NTNU – Trondheim
Norwegian University of
Science and Technology

Using Infrastructure-less Wireless Networks to Synchronize Data among Mobile Devices

Extending UbiShare

Kato Stølen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Babak Farshchian, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NORWEGIAN UNIVETSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Master of Science in Computer Science

Using Infrastructure-less Wireless Networks to Synchronize Data among Mobile Devices

by Kato Stølen

During the last ten years, social networking has become an integral part of modern society. People use the wide range of available social networking services to create relationships and communities, in which they share data with family, friends and co-workers. These social networking services works great for their intended purpose, connecting people, but a privacy issue arises when using such services. Even though you have the ownership of everything you share, you often grant the service provider the possibility to use your data as they please.

UbiShare was proposed as a solution of this problem. By using UbiShare, innovative sharing applications could provide smaller, private communities, such as rescue crews, a way of communicating while in action. The idea was to use private social networks to enhance disaster management by providing an easy and intuitive way of sending messages and sharing data that would aid in decision making. This way, disaster management would be much easier to coordinate, but if the disaster wipes out communication infrastructure, the current version of UbiShare would not work.

This thesis presents the development and evaluation of a data synchronization system that uses infrastructure-less wireless networks, such as Bluetooth or Wi-Fi Direct, as communication channel. By making this an extension of UbiShare, community members in close proximity can continue to share and synchronize data even when fixed internet infrastructure is inaccessible. The motivation behind this system is to show how it can enhance disaster management in situations when only infrastructure-less wireless networks are available.

(Norwegian)

Sosiale nettverk har, de siste ti årene, blitt en del av det moderne samfunn. Folk bruker de utallige sosiale nettverkene til å lage relasjoner til andre og grupper hvor de kan dele data med familie, venner og kolleger. Disse sosiale nettverkene fungerer veldig bra for sitt formål, å knytte mennesker sammen, men det er ikke alt som egner seg å dele gjennom de. Selv om du eier det som blir delt, må du ofte tillate at de sosiale nettverkene kan bruke det du deler som de vil.

UbiShare ble utviklet som en løsning til dette problemet. Innovative datadelingsapplikasjoner kan bruke UbiShare til å gi mindre, private grupper, som for eksempel redningsmannskap, en måte å kommunisere på mens de er i aksjon. Ideen var å bruke private sosiale nettverk til å forbedre katastrofehåndtering ved å tilby en enkel og intuitiv måte å sende meldinger og dele data som kan hjelpe med å ta avgjørelser. På denne måten blir katastrofehåndtering enklere å koordinere, men dersom katastrofen ødelegger kommunikasjonsinfrastruktur, vil den nåværende versjonen av UbiShare ikke fungere.

Denne masteroppgaven presenterer utviklingen og evalueringen av et datasynkroniseringssystem som bruker infrastrukturløse trådløse nettverk, som for eksempel Bluetooth og Wi-Fi Direct, til å kommunisere mellom mobile enheter. Ved å gjøre dette til en utvidelse av UbiShare, kan gruppemedlemmer i nærheten av hverandre fortsette å dele og synkronisere data, selv om internettinfrastruktur ikke er tilgjengelig. Motivasjonen bak dette systemet er å vise hvordan det kan brukes til å forbedre katastrofehåndtering når bare infrastrukturløse trådløse nettverk er tilgjengelig.

Acknowledgements

I would like to thank my supervisor, Babak Farshchian, for the continuous feedback and help during this master thesis. His encouragement and input made the project exciting to work with.

I would also like to thank my parents for their unconditional love and support during the project. Mamma, I'm coming home. At least for the summer holiday.

Contents

List of Figures

List of Tables

Abbreviations

AGAPE	**A**llocation and **G**roup **A**ware **P**ervasive **E**nvironment
API	**A**pplication **P**rogramming **I**nterface
DBMS	**D**ata**b**ase **M**anagement **S**ystem
ER	**E**ntity-**r**elationship
IP	**I**nternet **P**rotocol
JSON	**J**ava**S**cript **O**bject **N**otation
JXTA	**J**u**xta**pose
MANET	**M**obile **A**d-hoc **Net**work
NFC	**N**ear **F**ield **C**ommunication
NTNU	Norwegian University of Science and Technology
P2P	**P**eer-**to**-**P**eer
RFID	**R**adio-**f**requency **Id**entification
SN	**S**hared-**n**othing
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
UML	**U**nified **M**odeling **L**anguage
WAP	**W**ireless **A**ccess **P**oint
XML	**Ex**tensible **M**arkup **L**anguage
XMPP	**Ex**tensible **M**essaging and **P**resence **P**rotocol

Chapter 1

Introduction

This chapter introduces the master thesis by presenting its background, problem description and goal.

1.1 Thesis Backgroud

This thesis is a continuation of the specialization project. In the course *TDT4501 Computer Science, Specialization Project* students learn how to complete a larger, independent project and how to go into a specific problem using scientific methods [1]. This course introduced a project where the goal was to develop an innovative sharing tool called UbiShare. UbiShare was developed for Android and its purpose was to help other mobile applications to share data in a group of mobile devices. The original project description was the following:

> *"It has become a known fact that even if users on e.g. Facebook have hundreds of friends, they interaction only with a small fraction of their friends at any time. Social media such as Facebook are therefore implementing functionality for supporting smaller groups of people share information with each other. In UbiCollab we are developing an innovative sharing tool for Android devices called UbiShare. UbiShare allows users to share information about their physical environment. This task aims to implement the next generation of UbiShare based on cutting edge technologies such as XMPP and Android."* [2]

Due to some complexities with XMPP, the project description was slightly modified. Instead of using XMPP and a peer-to-peer protocol, UbiShare would use a centralized online storage service to store the master copy of the shared data.

1.1.1 UbiShare

UbiShare is a service that lets Android applications create small, private social networks. The purpose of UbiShare was to inspire the development of mobile applications that use innovative ways of sharing data within communities. An example of such application is one that helps rescue crews communicate when in action. This application would use UbiShare to create a community and synchronize data between the members of the community. UbiShare was developed to provide an alternative to sharing data through larger, less private social networks, such as Facebook or Google Plus, but also to have the possibility to interact with these [3]. To put UbiShare into perspective, we can consider the following scenario:

Scenario: Crowd Management There is a concert with an audience of several thousand people. A couple of hundred crew members are hired to manage the crowd. The crew members are equipped with a mobile device that can send and receive messages and share the location of the crew members within the crowd management community. UbiShare is used to synchronize the data among the crew members. The mobile devices are used to share the status of a crew member's assigned area. A coordinator uses these statuses, in addition to the location of the crew members, to manage the crew members. If an accident happens, the coordinator can easily choose the most suitable crew members to come to aid. Also, if a severe accident happens and people have to be evacuated, the coordinator can notify the crew members of where to evacuate the crowd, based on the statuses of the different areas.

