.1.: Screenshots of OppNet management GUI

installed on the device, all protocols registered with the platform, and the most recent packets received or to be sent.

Located in the top right corner is a master switch which allows to start/stop the platform from any subview. The GUI provides the same clean, modern look even on older Android devices. Note that technically the GUI is completely optional: The platform also runs without it.

5.3. Discovery Beacon

The discovery beacon is OppNet's way of telling other devices about its presence. In IP-based networks it is sent to the multicast addresses `224.0.0.251` and `ff02::fb` on port 5353 over UDP. These are the same addresses as used by Multicast DNS, however OppNet beacons have a fundamentally different packet structure. In some cases discovery beacons are also sent as UDP unicasts to port 3108.

A discovery beacon carries information about the sending device, and potentially similar information about other neighbors it already knows. Figure 5.2 depicts the wire format of such a beacon. It is basically composed of five fields:

B1: The beacon type can be either "original" (default) or "reply". The implication is that upon receiving a beacon of type "original" an OppNet client should always send a beacon of type "reply" back, and never send anything back upon receiving a beacon of type "reply".

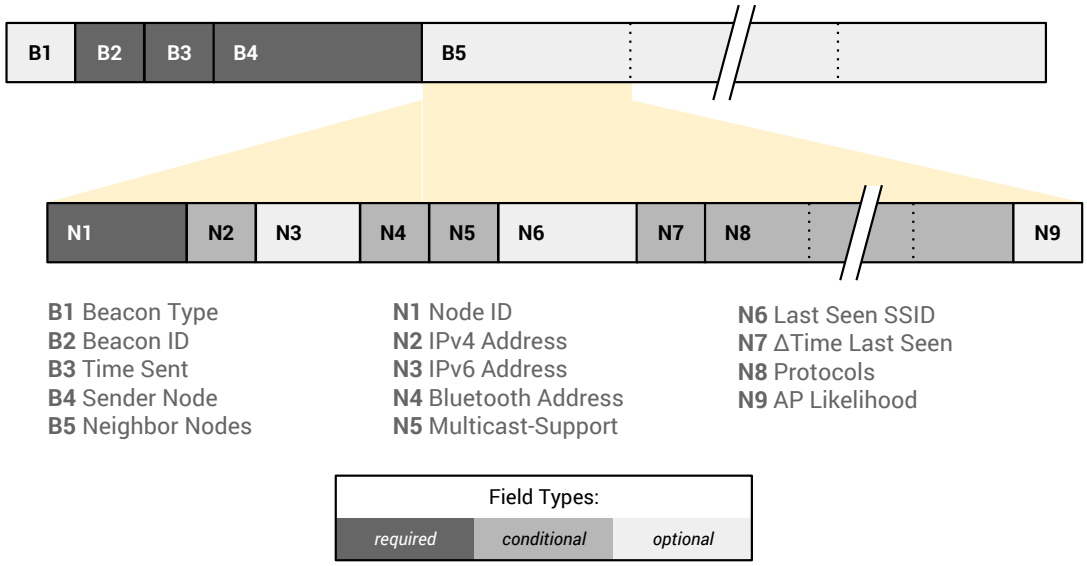


Figure 5.2.: Wire format of discovery beacon

B2: When receiving multiple beacons with the same beacon ID an OppNet client may consider them duplicates of the original one, and discard them.

B3: The time when the beacon has been sent, in seconds elapsed since 01.01.1970 00:00:00 UTC.

B4: Information about the sender of the beacon, in the form of a *Node* sub-message (explained below).

B5: This field contains *Node* sub-messages for each neighbor known to the sender of the beacon.

A Node message contains contact information and other metadata about a device. The presence of certain fields depend on what device the message describes (i.e., the sender of a beacon or its neighbors). All in all it can consist of up to nine fields:

N1: The public key as introduced in Section 4.5. This field must always be set.

N2: The IPv4 address to reach the described node.

N3: The IPv6 address to reach the described node. Only one IP address is required to be set.

N4: The Bluetooth address of the described node, if available.

N5: A boolean flag whether the described node is able to receive multicast messages when it is asleep (defaults to "true"). On some Android devices the networking chip drops multicast packets when in energy savings mode (i.e., the device is asleep).

