WLAN-OPP Privacy

Semester Thesis

Steven Meliopoulos
Suhel Sheikh

March 3, 2012

Advisors: **Bernhard Distl**
Sacha Trifunovic
Supervisor: **Prof. Dr. Bernhard Plattner**

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

Opportunistic networks are networks which are formed by participating users' devices without the need for a fixed network infrastructure, such as a stationary wireless access point. Due to the mobile nature of these networks devices are rapidly changing their communication partners without the user being aware of whom he is connected with. Various privacy concerns arise because users are not aware of whom they are sharing their data with and also whether the person they are consuming data from is a regular user or an adversary.

In our thesis we first implemented an Android application that establishes opportunistic networks (Wifi-Opp). Because most Android devices do not support ad-hoc wireless mode at this time we used the built-in wireless access point feature in order to communicate between devices without the availability of a fixed hotspot.

We then implemented a privacy protection method on top of the Wifi-Opp stack which enables the user to define friendship relations with other users. Our privacy protection implementation does not rely on devices having any fixed identification. This is important because it protects the user's movement from being tracked by allowing the device's identification to change randomly from time to time as it moves through different networks. Finally we evaluated the performance of our implementation and determined possibilities for improvements and future work.

Contents

Chapter 1

Introduction

Wireless networks are broadly used today and are available not only at people's homes but they are also offered at many public places for a fee or for free. These networks rely on a fixed infrastructure, usually a wireless access point connected to a wired network, which serves all the wireless users. Opportunistic networks do not rely on such a fixed infrastructure. They use a decentralized communication model, completely eliminating the dependency on a single device, enabling devices to communicate among each other everywhere at any time.

While the traditional network model theoretically offers connectivity between any two devices that have internet access, opportunistic networks still offer a number of advantages. Due to their decentralized nature they are resilient to censorship and can still operate when infrastructure fails, for example due to natural disasters. While the range of an opportunistic network at any given moment is limited by the reach of the wireless signal, information can still travel over large distances as an effect of the users' mobility.

With this decentralized approach used in opportunistic networks we no longer consume data from well-known web services, which we know and trust, but from neighbors in the network. Let's look at an example to give a possible use case of such networks: Alice and Bob are avid fans of the weekly podcasts recorded by Charlie and they are both subscribed to his podcast feed. They ride the same train every morning, but they do not know each other. Bob meets Charlie for dinner from time to time and during their conversations their smartphones automatically form an opportunistic network. The podcast application on Bob's device then recognizes that Charlie's device offers a new item in a feed Bob is subscribed to and downloads it right away. On Bob's commute home his and Alice's devices will form yet another opportunistic network giving Alice access to the latest episodes of the podcast.

Due to the fact that opportunistic networks are formed at random between devices of people who might not know each other there are privacy considerations to be made, some of which will be outlined in this paper.

We developed an application for mobile devices running the Android operating system that establishes opportunistic networks (Wifi-Opp). It continually tries to build networks with other participating devices in its vicinity and provides a list of available neighbors to other applications on the same device which can then initiate direct contact with neighbors as required.

On top of the opportunistic network module we implemented a privacy protection scheme (Privacy-Opp), which improves privacy on the opportunistic network in three ways. Firstly, it enables applications to define friendships (trusted devices) and thus making it possible for them to only exchange sensitive data with friends. At the same time Privacy-Opp allows the underlying Wifi-Opp service to change the device's identity frequently to avoid device tracking by adversaries. Finally, our implementation never sends raw friendship information over the network, thus protecting the user's social relationships.

In the following chapters we will present our implementation in more detail.

1.1 Related Work

Our work largely builds on related work done by the Podnet group at ETH Zurich. In particular we built our implementation based on the paper on Ad-hoc-less Opportunistic Networking [1], which outlines a general implementation of opportunistic networks, and the paper about Privacy in Opportunistic Networks with Reputation [2].

Chapter 2

Design

There are two major components called Wifi-Opp and Privacy-Opp, which are implemented as two completely separate Android applications. Additionally there are two smaller applications, which make use of the Wifi-Opp API for both testing and demonstration purposes.

2.1 Wifi-Opp Application

The Wifi-Opp application can be divided into two main parts. One is an Android service that runs continually and provides all the functionality necessary for creating an opportunistic network. The second part is a graphical user interface that shows status information and allows for configuration of the service's parameters and manual operation.

2.1.1 Wifi-Opp service

In a truly decentralized network, direct peer to peer connections would be made between devices that are within each other's reach. However since at this time Android does not offer wireless ad-hoc networking, we build our opportunistic networks by having devices connect to a wireless access point, then detecting other participating devices on the same network and relying on the access point to forward the traffic between two devices.

Android devices have the capability to act as a wireless hotspot which is originally intended to share their cellular internet connection with other devices such as laptops when no other connections are available. This feature can however also be utilized for our purposes. The Wifi-Opp service activates the device's wireless hotspot function after randomized intervals, which allows for the formation of an opportunistic network as soon as at least one more participating device is within reach. The downside of this approach is that the device's cellular internet connection is automatically shared with connected devices. To avoid this, the Wifi-Opp service ensures that the device's cellular data connection is disabled and restores its previous state when the service is terminated.