The scenario above shows how UbiShare can be used to share data within a community.

After the completion of the first version of UbiShare, there was a growing desire for UbiShare to work without an internet connection. This would make UbiShare usable in places and situations where an internet connection is nonexistent, such as disaster management in places where internet infrastructure has been wiped out or is inaccessible for some other reason. It is this idea that forms the base of this thesis.

1.2 Problem Description

During the last ten years, social networking has become an integral part of modern society. People use the wide range of available social networking services to create relationships and communities, in which they share data with family, friends and co-workers. These social networking services works great for their intended purpose, connecting people, but a privacy issue arises when using such services. Even though you have the ownership of everything you share, you often grant the service provider the possibility to use your data as they please.

UbiShare was proposed as a solution of this problem. By using UbiShare, innovative sharing applications could provide smaller, private communities, such as rescue crews, a way of communicating while in action. The idea was to use private social networks to enhance disaster management by providing an easy and intuitive way of sending messages and sharing data that would aid in decision making. This way, disaster management would be much easier to coordinate, but if the disaster wipes out communication infrastructure, the current version of UbiShare would not work. Consider the following scenario:

Scenario: Earthquake A severe earthquake has occurred and a rescue crew is searching for survivors in the ruins. The crew members are equipped with devices to receive messages and share environmental data which a coordinator can use to organize the search. These devices use UbiShare to share the data within the disaster management community. The only problem is that the earthquake has wiped out network and communication infrastructure. UbiShare cannot access the online storage service used to synchronize the mobile devices.

There is a need for a system that can share data within a community when internet infrastructure is absent; a system that can aid in scenarios like the one above. Such system could also be used in less serious situations. Consider the following scenario:

Scenario: Transient Meeting A group of people gathers for a transient meeting. In the meeting a large set of photos have to be shared among the attendants. There are no wireless network available and using the mobile network would take too much time.

In this scenario, a system that could synchronize data among mobile devices using an infrastructure-less wireless network would be beneficial. Such system would also make the data sharing more natural by providing a communication channel between devices in close proximity, as discussed in [4]. This is a step towards Mark Weiser's vision of ubiquitous computing [5].

1.3 Thesis Goal

The goal of this thesis is to develop a proof-of-concept system that lets you synchronize social data in a group of mobile devices using infrastructure-less wireless networks. Since UbiShare already has the underlying functionality; the possibility to create social networks and manipulate their data through APIs, it is natural to extend UbiShare with a synchronization feature that uses infrastructure-less wireless networks. The purpose of this extension is to demonstrate how the coordination of disaster management can benefit from having a system that can provide a communication channel between members of a rescue crew in action.

1.4 Thesis Structure

The thesis has the following structure:

Chapter 2: Problem Analysis presents an analysis of the problem and the requirements of a proof-of-concept system that aims to solve it.

Chapter 3: State of the Art presents cutting edge communication technologies and the latest advances in mobile data synchronization.

Chapter 4: Proposed Solution presents the proposed solution to solve the problem.

Chapter 5: Development presents this thesis' contribution to solve the problem; the design and implementation of the proof-of-concept system.

Chapter 6: Evaluation presents the evaluation of the developed system.

Chapter 7: Conclusions and Further Work concludes the thesis and suggests enhancements that can be implemented into the proof-of-concept system to make it more suitable for disaster management.

Chapter 2

Problem Analysis

This chapter presents a deeper analysis of the problem at hand.

2.1 Requirements Specification

In order to form the requirements of the system, we have to consider the earthquake scenario. In this scenario the internet infrastructure has been wiped out, and to enable data sharing, the system must support data synchronization using infrastructure-less wireless networks. The rescue crew consists of several people, hence the system must support data synchronization among several devices. In this proof-of-concept system, we can assume that a rescue crew consists of 2-8 people. This will form a more measurable requirement.

When synchronizing data within a group people, a community, the system needs to have notions of different parts of a social network, such as people, communities, relationships and memberships. UbiShare already has these notions and can easily be extended with this infrastructure-less data synchronization system. The fact that UbiShare is developed for Android results in the requirement that the proof-of-concept system should be available for Android devices. In order to support as many Android devices as possible the system should work out-of-the-box on high-end, non-rooted Android devices.

Since the primary focus are disaster management groups, the system needs to have some requirements of range, throughput and battery usage, where range and battery usage

are most important. It is desirable to have as high values for these as possible, but since both range and throughput affect the battery usage, more realistic values needs to be set. For this proof-of-concept system a range of 15 meters inside (30 meters outside), a throughput of at least 1 Mbps and a battery lifetime greater than 6 hours are realistic requirements. The requirements mentioned so far is of high priority since they are essential to prove the use of this system.

Even though the system could easily synchronize several communities and their belonging data simultaneously, a more realistic scenario would be that only members of a specific community would gather to share data. This forms a new requirement; the system should let the user choose which community is to be synchronized, or create a new one if none of the existing communities are satisfactory. When adding this feature the system should also only allow members of the selected community to participate in the peer-to-peer synchronization. These requirements are of medium priority.

Another requirement is that the system should be easy and quick to set up. If you are a member of a disaster management crew or you just want to quickly share a document to the members of a meeting, you wouldn't want the system to take several minutes to set up. A realistic maximum initiation time of this system is 30 seconds. This requirement is of low priority since it does not affect the purpose of this proof-of-concept system.

Since this system is just a proof-of-concept, non-functional requirements, like usability and security, are not listed. Table 2.1 sums up the requirements of the system.

ID	**Requirement**	**Dependency**	**Priority**
FR1	The system should be available for high-end, non-rooted Android devices		High
FR2	The system should be an extension of UbiShare	FR1	High
FR3	The system should support data synchronization using a infrastructure-less wireless network		High
FR4	The system should support data synchronization among 2-8 devices	FR3	High
FR5	The system should have a range of at least 15 meters inside and 30 meters outside		High

Continued on next page

Table 2.1 – continued from previous page

ID	Requirement	Dependency	Priority
FR6	The system should have a throughput of at least 1 Mbps		High
FR7	The system should have a battery lifetime of at least 6 hours		High
FR8	The system should let the user choose which community that is to be synchronized		Medium
FR9	The system should allow only the members of the chosen community to participate in the synchronization	FR8	Medium
FR10	The system should have a maximum initiation time of 30 seconds		Low

TABLE 2.1: Table of System Requirements

2.2 Problem Breakdown

The problem can be broken down into four layers, as shown in Figure 2.1. The three bottom layers, local data storage, communication technologies and data synchronization, are a part of UbiShare. UbiShare is a platform for mobile applications that require data synchronization among mobile devices. Through a set of APIs, applications can use the UbiShare platform to create social networks, and store and synchronize data within these social networks among mobile devices. UbiShare implements the OpenSocial standard, which specifies APIs of how applications can be built on top of social networks.