N6: The name of the WLAN network on which the described node has been seen last. Only relevant when the node describes a neighbor, and only if it is on a different network than the beacon sender.

N7: The difference between the time the node has been seen last and the time the beacon has been sent. This field is only required when the node describes a neighbor.

N8: A list of SHA-1 hashes of the protocol names a node supports (see Section 4.3).

N9: If the nodes are communicating via an OppNet access point, this field signals the likelihood for a node to become an access point. The node with the highest likelihood is expected to take over the access point role in the next discovery cycle.

6. Conclusion

The goal of this thesis was to deliver a service platform for opportunistic networking under Android which is both energy-efficient, developer-friendly and backwards-compatible.

The first goal has been pursued by introducing *duty-cycling* to the WLAN-Opp discovery mechanism, giving devices the possibility to sleep and therefore saving a significant amount of energy. A second strategy was to allow for finer-grained control over the energy budget by selecting single *technology features* to be used in the discovery process. This also included adding Bluetooth to the technology pool. The different combinations of duty-cycles and technology features have then been evaluated. From the results five distinct *policies* have been proposed, and their final power consumption has been measured. The results prove that energy-efficient opportunistic discovery is indeed possible.

With the addition of a full-blown service layer developers now have a more powerful tool at hand to build apps leveraging opportunistic principles. With service announcement, automatic service discovery and opportunistic data exchange built into the platform, the developer has less to take care of. The service layer is complemented by strong elliptic curve cryptography, which developers can take advantage of on the data transport layer.

Great care has been taken to keep OppNet compatible with most Android devices: The minimum requirement is version 2.3.3 ("Gingerbread"). Finally OppNet will soon be available as open-source software.

6.1. Future Work

The OppNet platform already is a reasonable foundation for opportunistic applications, yet there is still room for improvement in some areas:

- The energy consumption can and should be further optimized, especially for WLAN discovery. A good starting point is the WLAN beaconing frequency, as producing and sending such beacons is the major energy consumer during discovery: Each beacon costs around 100 mJ, which is equivalent to waking up the CPU for 2.5 seconds.

- Policies were a big step forward, but they fail to automatically adjust to changing environments. It seems worth to investigate adaptive discovery techniques, for example by extending the sleep periods when a device does not move (and vice versa). Another step would be trying to detect patterns: When the user is at work, expect to meet specific neighbors and tune the discovery process accordingly.
- OppNet's Bluetooth support is still in its infancy. Bluetooth proved to provide low-energy discovery, and could save some more energy in passively listening devices. Furthermore it is only utilized in the discovery process yet.
- On the service layer a more sophisticated packet distribution algorithm than "flooding" can minimize the exchange of already transmitted data. Adding a pull-based mode would make exchanging of big files more efficient: Devices could announce the mere existence of such a file instead of directly pushing it towards the target. If the target is interested in the file it would initiate the download itself.
- The platform itself could be extended with a more comprehensive identity layer, adding concepts like *friends* or *social trust*. This would simplify building apps for OppNet even more, while also giving the platform a way to enforce privacy.

A. Energy Optimization Measurements: Raw Data

The following tables contain the average power consumption measured per run, grouped by feature and scenario as presented in Chapter 3.

Scenario	P_{10min} mW	P_{5min} mW	P_{2min} mW
Active (0 neighbors)	21.99	24.73	41.55
	20.53	25.15	40.16
	21.66	25.41	42.46
	20.86	28.44	41.93
			42.07
			41.47
			41.71
			42.86
Active (2 neighbors)	26.00	33.51	48.84
	30.41	34.95	49.52
	28.86	33.63	49.29
	29.89	32.99	51.85
Passive, Reply-Only (2 neighbors)	19.15	22.52	29.24
	20.06	23.20	24.36
	18.59	23.13	29.31
	19.48	20.73	28.81
	21.58		26.52
	22.80		27.09
			23.32

Table A.1.: Raw data per run for average power consumption of feature scenarios with Bluetooth