However, especially in urban areas, there are also many publicly available wireless hotspots. Even though many of them do not offer Internet connectivity for free, they still allow any device to connect to their network so that one can then authenticate as a registered user or make a payment through a web interface to gain access to the Internet. And some of these hotspots forward local traffic between connected devices even for unauthenticated devices. Our application leverages this fact by connecting to these networks as well with a certain probability. This has the advantage that in this case none of the participants of the opportunistic network need to take on the role of the access point which would cause an increased power consumption that is significant for mobile devices.

The Wifi-Opp service continually observes the available WiFi networks, which might be provided either by infrastructure access points or other participating devices. It then either selects one network to connect to or it decides to set the own device into access point mode so that others can connect to it. This decision is delegated to a component called the strategy, which will be covered in more detail in the following chapter.

Once the device is part of a network the service keeps track of all available communication partners on the same network. This is done by continuously sending out beacons containing the own device ID in short intervals. When a beacon is received, the neighbor with the contained ID is added to the list of neighbors. It is removed if no beacon from this neighbor is seen for a certain duration. The Wifi-Opp service shares this list with other applications running on the devices, such that they can then initiate direct communication with neighboring devices.

Applications which want to use the Wifi-Opp service can start it using an Android intent. The service will register all applications which called it and automatically terminate when all applications unregistered. Optionally, applications can also inform the Wifi-Opp service about their desire to keep the current connection alive, such that the service can defer changes to the current Wi-Fi connection, allowing the application to finish its current communication.

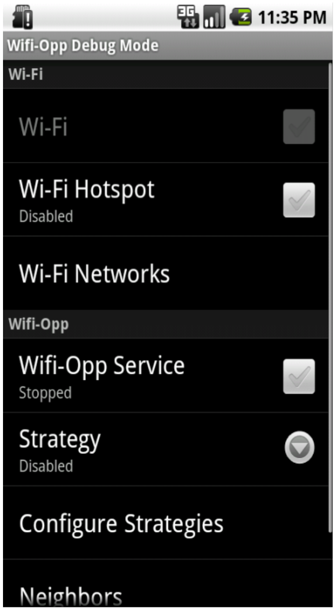
2.1.2 Configuration Interface

The entry point of the configuration interface is the MonitorActivity. It is the intended interface for an end user of the Wifi-Opp application. It allows the user to start and stop the Wifi-Opp service and displays status information such as whether Wi-Fi or the mobile hotspot is enabled, network information, the currently active strategy and a list of current neighbor devices.

The other part of the configuration interface is the DebugActivity, which offers a wide variety of configuration options and is useful for testing and further development of the application. In addition to all the functionality in the MonitorActivity it allows the user to manually enable or disable Wi-Fi and the access point mode, select a network to connect to, select and configure the available strategies and access the service's log file.

2.2 Privacy-Opp

The second major component of our project is the Privacy-Opp application. Its purpose is to provide the user with increased privacy as outlined in the introduction. We use parts of the method presented in "Privacy in Opportunistic



11:35 PM

Wifi-Opp Debug Mode

Wi-Fi

Wi-Fi ☑

Wi-Fi Hotspot ☑
Disabled

Wi-Fi Networks

Wifi-Opp

Wifi-Opp Service ☑
Stopped

Strategy
Disabled

Configure Strategies

Neighbors

Figure 2.1: DebugActivity

Networks with Reputation" [2] to improve privacy in the opportunistic network. The general idea is to have a list of trusted neighbors with whom we want to exchange sensitive data. A simple approach would be to give every device a fixed identification and keep a list on each device that holds all the IDs of trusted devices. We did not choose this approach because it comes with a big implication on privacy caused by forcing the device to use a persistent ID: The fixed identification enables adversaries to track devices without any difficulty. An adversary could install observing devices at key locations (e.g. at the train station, work and home) and thereby track a user's behavior throughout the day.

Our implementation does not rely on any fixed device identification, making tracking much more difficult. This is achieved by storing information about friendships instead of device IDs. A friendship consists of a friendship identification number and a symmetric encryption key. In order to decide whether a neighbor is a friend or not, devices will send out a bloom filter containing all friendship identification numbers known to them. The bloom filter only supports querying for specific elements but not listing all of its contents. Therefore the user's social relationships are still kept private. The recipient can only check if any entries of his own friendship list produce a hit. Trying all possible IDs is impractical if the space of possible friendship IDs is chosen large enough (for a more detailed analysis see [2]). If any of the recipient's local entries produces a hit in the bloom filter they will enter a second phase to determine if the matched friendship is indeed present on both devices, because bloom filters produce false positives at a certain rate. This is done by sending a challenge encrypted with the friendship's private encryption key. If the message can be successfully decrypted, they will determine that they are friends.