The bottom layer is the local data storage. UbiShare provides a local data storage where applications can store data that is to be synchronized. The local data storage acts as a local cache of the data, enabling the applications to access the data without having to request it from a remote service for each use. The problem introduced in this thesis is not concerned about the local data storage layer, as this layer was covered in [3], but the layer is included here to better explain which part of UbiShare this thesis relates to.

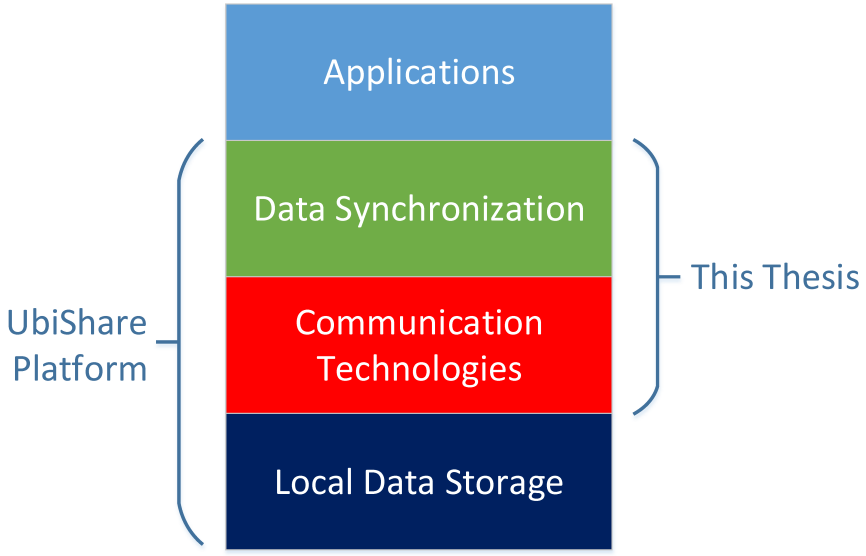

FIGURE 2.1: The problem broken down into three layers.

As mentioned, the system that is to be developed in this thesis should be an extension of UbiShare. Currently, UbiShare synchronizes the data with online storage services through wireless networks with fixed infrastructure; through wireless access points with an internet connection. It is at these layers, the communication technologies and data synchronization layer, this thesis has its focus.

The problem in the communication technologies layer is to find suitable technologies that do not rely on fixed infrastructure and meet the requirements of the system. Characteristics that affect the choice of communication technology are its discovery method, range, transfer rates and network protocol. Section 2.3 presents a deeper analysis of communication technologies.

Data synchronization is the next layer of the problem. The problem here is to find a data synchronization strategy that is compliant with the chosen communication technology and suitable for synchronizing data among mobile devices. Challenges with data synchronization among mobile devices are their mobility; in terms of sudden disconnects and network partitioning, and their reduced resources, such as battery lifetime and computational power. A deeper analysis of data synchronization is presented in Section 2.4.

The top layer is the applications using the system. The development of such applications are not covered in this thesis, but they are an essential part of the problem. UbiShare is

only a platform, and without applications built on top, the purpose of the system would be completely defeated. Applications built on top of UbiShare are responsible for the creation of social networks, and provides the data that is to be synchronized with other devices. They also specify which service to use when synchronizing the data. In order properly evaluate the proof-of-concept system developed in this thesis, such applications are needed.

2.3 Communication Technologies

The choice of communication technology depends on a variety of factors. Since the system to be developed is run on mobile devices, the mobility of the devices and their reduced resources are key factors. The mobility of the devices can cause sudden disconnects and network partitioning, and with their reduced resources, such as battery lifetime and computational power, the communication technology needs to be lightweight and less power demanding. One of the most important requirements is that the chosen communication technology is supported on as many mobile devices as possible, so that the system can be used on a variety of devices.

Discovery method and connection setup are important factors when choosing communication technology. When a group of mobile devices are brought together to form a network, each device needs to know how to discover and connect to each other. This requires the communication technology to allow publishing of services and have a service discovery method, so that devices can filter them and choose the correct service. The connection setup are also important, since it defines the process of connecting the devices. The amount of manual labor is key here. It is desired that the connection setup is as automated, quick and intuitive as possible. This will enhance the usability of the system.

The mobility of the devices may cause sudden disconnects and frequent network partitioning. A key factor is how the communication technologies handles these. If one or more of the mobile devices disconnects from the network, it is desired that the network is still operative, so that the other devices can continue synchronizing data within the partitioned network. Another factor to consider is how the communication technology handles reconnects. If a device loses the connection to the network, because it is out of

range, it is desired that the communication technology automatically reconnects when the device reenters the network area.

Security is another key factor when choosing communication technology. You wouldn't want intruders to be able to connect to the network or sniff the data traffic between the devices. This requires the communication technology to have a secure connection setup and an encrypted data transfer protocol. Security features are usually tradeoffs. A secure connection setup would probably require more manual labor, and encrypted data traffic uses more computational power and has a negative effect on battery lifetime.

Since the system developed in this thesis aims to connect members of a rescue crew, it needs a communication technology with a great range. If the rescue crew are searching for survivors after a severe earthquake, it is desired that the members of the crew could be as far apart as possible to broaden the search. Again, there are tradeoffs with respect to battery lifetime. The best thing would probably be to support different communication technologies with different range and battery usage. This way, different communication technologies could be used in different scenarios. When battery lifetime is more important than range, a less power demanding, lower ranged communication technology could be used.

Another factor to consider is the data transfer rates of the communication technology. The data shared in this system is ranging from simple text messages to photos, and maybe even videos. In order to transfer such data in a satisfactory manner, the communication technology has to have great transfer rates. As with range, there are tradeoffs regarding battery lifetime. An optimal communication technology combines great range and transfer rates with long battery life.