Scenario	P_{10min} mW	P_{5min} mW	P_{2min} mW
Active, no switching (0 neighbors, 1 network)	24.04	27.89	39.30
	25.69	27.04	38.82
	26.38	28.22	39.86
	23.74	27.61	39.18
		27.95	37.30
		27.38	38.09
		28.16	40.16
		28.76	43.80
Active, switching (0 neighbors, 2 networks)	38.81	59.41	124.56
	42.11	58.32	126.30
	40.76	60.73	125.86
	40.18	62.27	126.51
			126.03
			123.89
			124.58
			125.04
			123.12
			124.29
Active, switching (0 neighbors, 4 networks)	54.68	85.61	176.50
	53.81	83.74	176.10
	53.93	85.38	176.74
	54.36	86.08	175.24
			177.04
			177.75

Table A.2.: Raw data per run for average power consumption of feature scenarios with WLAN client mode, without neighbors

Scenario	P_{10min} mW	P_{5min} mW	P_{2min} mW
Active, no switching (2 neighbors, 1 network)	32.70	47.46	80.41
	31.39	45.09	80.63
	32.99	45.93	81.14
	33.23	46.53	81.36
	29.93		84.44
	29.68		83.60
	29.64		83.94
	29.34		85.13
Active, switching (2 neighbors, 2 networks)	40.53	61.58	150.27
	44.84	72.63	158.43
	43.80	69.06	151.29
	41.44	74.26	150.71
Passive, Reply-Only (2 neighbors, 1 network)	37.69	58.31	109.56
	35.81	59.19	112.43
	37.98	59.00	107.75
	38.21	60.43	96.74
		60.64	103.58
		61.23	93.80
		57.98	107.39
		44.76	

Table A.3.: Raw data per run for average power consumption of feature scenarios with WLAN client mode, with two neighbors

Scenario	P_{10min} mW	P_{5min} mW	P_{2min} mW
	31.04	43.94	88.06
	30.42	43.17	85.49
	28.51	42.64	86.11
Active, no switching	27.25	42.33	87.13
(2 neighbors, 1 network)	30.17	45.08	88.60
	29.71	45.96	85.76
			86.78
			88.06
	54.97	95.19	209.04
	55.94	95.78	202.25
Active, switching	56.19	93.74	204.08
(2 neighbors, 2 networks)	55.54	92.65	207.32
			204.33
			207.91

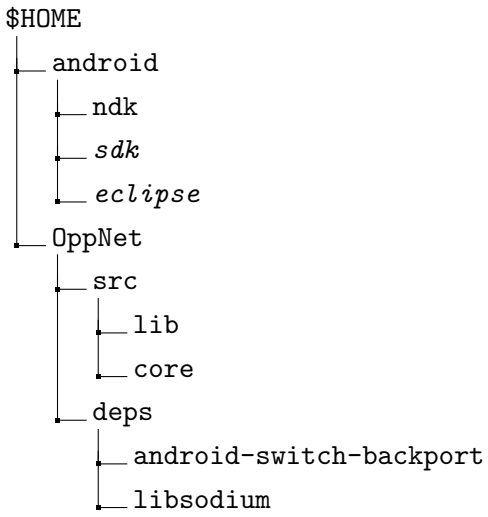
Table A.4.: Raw data per run for average power consumption of feature scenarios with WLAN access point mode

Policy	$P_{0neighbors}$ mW	$P_{2neighbors}$ mW
Passive	13.49	37.69
	13.48	35.81
	13.33	37.98
	13.40	38.21
LowPower	28.29	50.01
	33.50	47.97
	28.49	47.82
	42.23	46.99
		50.26
		46.16
LowPower+	35.13	54.13
	34.47	49.47
	35.57	54.39
	34.83	49.86
HighConnectivity	64.47	73.05
	70.53	76.75
	73.75	75.30
	58.89	73.26
		72.21
Disaster	154.20	207.55
	148.52	216.40
	149.09	213.49
	145.98	215.85
	143.69	

Table A.5.: Raw data per run for average power consumption of policies, without and with neighbors

B. Installation Guide

The following chapter contains the instructions to setup the platform for development with the official *Android Developer Tools* (ADT). Throughout the guide the following directory structure is assumed:



The `android` folder will contain the development tools for Android (the `sdk` and `eclipse` subfolders will be created while installing the ADT), while the `OppNet` folder contains everything else to build OppNet. Besides the OppNet source code in the `src` subfolder, the `deps` subfolder will contain the dependencies which are not directly included in the OppNet source distribution. Before moving on, put the source code for the OppNet library and the OppNet core into the `lib` and `core` directories (e.g., with `svn checkout`).