2.3 Demo Applications

There are two demonstration applications included in the Wifi-Opp project. Ping-Opp is an application that demonstrates how applications consume the provided neighbor list in order to create a connection to a neighboring device. The application repeatedly pings all current neighbor devices and shows the number of neighbors, number of received pings and a log of all received pings. This application is also very useful to test different Wifi-Opp strategies because it demonstrates for how long devices are actually able to communicate. The second application is a very simple demo, which shows how applications can inform the Wifi-Opp service about their desire to keep the current wireless connection alive.

Chapter 3

Implementation

This chapter will describe some of our implementation decisions.

3.1 Service Properties

The creation of opportunistic networks is handled completely by the Wifi-Opp service. It is therefore important, that it runs continuously and reliably. Being an operating system for mobile devices, which typically have limited resources, Android kills processes when more memory is needed for other tasks with higher priority. To prevent our service from being killed, we declared it to be a foreground service. This has the additional benefit that a notification icon is displayed in the status bar. The constant use of wireless communication by our application causes considerable power consumption and randomly activating the mobile hotspot function as well as deactivating the cellular data connection might interfere with the user's activities. It is therefore desirable that the user is informed at all times about whether the service is running.

In addition, we assigned the service its own process so that the rest of the application (i.e. the configuration interface) can be terminated independently.

3.2 Starting and stopping the Service

As mentioned previously, the Wifi-Opp service has a strong influence on the device's state. Therefore the Wifi-Opp service should only be running when it is needed. In addition, the user should not have to activate and deactivate it manually through the configuration interface but client applications should be able to start it when they require its services. Clients should however not be able to shut the service down at will as this might interfere with other clients' needs. The service should rather terminate once none of the clients requires it anymore. Only the application's own configuration interface should give the user the possibility to stop the service prematurely so that he does not need to track down and end every single client application individually.

Android services can be started and stopped by sending Intents via the `startService` and `stopService` methods. Using these methods, keeping track of active clients would be tricky and unreliable. The sender of an Intent can not be determined and sloppily implemented clients that fail to send the stop signal could keep the service running indefinitely, draining the battery for no reason.

Luckily, Android also provides the `bindService` method, which starts a service if it is not already running and creates a connection to it. This connection could be used for function calls between processes but this is not needed in our case. The service automatically terminates once all clients have unbound. Even if a client crashes before he can unbind properly this is still detected due to loss of the connection. This is exactly the functionality we are looking for, except that this would not allow for premature termination of the service by the user.

To achieve the desired result we combined both approaches. The Wifi-Opp service is only controlled by `startService` and `stopService` calls from within the application. Additionally there is a second dummy service that accepts `bindService` calls from external clients, forwards a `startService` call to the WifiOpp service each time it is bound, and terminates it when it is killed itself.

3.3 Neighbor List

Wifi-Opp allows access to the current neighbor list by implementing an Android ContentProvider. This is the standard way of providing content to a different process on Android and it is also used by Android itself to give applications access to things such as the user's phone book. Applications implement a ContentResolver which is linked to the Wifi-Opp ContentProvider URI. They can then query the list or register a ContentObserver, which will get triggered every time the neighbor list's content changes. A sample implementation of a ContentObserver can be found in the Ping-Opp application.

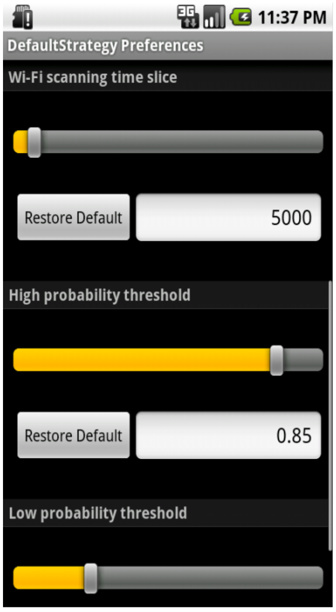
Apart from querying, ContentProviders in Android usually provide methods for inserting, updating, and deleting entries that are available to every application that has access via a ContentResolver. In our case the list should of course not be modifiable by other applications. Therefore our implementation throws an `UnsupportedOperationException` when one of them is called. To insert the neighbor information we added additional private methods instead that are only accessible from within our application.

3.4 Wifi-Opp Strategies

As explained in the design section, strategies are responsible for telling our service when and how to establish Wi-Fi connections or whether the device's wireless hotspot feature should be enabled or disabled. Therefore the strategy largely determines the stability of the opportunistic networks. The optimal strategy varies strongly between different scenarios. Possible goals could be to encounter as many different devices as possible, to stay connected to them as long as possible once we found them, to use as little power as possible, or combinations thereof. And even once the aim is clearly defined, optimizing the parameters requires a great amount of experimentation. For these reasons our main focus was not on implementing the perfect strategy but to make strategies easy to create, interchange, and configure. We did however provide a fairly exhaustive example strategy with our project, showing how strategies can be implemented and how they can take advantage of parameter variables.