When choosing a communication technology, the supported transfer protocols are also a key factor. If none of the supported protocols have reliable data transmission, a complex architecture might be needed to ensure that the data is delivered to its destination. Most of today's communication technologies support reliable data transmission, hence the more interesting question is whether they support broadcasting and multicasting. Broadcasting refers to transmitting data packets to all the devices on the network, simultaneously. This can be used to reduce the number of packets needed to transfer data to multiple devices. The same goes for multicasting, but when sending a multicast packet,

only devices in the multicast group can receive it. Some data synchronization strategies require either broadcasting or multicasting. Section 2.4 describes such strategies in further detail.

2.4 Data Synchronization

The idea behind data synchronization is to establish *consistency* among two or more data sources. If one of them changes, the same change needs to be performed on the other sources in order to keep them consistent. In this thesis, data synchronization will be used to keep the data on multiple mobile devices consistent [6]. These mobile devices form a synchronization group, and acts as peers in a peer-to-peer network. If a mobile device makes a change to its data, it needs to notify the other devices of the changes that have been made in order to keep them in sync. This can be done by sending a change notification. The following section about synchronization strategies presents different ways of doing this.

2.4.1 Strategies

This section presents a couple of synchronization strategies that can be used in a peer-to-peer system.

2.4.1.1 Peer-to-Peer

In the peer-to-peer strategy, each peer is responsible for notifying all the other peers when changes occur. If a peer adds, modifies or deletes some data, it has to send a notification to all the other peers in the group in order to keep them up to date. This can be done by either sending the notification explicitly to each peer (shown in Figure 2.2), or by broadcasting it. The former requires that each peer is aware of all the other peers in the synchronization group and how to reach each one of them. When a peer joins a synchronization group, it needs a way to introduce itself to the other peers. In a peer-to-peer group without a centralized “peer lookup” service, it is an awkward process to find the other peers. One way would be to scan IP ranges hoping for a strike of luck, but this is like placing a group of deaf people in a completely dark room, instructed to

find each other without making a sound. This is why the preferred method, when using this strategy, is to broadcast the notifications.

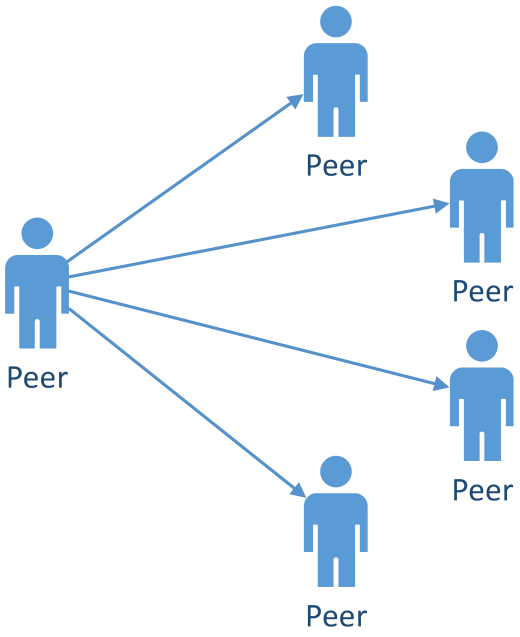

FIGURE 2.2: When using the unicast method and a peer has made a change, it needs to send a notification to each of the peers in the network.

When broadcasting, notifications are sent to all the recipients, simultaneously. This way the peers does not need to know of each other. When a peer needs to notify the others of a change, the notification can be broadcasted to all the peers that are listening for broadcast messages (shown in Figure 2.3). Peers are not required to introduce themselves when joining a synchronization group. It is also safe to assume that broadcasting is less resource demanding, since a peer only needs to send the notification once instead of one for each peer in the group. There are, however, some disadvantages of using this method. First, since everyone can receive the broadcasted notifications, there is a need for enhanced security. Encrypting the messages can solve this problem, but this requires a safe way to distribute the encryption keys. Second, broadcasting is carried over an unreliable network protocol, e.g. the *User Datagram Protocol* (UDP). There is no way for a peer broadcasting a notification to know if every recipient received the notification without implementing a more complex architecture. There are several ways of broadcasting reliably, but they all result in more complex architectures.

A drawback of the peer-to-peer strategy is that there is no master copy of the data that can be used to solve conflicts. Conflicts occur when multiple peers modifies the

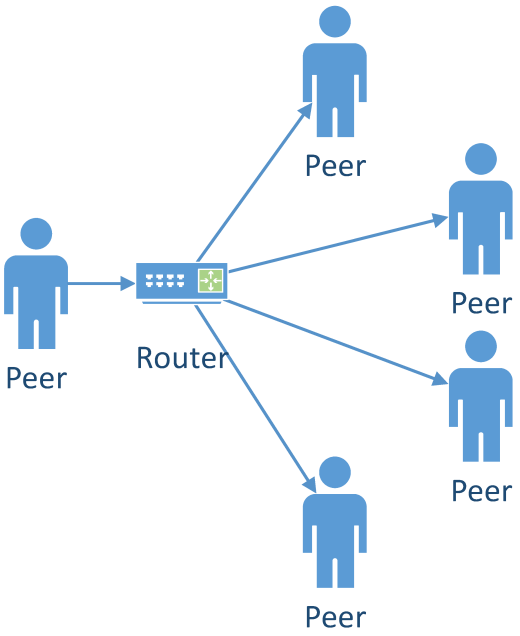

FIGURE 2.3: Using the broadcast method, a peer that has made a change only needs to send the notification once, and the router will deliver it to all the peers in the network.

same data before they are able to notify each other. In best case scenario, the only consequence is that the most recent change overwrites the conflicting change. The data on the peers are still consistent, but if peers receive notifications of the same data in different orders, they will become inconsistent and from there on out the peers will not be in sync. Either way, none of the scenarios are desirable, hence a more complex architecture are needed to handle conflicts.

2.4.1.2 Client-Server

When using a client-server strategy, one of the peers takes on the role as a server, while the rest of the peers are clients. The server peer has the master copy of the data and the client peers sends data changes to the server peer only. Depending on which technology is used, the server peer could either *push* changes to the client peers, or the client peers could *pull* the changes from the server peer. The client-server strategy is inspired by the ward model described in [7].

Server Push When the server peer receives a change notification from a client peer, or makes changes to some data itself, it *pushes* a change notification to the rest of the

client peers (Figure 2.4). This technology requires the server peer to have an overview of all the client peers. A simple client register where client peers are added when they first connect to the server, will satisfy this requirement.