B.1. Eclipse/ADT Setup

To setup the Android Developer Tools, download the suitable "ADT Bundle" from the Android Developers website[1]. Unpack the retrieved ZIP file to the `$HOME/android` folder, which will create the `sdk` and `eclipse` subfolders. Start the development environment, Eclipse, with the executable located in the `$HOME/android/eclipse` directory.

[1] https://developer.android.com/sdk/index.html

Launch the SDK tools (from Eclipse via "Window" → "Android SDK Manager") and make sure that at least the SDK platform for Android 4.4 (API level 19) and the Android Support Library (under "Extras") are installed.

Both OppNet projects can easily be imported into Eclipse. Select "File" → "Import" and choose "Existing Android Code Into Workspace". Choose `$HOME/OppNet/src` as the "Root Directory" and select both `lib` and `src` to import. Your Eclipse view should now contain the two OppNet projects, with the `core` project containing compile errors because of missing dependencies.

B.2. Dependencies

This section lists the dependencies for the OppNet platform, and how to resolve them.

B.2.1. Library

The OppNet library only has one dependency:

Guava 16.0.1 Already included in the source distribution, no action needed.
https://code.google.com/p/guava-libraries/

B.2.2. Core

The OppNet core depends on the library, plus the following libraries:

Protobuf 2.5.0 Already included in the source distribution, no action needed.
https://code.google.com/p/protobuf/

Android Support Library v7 appcompat See below for instructions.
https://developer.android.com/tools/support-library/features.html#v7

Android Switch Widget Backport 1.3.1 See below for instructions.
https://github.com/BoD/android-switch-backport

To resolve the Support Library dependency, follow the guide on the Android Developers website to "add libraries with resources"[2]. The source to include is located in the `$HOME/android/sdk` directory.

The Android Switch Widget Backport dependeny can be resolved by downloading the source code (either with `git clone` or downloading and unpacking into the `$HOME/OppNet/deps/android-switch-backport` folder). Next, import the "library" project into Eclipse. You should now see two library projects along with the two OppNet projects in Eclipse, and the errors should have disappeared.

[2] https://developer.android.com/tools/support-library/setup.html#libs-with-res

B.3. Protobuf

When changing the Protocol Buffer message definitions in the *ch.ethz.csg.oppnet.protobuf* package of the OppNet core, they have to be recompiled. To do so, download and install the Protocol Buffer compiler from the website[3] or the package manager of your operating system. Take care to use the compiler for the same protobuf version as the included JAR (i.e., 2.5.0). To regenerate the Java classes from the message definitions, run the following command:

```
cd $HOME/OppNet/src/core
protoc --java_out=src/ \
    src/ch/ethz/csg/oppnet/protobuf/PacketDescriptions.proto
```

B.4. Native Code

The OppNet core contains C code for the cryptographic functions mentioned in Section 4.5. The source distribution already contains the compiled shared library, which is automatically detected by Eclipse/ADT and included in the compiled Android application package (APK).

If the C source ever needs to be updated, the shared library has to be recompiled. To do this, the *Native Development Kit* (NDK) is needed. Download the appropriate NDK build from the Android Developers website[4] and unpack it into the `$HOME/android/ndk` directory. Also install SWIG[5], a generator for wrappers around code in other languages (in this case generating Java wrapper classes for C code).

The cryptographic functions originate from libsodium[6]. Download a copy into the `$HOME/OppNet/deps/libsodium` folder, then cross-compile shared libraries using the build scripts supplied by libsodium (in the `dist-builds` subdirectory). Copy the cross-compiled statically linked libraries (`libsodium-android-<arch>/*.a`) into the corresponding subdirectories in the `$HOME/OppNet/src/core/jni/<arch>` folder, and one set of header files into the `$HOME/OppNet/src/core/jni/include` folder.