Creating a new strategy only requires extending the Strategy base class, implementing the decision logic in the execute method, and adding it to the list of available strategies. Other than that, no code in the rest of the application needs to be touched. Common tasks such as keeping an updated neighbor and Wi-Fi network list are taken care of in the Wifi-Opp service or the Strategy base class.

To make strategies easily configurable we provide a small framework for managing parameters. The StrategyPreference class represents a single parameter of a strategy such as a probability threshold or a time interval. StrategyPreference objects have a default value, a range of valid values, and they store their current value persistently. We provided implementations for integer and floating point numeric parameters, which should cover most needs but further types can be added easily. A strategy simply needs to publicly list all its parameters and then use them internally to retrieve their current values. Each preference also generates a ViewGroup comprised of interface components for changing the current value according to its type and its valid range. This allows for the dynamic creation of a complete configuration interface for any strategy.



DefaultStrategy Preferences

Wi-Fi scanning time slice

5000

High probability threshold

0.85

Low probability threshold

Figure 3.1: Dynamically generated strategy configuration interface

Due to this loose coupling it is possible to change the currently employed strategy and all of its parameters at runtime. This makes experimentation in the field extremely easy and fast as no computer is needed for recompilation.

3.5 Friendship Pairing

Many different approaches to pairing two devices through an insecure network connection exist. A much used technique is to establish a shared private key using Diffie Hellman key exchange, which is known as a secure method to establish keys in an open network. However Diffie Hellman and similar key exchange algorithms are not inherently immune against man-in-the-middle attacks. There exist techniques to protect against man-in-the-middle attacks, for example by showing a hash of the established secret key on both devices. If the shown hashes dont match the users know they have established a connection to a different device and can try again. We chose to implement a different approach, which does not rely on using the insecure network channel for key establishment at all. Instead our Privacy-Opp application displays a QR code containing the device's current identification number and a private key that is created randomly. Two devices can now be paired very simply by one of them scanning the QR code shown on the other device. For scanning QR codes we use the ZXing barcode scanner [3] and we generate QR codes using the Google Chart API [4].

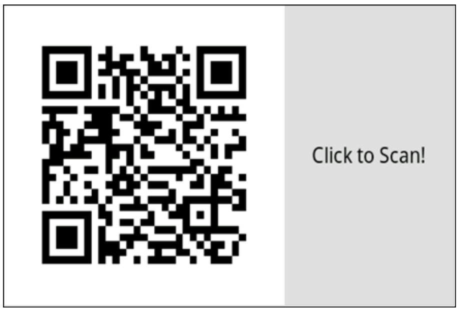


Figure 3.2: QR code used for establishing friendships

Using the camera as a side channel for key establishment comes with the big advantage that adversaries have no possibility to perform eavesdropping or man-in-the-middle attacks. The ZXing barcode scanner needs to be installed on both devices in order to create and read the QR code. Alternatively, if one device has an active internet connection, that device does not need to have the ZXing application installed as it will automatically use the Google Chart API to create a QR code.

Chapter 4

Documentation

4.1 Using the Configuration Interface

The configuration interface allows the user to interact with the Wifi-Opp service. There is a regular mode and a debug mode which gives the user even more possibilities to access and change Wifi-Opp settings.

4.1.1 Regular Mode

The following elements are available on the simple screen which is shown first when the application is launched:

Wi-Fi Shows whether Wi-Fi is enabled, the connectivity status and the current network's SSID as well as the device's IP address.

Wi-Fi Hotspot Shows whether the device's Wi-Fi hotspot feature is enabled.

Wifi-Opp Service Displays the currently selected Strategy. The user can manually start and stop the service from here.

Current Strategy Displays the strategy that the service is currently set to use (can be changed in debug mode).

Neighbors Displays the currently available number of neighbors. Clicking it opens a new activity that lists all the neighbors' device IDs and IP addresses.

4.1.2 Debug Mode

The debug interface contains all the elements of the regular mode and adds some additional functionality.

Wi-Fi Same as in regular mode. Additionally allows enabling and disabling Wi-Fi.

Wi-Fi Hotspot Same as in regular mode. Additionally allows enabling and disabling the device's Wi-Fi hotspot feature. Turning on the hotspot automatically disables Wi-FI and vice versa.

Wi-Fi Networks Opens a new activity showing currently available networks which are either open (unencrypted) or already configured on the device. The signal strength in dBm is shown next to the SSID. Clicking an entry will cause the Wifi-Opp service to change its current connection to the selected network. Encrypted networks cannot be connected to from within this activity. Use Android's networking settings for this.

Mobile Data Displays and allows manual toggling of the cellular data connection's state.

Wifi-Opp Service Same as in regular mode.

Strategy Displays the currently selected Strategy. Clicking on this option will display a list of all available strategies to choose from.

Configure Strategies Opens a new activity showing all strategies defining parameter variables. Clicking one of the listed strategies will open an automatically generated settings activity where all parameter variable values can be manually set within the limits the strategy defined for them.