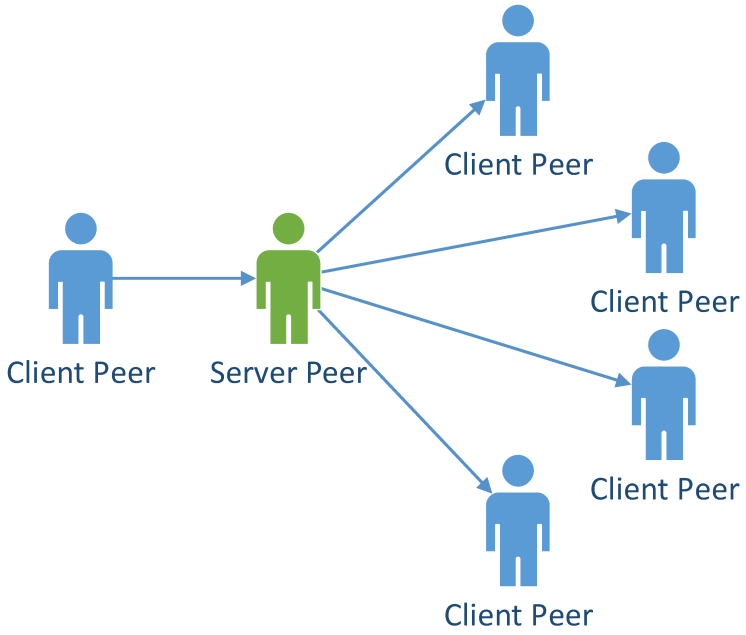

FIGURE 2.4: When using the server push method, a peer that has made a change sends the notification to the server, and the server will send the notification to the other peers.

Client Pull Using this method, clients periodically *pulls* changes from the server peer (Figure 2.5). This way, the server peer does not need to have an overview of the client peers. It is merely a servant handling requests from the clients, requiring a less complex architecture to function satisfactory.

When comparing the two technologies, we see that the client pull technology has more overhead when it comes to data traffic. The client peers might continually request changes from the server peer when there are no changes made. This would create unnecessary data traffic, hence client pull is more resource demanding.

Conflict solving is less complex when having a master copy of the data. Conflicts will still occur if multiple peers modifies the same data at the same time, but the server peer who has the master copy could easily detect conflicts and respond to them accordingly. This would prevent inconsistency among the peers.

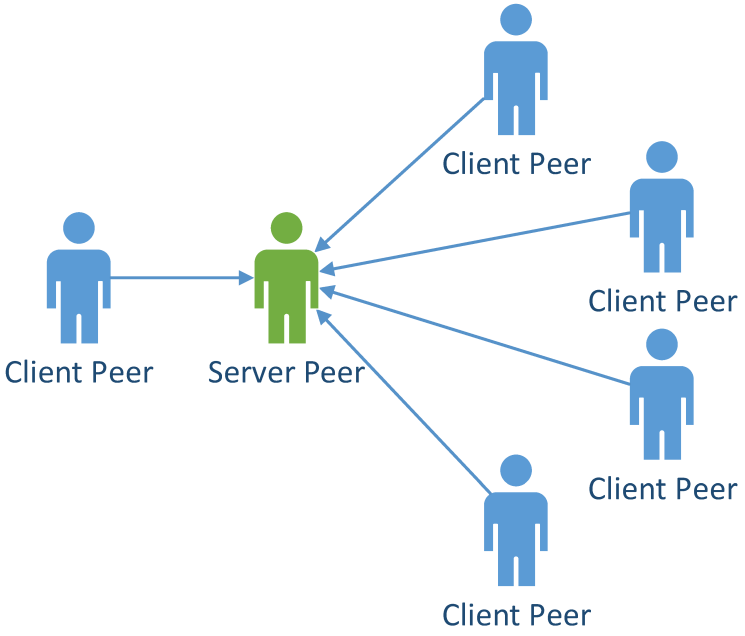

FIGURE 2.5: When using the client pull method, a peer that has made a change sends the notification to the server, and the other peers will receive the change by continually requesting changes from the server.

The client-server strategy would also be less complex than the peer-to-peer strategy in terms of communication. Since the communication of the client-server strategy can be carried over a reliable network protocol, e.g. the *Transmission Control Protocol* (TCP), there is no need for a complex communication architecture. However, the client-server strategy is more resource demanding than the peer-to-peer strategy, since it requires more network traffic. Table 2.2 presents a comparison of the strategies.

Strategy	**Method**	**Pros**	**Cons**
Peer-to-Peer	Unicast	No server peer with heavy load	Requires knowledge of all the peers in the group, and complex architecture to ensure reliable data transfer
	Broadcast/Multicast	No server peer with heavy load, and least network traffic	Requires complex architecture to ensure reliable data transfer

Continued on next page

Table 2.2 – continued from previous page

Strategy	**Method**	**Pros**	**Cons**
Client-Server	Client Pull	Less complex server architecture than Server Push	Unnecessary requests for changes, when no changes have been made
	Server Push	Less power demanding than Client Pull	Requires knowledge of all the client peers, more complex server architecture

TABLE 2.2: Comparison of Synchronization Strategies

2.4.2 Synchronization among Mobile Devices

The mobility of the devices introduces some challenges when synchronizing data among mobile devices. Sudden disconnects and frequent network partitioning needs to be addressed by the synchronization logic. If a device loses its connection to the network and changes are made during this period, it is essential that the disconnected device receives these changes when it rejoins the synchronization group. This can prove quite challenging when using the peer-to-peer strategy, since none of the devices have a master copy of the data. With the client-server strategy, the rejoining device could just send a request to the server, requesting the changes made during the time the device was disconnected, given that the server has such functionality.

Network partitioning can also cause problems during the synchronization. If a device disconnects from the synchronization group, the data synchronization should continue among the still active devices as if nothing happened. Handling such disconnects should be pretty straight forward for non-server devices, but if a server device disconnects from the synchronization group, there is a need for a *server migration* functionality. This would allow the still active devices to continue synchronizing data by appointing a new server peer.

Chapter 3

State of the Art

This chapter presents the state-of-the-art communication technologies and mobile data synchronization systems. Both communication technologies and the data synchronization systems are compared with the requirements of the proof-of-concept system that is to be developed in this thesis.

3.1 Communication Technologies

This section presents state-of-the-art communication technologies in mobile devices.

3.1.1 Bluetooth

Bluetooth is a short-range, peer-to-peer, communication technology intended for transferring data from fixed and mobile devices. It is a popular technology embedded in a wide range of devices ranging from mobile devices and computers to medical devices and home entertainment products. Bluetooth has for many years been used to transfer data between mobile phones. It has also been the preferred communication technology in wireless keyboards and mice and in hands-free headsets due to its low power consumption. Most of today's mobile phones are Bluetooth enabled, hence it is highly relevant for the system that is to be developed in this thesis.