In the `jni` directory there is a file called `sodium.i`, which is a SWIG interface file. It tells SWIG for which parts of the native code it should produce wrapper code, and it may need to be updated. If so, the C wrapper file `sodium_wrap.c` needs to be regenerated:

```
cd $HOME/OppNet/src/core/jni
swig -java -package org.abstractj.kalium \
    -outdir ../src/org/abstractj/kalium sodium.i
```

[3] https://code.google.com/p/protobuf/downloads/list
[4] https://developer.android.com/tools/sdk/ndk/index.html#Downloads
[5] http://www.swig.org/
[6] https://github.com/jedisct1/libsodium

Finally, the NDK can glue together the SWIG wrappers with the shared libraries:

```
cd $HOME/OppNet/src/core/jni
$HOME/android/ndk/ndk-build
```

The NDK places the resulting shared libraries in the `$HOME/OppNet/src/core/libs` directory, where they are picked up by Eclipse/ADT when compiling the APK the next time.

Bibliography

[1] Android distribution — droid life. http://www.droid-life.com/tag/distribution/. [Online; accessed 14-August-2014].

[2] Chance Miller. Google updates android distribution data for february, 80now on android 4.x.

[3] Sacha Trifunovic, Bernhard Distl, Dominik Schatzmann, and Franck Legendre. Wifi-opp: Ad-hoc-less opportunistic networking. In *ACM MobiCom Workshop on Challenged Networks (Chants 2011)*, Las Vegas, NV, USA, September 2011.

[4] Steven Meliopoulos and Suhel Sheikh. WLAN-OPP privacy. ftp://ftp.tik.ee.ethz.ch/pub/students/2012-FS/SA-2012-01.pdf, March 2012. [Online; accessed 14-August-2014].

[5] Teemu Kärkkäinen, Mikko Pitkänen, and Jörg Ott. Enabling ad-hoc-style communication in public wlan hot-spots. In *Proceedings of the Seventh ACM International Workshop on Challenged Networks*, CHANTS '12, pages 31–38, New York, NY, USA, 2012. ACM.

[6] Steven Meliopoulos. WLAN based opportunistic networking performance evaluation and optimization. ftp://ftp.tik.ee.ethz.ch/pub/students/2012-HS/MA-2012-31.pdf, 2013. [Online; accessed 14-August-2014].

[7] Sacha Trifunovic, Andreea Picu, Theus Hossmann, and Karin Anna Hummel. Slicing the battery pie: Fair and efficient energy usage in device-to-device communication via role switching. In *Proceedings of the 8th ACM MobiCom Workshop on Challenged Networks*, CHANTS '13, pages 31–36, New York, NY, USA, 2013. ACM.

[8] Cheshire S. and Krochmal M. Multicast dns. http://tools.ietf.org/html/rfc6762, February 2013.

[9] Cheshire S. and Krochmal M. Dns-based service discovery. http://tools.ietf.org/html/rfc6763, February 2013.

[10] Daniel J. Bernstein. Nacl: Networking and cryptography library. http://nacl.cr.yp.to/. [Online; accessed 14-August-2014].

[11] The serval project - mobile telephony for those in need. http://developer.servalproject.org/dokuwiki/doku.php?id=content:about. [Online; accessed 14-August-2014].

[12] Mikko Pitkänen, Teemu Kärkkäinen, Jörg Ott, Marco Conti, Andrea Passarella, Silvia Giordano, Daniele Puccinelli, Franck Legendre, Sacha Trifunovic, Karin Hummel, Martin May, Nidhi Hegde, and Thrasyvoulos Spyropoulos. Scampi: Service platform for social aware mobile and pervasive computing. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 7–12, New York, NY, USA, 2012. ACM.

[13] Carlos Anastasiades and Sascha Trifunovic. Secure content dissemination in opportunistic networks. ftp://ftp.tik.ee.ethz.ch/pub/students/2008-HS/MA-2008-27.pdf, 2009. [Online; accessed 14-August-2014].

[14] Android 2.2 platform highlights. https://developer.android.com/about/versions/android-2.2-highlights.html. [Online; accessed 14-August-2014].

[15] Wikipedia. Nitz — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=NITZ&oldid=617298817, 2014. [Online; accessed 14-August-2014].

[16] Wikipedia. Network time protocol — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Network_Time_Protocol&oldid=620920182, 2014. [Online; accessed 14-August-2014].

[17] Monsoon Solutions Inc. Power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/. [Online; accessed 14-August-2014].

[18] Wikipedia. Sha-1 — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=SHA-1&oldid=619460703, 2014. [Online; accessed 14-August-2014].