Neighbors Same as in regular mode.

Log Opens a new activity where logging can be enabled. When logging is enabled the activity shows all log messages with their corresponding timestamps. There are options to enable and disable the log's auto-scrolling feature and to clear the log. The log contains all messages added through Wifi-Opps sendToLog method, such as sent/received beacons, connection changes, strategy decisions and more. The log is written to a file on the device's external storage (e.g. an SD card) and can also be retrieved by connecting the device to a computer.

4.2 Writing New Strategies

Adding new strategies is very simple. We have provided a fairly exhaustive sample strategy in the project (`DefaultStrategy.java`) which demonstrates how we suggest strategies should be written. This is however just a guideline and different approaches could be valid as well. The process of creating a new strategy is outlined in the following:

1. Create a new class in the package `ch.ethz.csg.burundi.strategies` and make it extend the Strategy class (`Strategy.java`).

2. Add the default constructor and call `super(WifiOppService)` in it.

3. Implement the mandatory `execute()` method in order for the code to compile. This is the basic outline of a minimal strategy implementation:

   ```
   package ch.ethz.csg.burundi.strategies;

   public class NewStrategy extends Strategy
   {
           public NewStrategy (WifiOppService
               wifiOppService)
   ```

```
        {
                super(wifiOppService);
        }

        @Override
        protected int execute()
        {
                // your decision logic here
                return timeout; // timeout specifies
                    when the strategy should be called
                    again
        }
}
```

4. The final step is making the new strategy available by adding its class attribute to the `STRATEGIES` list in `WifiOppService.java` inside the `ch.ethz.csg.burundi` package.

After completing all the above steps and compiling the application the newly added strategy will be visible in the debug activity's strategy picker dialog.

4.2.1 The execute Method

The strategy's complete functionality is contained in the `execute` method. Its execution consists of three stages:

1. Determining which actions to take based on any information that is available and appropriate.
2. Making the according changes to the connectivity, e.g. selecting a network to connect to or enable the device's mobile hotspot. The strategy base class already implements all the methods needed for changing the connectivity. It should be consulted before writing a new strategy.
3. Returning the waiting time in milliseconds before the next execution.

4.2.2 Automatically Updated Variables

For convenience the following three often used variables are automatically updated by the strategy base class and are available to all subclasses:

wifiOppNetworks A list containing all available hotspots created by other devices running the Wifi-Opp service. The items of this list can be passed to the `connectToWifi` method.

publicNetworks A list containing all available open (unencrypted) and also all available encrypted networks which are already configured on the current device. The items of this list can be passed to the `connectToWifi` method.

neighbors A list containing all available neighbors on the network the device is currently connected to. Use `neighbors.size()` to check whether other Wifi-Opp users are present on the current network.

4.3 Using Wifi-Opp as a Client Application

4.3.1 Starting the Service and Registering as a Client

If a client application requires the Wifi-Opp service to be running it can use the following code to create a binding:

```
private ServiceConnection connection = null;
private void bindWifiOppService()
{
        connection = new ServiceConnection()
        {
                @Override
                public void onServiceDisconnected(
                    ComponentName arg0 )
                {
                        connection = null;
                }
                @Override
                public void onServiceConnected(
                    ComponentName arg0, IBinder arg1 )
                {
                        // do nothing
                }
        };
        bindService( new Intent( "ch.ethz.csg.burundi.
            BIND_SERVICE" ), connection, Context.
            BIND_AUTO_CREATE );
}

private void unbindWifiOppService()
{
        if (connection != null)
        {
                unbindService( connection );
                connection = null;
        }
}
```

Between the calls to `bindWifiOppService` and `unbindWifiOppService` the service is guaranteed to run except if it is manually terminated by the user through the configuration interface. The service must be unbound before the client application terminates or else the connection is leaked.

4.3.2 Marking the Current Connection as Used

If a client application is using the current network connection, for example because a file transfer with a neighbor is in progress, it can communicate to the Wifi-Opp service that it would prefer the connection to be maintained. This is done by sending a broadcast intent with the action string `"ch.ethz.csg.wifiopp.announceConnectionUsed"`. The connection will be marked as used for the next five seconds. Note however that it depends entirely on the active strategy whether this information is considered!

4.3.3 Obtaining the Neighbor List

Wifi-Opp provides access to the current neighbor list through Android's ContentProvider [5]. To access the neighbor list a ContentResolver pointing to the URI of the neighbor list is needed. A sample implementation can be found in the Ping-Opp demo application. The following code snippet can be used to register a listener which will get notified whenever the neighbor list changes.

```
neighborObserver = new ContentObserver(new Handler())
{
        @Override
        public void onChange(boolean selfChange)
        {
                super.onChange(selfChange);
                        // Do stuff here...
        }
};
getContentResolver().registerContentObserver(Uri.parse("
    content://ch.ethz.csg.burundi.NeighborProvider/dictionary
    "), false, neighborObserver);
```

To obtain a cursor, call the ContentResolver's query method using the same URI as in the above example. The available columns are defined in `NeighborProvider.java`. They are:

_id A unique row ID (e.g. for use in a CursorAdapter)

device_id The neighbor's current device identification number

ip The neighbor's current IP address

Chapter 5

Evaluation

5.1 Power Consumption

We measured our application's power consumption while running different strategies using the Power Tutor application that was developed at the University of Michigan [6]. The results in table 5.1 present the average of two measurements per strategy over 40 minutes. During the tests five devices were present in total.