The Bluetooth technology uses a master-slave structure. This means that when two Bluetooth enabled devices connects to each other, also known as pairing, one of them

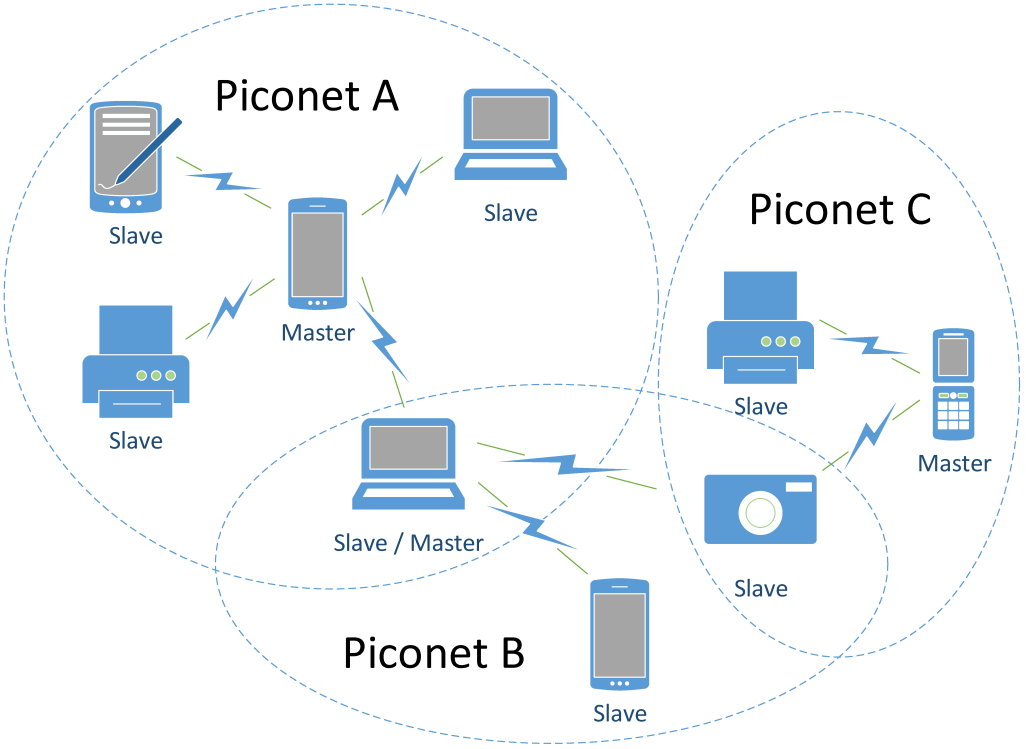

FIGURE 3.1: A scatternet can consist of several piconets.

takes the role of a master. One master may communicate with up to seven slave devices simultaneously forming an ad hoc network known as a piconet. A device can belong to several piconets simultaneously, forming scatternets [8]. This is illustrated in Figure 3.1.

The fact that Bluetooth is a low power consuming communication technology results in a limited range. It was meant to be an easy-to-set-up communication channel used to transfer data between devices in close proximity, hence there is no need for a great range. However, when providing a communication channel between members of a rescue crew that might be far apart, Bluetooth might fall short.

The transfer rates and range of Bluetooth is dependent on which version is implemented and the class of the Bluetooth radio. Table 3.1 lists the different version and transfer rates, while Table 3.2 lists the different radio classes and their approximated range.

Version	**Maximum Transfer Rate**
Version 1.2	1 Mbps
Version 2.0 + EDR (Enhanced Data Rate)	3 Mbps
Version 3.0 + HS (High Speed)	24 Mbps
Version 4.0 + LE (Low Energy)	1 Mbps

TABLE 3.1: Table of Bluetooth Versions

Radio Class	**Range**
Class 1	~100 m
Class 2	~10 m
Class 3	~1 m

TABLE 3.2: Table of Bluetooth Radio Classes

The classical Bluetooth versions, 1.2 and 2.0, have low transfer rates. However, *Bluetooth Version 3.0 + HS* (High Speed) introduced a new architecture where the Bluetooth channel was used for negotiation and establishment, while the high speed data traffic was carried over a 802.11 link (Wi-Fi). This resulted in higher data rates and throughput (up to 24 Mbps). The range, however, was unaffected by the new architecture. This is due to the fact that, even though the data transfer is carried over a longer ranged 802.11 link, the classic Bluetooth link is still needed to maintain the connection between devices [9].

Version 4.0 of Bluetooth did not make any changes regarding the transfer rates, hence it uses the same technology as Version 3.0 + HS. However, it introduced a new technology called *Bluetooth low energy*. This technology aims to provide the same range, but with a considerably reduced power consumption [10]. *Bluetooth Version 4.0 + LE* (Low Energy) has a transfer rate of only 1 Mbps, but targets devices where low power consumption is essential.

In mobile devices, the Class 2 radio is the most common [8]. This means that most of the mobile devices only have a range of approximately 10 meters. However, newer mobile phones have started using the Class 1 radio, which means ranges up to 100 meters. In order to reach such ranges, it is required that a Class 1 radio is present in all the devices

in the *piconet*. Chances are that only a few, if any, of the devices have the Class 1 radio, and the range is limited by the shorter ranged devices.

3.1.2 Wi-Fi Direct

Wi-Fi Direct is a standard that allows Wi-Fi devices to connect and transfer data to each other without the need for a wireless access point (WAP). It uses software to utilize the 802.11 radio on Wi-Fi enabled devices to act as a WAP with a limited set of services. Wi-Fi Direct-certified devices can connect one-to-one or one-to-many. In one-to-many connections one of the devices is chosen to be the group owner. The group owner acts as a WAP and all other devices transfers data through this device. This is illustrated in Figure 3.2.

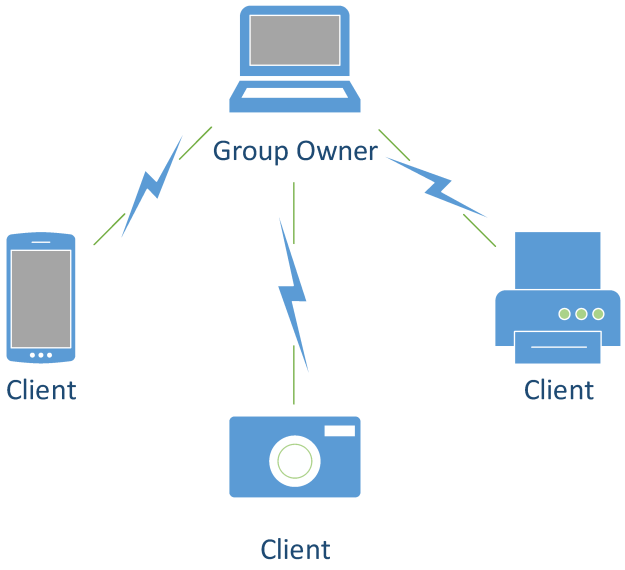

FIGURE 3.2: One of the devices in a Wi-Fi Direct network has the role as Group Owner.