	Wifi-Opp [J/min]	System [J/min]
WifiOnly	0.288	0.453
APOnly	0.349	0.169
Default	0.382	0.663

Table 5.1: Power consumption measurements

The results suggest that from a power saving perspective, activating the hotspot is undesirable. The strikingly low value for the system's power usage using the APOnly strategy stems from the fact that the power used for the wireless antenna cannot be measured when the hotspot is activated.

5.2 Connectivity

The second experiment we conducted aimed at evaluating the effectiveness of our default strategy in terms of finding and staying connected to other devices. Before every round, the strategy selects a network to connect to, prioritizing those created by other participating devices over public hotspots as they guarantee the presence of at least one neighbor. If neither is available the device will activate its own access point feature for one round and only with a small probability because of the increased power consumption. If the device is currently connected to a network on which neighbors are present, the strategy will leave the connection unchanged with high probability. Using the default settings, round lengths vary between 10 and 60 seconds and are randomized in order to prevent the synchronization of devices.

The setup consisted of five devices that were left stationary once for 70 minutes and once for 20 minutes, each running the Wifi-Opp application using the default strategy. There was one public infrastructure Wi-Fi network present in addition to the ones that the devices would create themselves.

Figure 5.2 illustrates the average fraction of the total time during which a certain number of neighbors was visible to the devices.

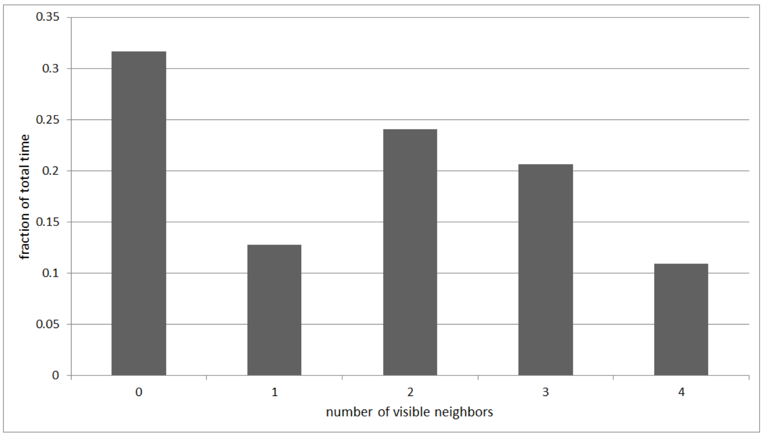


Figure 5.1: Results of connectivity experiment

During roughly one third of the total time the devices see no neighbors at all. This consists of the time spent being connected to a network on which no other devices are present and the time needed to change the connection, which on average occurred every 82 seconds. The rest of the time appears to be distributed along a bell curve, as it is to be expected, however more data points would be needed to confirm this. The first neighbor was discovered after 64 seconds on average and the average time until every other device had been seen once was 108 seconds. The amount of time during which there are neighbors visible and the time until discovery present a trade-off that depends on the strategy's parameters.

5.3 Pairing

The three most time-consuming operations during pairing (see appendix A.1) are the creation of challenges, solving of challenges and data transmission. Depending on the Bloom Filter's false positive probability the relative time-consumption is shifted either towards data transmission (for low false positive rates) or towards generation and solving of challenges (for high false positive rates). In order to achieve a good overall performance we measured pairing times using different false positive rates. The results are shown in figure 5.3. There are three local minima at 20%, 60% and 90%. We noticed that error rates over 50% caused pairing to fail in some cases, which is why we determined that 20% is the optimal false positive rate for our purposes. We later noticed that pairing in those cases in fact does not fail but takes an absurd amount of time. Unfortunately we could not determine the cause for this behavior.

A very important property of our pairing application is how many friendships it can handle while still providing acceptable performance. To check whether our pairing implementation can handle a high number of friendships we created

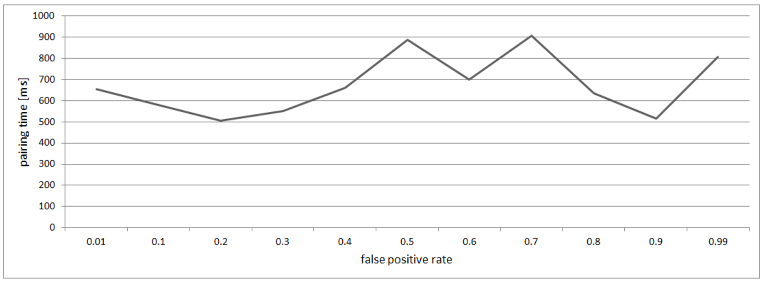


Figure 5.2: Pairing time as a function of the bloom filter's false positive rate averaged over ten measurements

random friendship data on both devices and measured the total pairing time for different numbers of friendships. Our measurements (see figure 5.3) suggest that our implementation's time-consumption is linear with respect to the number of neighbors, which was expected given that all three most time-consuming operations we mentioned before have linear time-dependencies with respect to the number of neighbors.

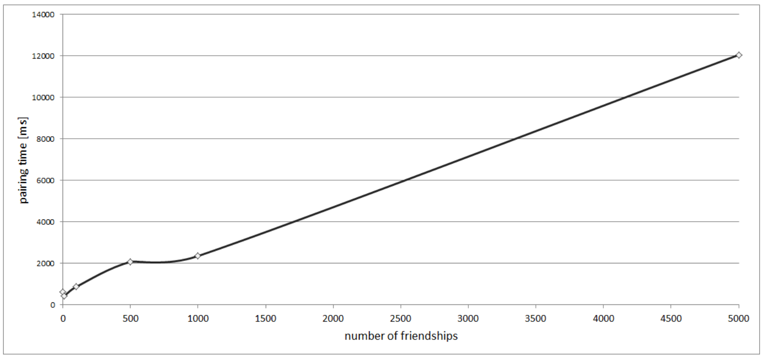


Figure 5.3: Pairing time as a function of the number of friendships averaged over 30 measurements

Our implementation provides reasonable performance for up to 1000 friendships and good performance for up to 100 friendships. A possible way to improve performance would be to change the algorithm to work asymmetrically with only one device sending challenges and the other participant using the correct solution to inform the former about their friendship.

Chapter 6

Conclusion

The application suite we developed serves as proof that opportunistic wireless networking is possible using only functionality that is available on standard versions of Android supporting API level 7 and upwards, which covers 98.4% of currently active Android devices.

The absence of ad-hoc networking capabilities in standard Android devices was handled by continuously creating and joining new infrastructure networks, detecting possible communication partners by sending beacons and having messages to them relayed by the access point. All required permissions to change the state of the device's WiFi module and preventing the sharing of the cellular data connection have been shown to be available to regular applications. ContentProviders are used to share the current neighborhood information with multiple other applications running in different processes simultaneously and in real time. Simple client applications were created as examples to verify this functionality.

The Privacy-Opp application demonstrates that the proposed privacy protection scheme relying on the comparison of friendship IDs is viable. Friendship establishment and checking is done within a few seconds. The friendship creation mechanism we implemented based on QR codes has the advantages that it uses a separate channel and the scanning of a code by the camera, which is a standard feature in current devices, is quicker, more reliable and more convenient than having the user enter a code manually.

6.1 Future Work

The Wifi-Opp application itself, which provides the basic networking functionality runs stable. There is however room for improvement regarding certain qualities. Power consumption could possibly be reuced by sending beacons more sporadically or in a dynamic manner depending on the current state of connectivity and available neighbors.

The strategies that govern the decisions on creating and joining networks could be further developed in various ways. We included a default strategy which is a best effort for achieving an all purpose solution, given our available time and resources. Our strategy bases its decisions on input values such as the type of available networks, the current number of neighbors and whether a client application is utilizing the current connection. Various other kinds of available information could however also be included. Power intensive tasks could be reduced when the battery level is low or the strategy could choose to only operate while the device's screen is turned off so as not to interfere with the user's other activities. Optimizing for example the amount of time during which neighbors are visible or the number of neighbors that are detected in the course of a day would require extensive measurements and experiments that model the users' mobility in a realistic manner.

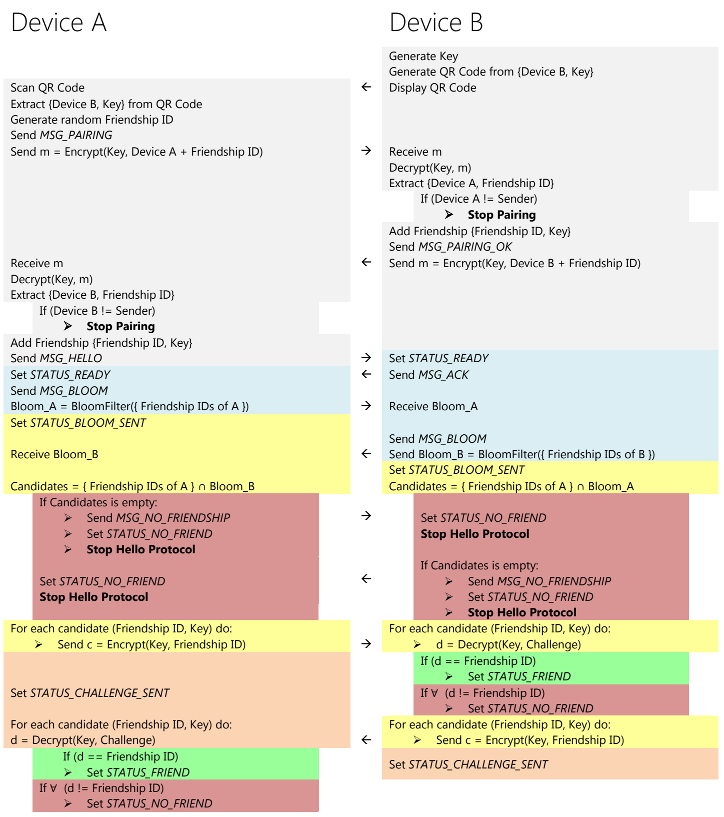
Evaluating the effective usefulness of the basic Wifi-Opp service as well as the friendship detection application would require the creation of more elaborate client applications such as a podcasting or file sharing system and running tests with a large user base.