Wi-Fi Direct has greater transfer rates and range than Bluetooth, but also more power consuming. The maximum number of devices in a Wi-Fi Direct network is expected to be smaller than the number supported by standalone access points [11]. This is due to the fact that Wi-Fi Direct is run on mobile devices with less powerful hardware.

By using the 802.11 radio for all its operations, Wi-Fi Direct supports typical Wi-Fi speeds and ranges, with data transfer rates up to 250 Mbps and ranges up to 200 meters [11]. Speeds and ranges are, however, dependent on the hardware used. It is safe to

assume that today's mobile devices offer slower speeds and shorter ranges than the maximum values of Wi-Fi, due to the fact that they use less power consuming hardware components. However, mobile devices are rapidly evolving, hence transfer rates and ranges of Wi-Fi Direct will likely increase as more powerful hardware is embedded in the devices.

3.1.3 Near Field Communication (NFC)

Near Field Communication is a short-range communications protocol that has evolved from *Radio-Frequency Identification* (RFID). RFID is a wireless data transfer protocol that uses electromagnetic fields to read tags attached to objects [12]. These tags contain electronically stored information. In opposition to RFID, NFC is a two-way communication protocol. This means that NFC can be used to exchange data between two NFC enabled devices, in addition to read tags [13].

NFC is a short range communication technology with low transfer rates. The strength of NFC lies in its low power consumption and ease of use. NFC does, in opposition to Bluetooth and Wi-Fi Direct, not require the process of *pairing* the devices. Due to NFC's transfer rates and range, it is not suitable for use as the primary communication channel of the proof-of-concept system that is to be developed in this thesis. However, due to the fact that NFC is very light weight, it can be used to make the process of initiating a Bluetooth or Wi-Fi Direct connection less labor intensive. Using NFC, you could just hold two devices in close proximity and with a push of a button the devices would exchange information to automatically initiate a Bluetooth or Wi-Fi Direct connection.

NFC has a maximum range of approximately 20 cm and a maximum data transfer speed of 0.424 Mbps [13]. It is estimated that 53 per cent of the mobile phones will be NFC-enabled by 2015 [14]. Table 3.3 presents a comparison of the communication technologies.

Technology	**Topology**	**Range**	**Transfer Rate**
Bluetooth (v3.1+)	One-to-One, One-to-Many	10 m (Class 2 Radio)*, 100 m (Class 1 Radio)	24 Mbps
Wi-Fi Direct	One-to-One, One-to-Many	Up to 200 m	Up to 250 Mbps
NFC	One-to-One*	20 cm*	0.424 Mbps*
* Does not satisfy the requirements of the proof-of-concept system.			

TABLE 3.3: Comparison of Communication Technologies

3.2 Data Synchronization

This section presents state-of-the-art data synchronization systems in mobile environments.

3.2.1 Couchbase Server

Couchbase Server is a distributed NoSQL database management system (DBMS) [15]. It uses a shared-nothing (SN) architecture where each node is independent and self-sufficient [16]. This results in great scalability. Nodes can be linked together in order to create several copies of the same data. If the connection is broken between linked nodes, each node can operate independently of each other due to the SN architecture, and when the connection is back up, the nodes get synchronized to make the data consistent among them.

The interesting thing about Couchbase Server, is that it has a version for mobile devices that support peer-to-peer synchronization of databases – Couchbase Mobile. Couchbase Mobile can be set up to accept connections, enabling other devices to connect and synchronize their database using peer-to-peer. This can be used to share data among mobile devices that lack an internet connection. Figure 3.3 illustrates this scenario.

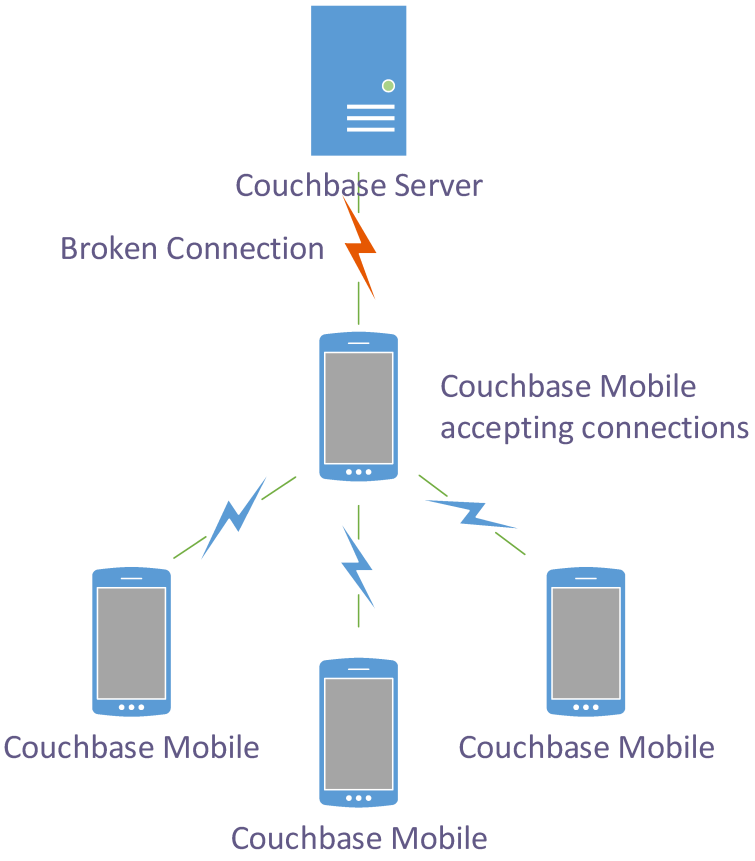

FIGURE 3.3: If the connection to the online master database is broken, a mobile device can be set up to accept connections, so that mobile devices in close proximity can continue to work on the same data.