It is conceivable that in the future Android will support ad-hoc Wi-Fi networking or a new wireless networking technology which would allow direct peer-to-peer connections between devices. In this case the underlying Wifi-Opp service would have to be reimplemented, potentially yielding even better results.

Appendix A

Diagrams

A.1 Pairing



Device A		Device B
		Generate Key
		Generate QR Code from {Device B, Key}
Scan QR Code	←	Display QR Code
Extract {Device B, Key} from QR Code		
Generate random Friendship ID		
Send *MSG_PAIRING*		
Send m = Encrypt(Key, Device A + Friendship ID)	→	Receive m
		Decrypt(Key, m)
		Extract {Device A, Friendship ID}
		If (Device A != Sender)
		➢ **Stop Pairing**
		Add Friendship {Friendship ID, Key}
		Send *MSG_PAIRING_OK*
Receive m	←	Send m = Encrypt(Key, Device B + Friendship ID)
Decrypt(Key, m)		
Extract {Device B, Friendship ID}		
If (Device B != Sender)		
➢ **Stop Pairing**		
Add Friendship {Friendship ID, Key}		
Send *MSG_HELLO*	→	Set *STATUS_READY*
Set *STATUS_READY*	←	Send *MSG_ACK*
Send *MSG_BLOOM*		
Bloom_A = BloomFilter({ Friendship IDs of A })	→	Receive Bloom_A
Set *STATUS_BLOOM_SENT*		
		Send *MSG_BLOOM*
Receive Bloom_B	←	Send Bloom_B = BloomFilter({ Friendship IDs of B })
		Set *STATUS_BLOOM_SENT*
Candidates = { Friendship IDs of A } ∩ Bloom_B		Candidates = { Friendship IDs of A } ∩ Bloom_A
If Candidates is empty:		
➢ Send *MSG_NO_FRIENDSHIP*	→	Set *STATUS_NO_FRIEND*
➢ Set *STATUS_NO_FRIEND*		**Stop Hello Protocol**
➢ **Stop Hello Protocol**		
		If Candidates is empty:
Set *STATUS_NO_FRIEND*	←	➢ Send *MSG_NO_FRIENDSHIP*
Stop Hello Protocol		➢ Set *STATUS_NO_FRIEND*
		➢ **Stop Hello Protocol**
For each candidate (Friendship ID, Key) do:		For each candidate (Friendship ID, Key) do:
➢ Send c = Encrypt(Key, Friendship ID)	→	➢ d = Decrypt(Key, Challenge)
		If (d == Friendship ID)
		➢ Set *STATUS_FRIEND*
Set *STATUS_CHALLENGE_SENT*		If ∀ (d != Friendship ID)
		➢ Set *STATUS_NO_FRIEND*
For each candidate (Friendship ID, Key) do:		For each candidate (Friendship ID, Key) do:
d = Decrypt(Key, Challenge)	←	➢ Send c = Encrypt(Key, Friendship ID)
If (d == Friendship ID)		Set *STATUS_CHALLENGE_SENT*
➢ Set *STATUS_FRIEND*		
If ∀ (d != Friendship ID)		
➢ Set *STATUS_NO_FRIEND*		

If a message is received while the receiving device is not in the expected state according to above diagram, the message is dropped.

Bibliography

[1] Sacha Trifunovic, Bernhard Distl, Dominik Schatzmann, and Franck Legendre. Wifi-opp: ad-hoc-less opportunistic networking. In *Proceedings of the 6th ACM workshop on Challenged networks*, CHANTS '11, pages 37–42, New York, NY, USA, 2011. ACM.

[2] Bernhard Distl, Franck Legendre, and Bernhard Plattner. Privacy in opportunistic networks with reputation.

[3] Zxing - Multi-format 1D/2D barcode image processing library with clients for Android, Java. `http://code.google.com/p/zxing/`.

[4] Google Chart QR Code API. `http://code.google.com/intl/de-DE/apis/chart/infographics/docs/qr_codes.html`.

[5] Android ContentProvider. `http://developer.android.com/guide/topics/providers/content-providers.html`.

[6] Power Tutor. `http://powertutor.org`.