3.2.2 BitTorrent Sync

BitTorrent Sync provides peer-to-peer synchronization of data among multiple devices. It uses the popular BitTorrent protocol with enhanced security to synchronize data within the local network or across the web. Currently, BitTorrent Sync is in the alpha stage and no mobile platforms are supported yet. However, it is stated that BitTorrent Sync is to support most of the popular mobile platforms in the future [17]. When support for mobile platforms is added, it is expected that BitTorrent Sync can be used to synchronize data among a group of mobile devices using the BitTorrent protocol.

3.2.3 JXTA

JXTA (Juxtapose) is a set of protocols that can be used to establish a peer-to-peer connection among devices. The purpose of JXTA is to provide a peer-to-peer communication layer that is independent of which communication technology is used. This way, JXTA can be used in many different scenarios and on many different devices. The implementation of JXTA is available in several programming languages, such as Java and C/C++ [18]. JXTA is not a standalone application, but rather a protocol that can be implemented into systems that need peer-to-peer group communication.

The peers can be divided into two categories; *edge peers* and *super-peers* [18]. Edge peers are usually peers run on devices with reduced resources, such as mobile devices, and have low responsibility within the peer-to-peer network. The super-peers can be further divided into *rendezvous* and *relay* peers. Rendezvous peers are responsible for coordinating the peers within the peer-to-peer network, and have stricter network, storage, memory and computation power requirements. The relay peers are used to allow peers that resides behind firewalls or NAT systems to take part in the peer-to-peer network. Any of the JXTA peers can be a rendezvous or relay peer as long as they satisfies the resource requirements.

FIGURE 3.4: JXTA consists of three layers.

JXTA consists of three layers (Figure 3.4); the platform, services and applications layer [19]. The platform layer is the base layer and consists of the implementation of essential peer-to-peer functionality. Ideally, JXTA peers implement all the functionality defined by the platform layer, but they are not required to [19]. The services layer consists of functionality that is not necessarily required in order for the peer-to-peer network to operate, but might be useful. File sharing is an example of such service. Applications built on top of JXTA defines the applications layer. These applications use the functionality defined by the platform and services layer to perform peer-to-peer operations.

XML is used to issue and exchange messages between JXTA peers. The fact that XML is human readable, and hence more verbose than e.g. binary data documents, may cause performance issues. This is due to the fact that XML documents are bigger than binary data documents containing the same information. It is suggested to use data compression within the XML documents to mitigate this issue [19].

3.2.4 AGAPE

AGAPE (Allocation and Group Aware Pervasive Environment) is a framework that uses context information, such as proximity and user attributes and preferences, to help form collaboration groups on mobile devices [20]. The purpose of AGAPE is to provide users with the same agenda (e.g. rescue crews) a way of communicating and sharing data in groups. The proximity of the mobile devices determines their visibility to others, and by using a predefined set of user attributes and preferences, AGAPE provides a set of services to arrange/dissolve and manage ad-hoc groups [20]. Consider the following scenario:

Scenario: Firefighters using AGAPE A group of firefighters are working in the same area. The captain starts the dynamic formation of a collaboration group, specifying that it is a group for firefighters. Other firefighters can join the newly created group since they are in close proximity and their user attributes states that they are firefighters. AGAPE assigns the captain a role of a coordinator since his user attributes states that he is of higher rank, while the other firefighters are assigned a generic "firefighter" role. The collaboration group is used to exchange messages and pictures that aid in decision making.

As the scenario shows, proximity and user attributes are used to form a collaboration group and to assign roles within the created group. AGAPE is a middleware framework using mobile ad-hoc networks (MANETs) as communication channel.

3.2.5 Summary

As a summary, each system is compared with the requirements of the proof-of-concept system that is to be developed in this thesis. Not all of the requirements are relevant in

this comparison, hence only a few is selected. The selected requirements are whether the systems support Android (FR1), data synchronization using infrastructure-less wireless networks (FR3) and data synchronization among 2-8 devices (FR4). Table 3.4 presents the comparison.

Requirement	**Couchbase**	**BitTorrent Sync**	**JXTA**	**AGAPE**
Android support	Yes	Not yet	Yes	Yes
Data synchronization using infrastructure-less wireless networks	Yes	Yes	Yes	Yes
Data synchronization among 2-8 devices	Yes	Yes	Yes	Yes

TABLE 3.4: Summary of State-of-the-Art Data Synchronization Systems

All the systems satisfy the requirements, except BitTorrent Sync which does support mobile devices yet. However, since the proof-of-concept system that is to be developed in this thesis is to be an extension of an existing system, embedding these systems might require a redesign of UbiShare, which is outside the scope of this thesis. Hence, they will only be used as inspiration when designing and implementing the proof-of-concept system.

Chapter 4

Proposed Solution

The peer-to-peer synchronization was going to be an extension of UbiShare. This meant that the target system was Android, and the synchronization strategy and communication technology had to be compatible with this operating system.

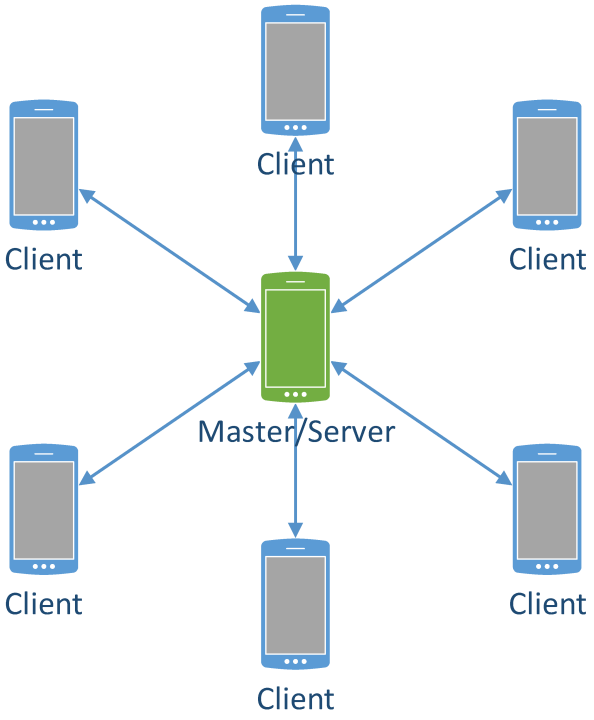

FIGURE 4.1: Physical view of the chosen strategy – Server Push. Clients send changes to the server, and the server pushes changes to the clients.

One of the most important requirements of the system, is its range. When providing a communication channel between members of a rescue crew, it is important that they can

be as far apart as possible to broaden the search of survivors. This made the choice of communication technology easy. Wi-Fi Direct has the greatest range, on paper, of the peer-to-peer communication technologies in today's mobile devices, and became available to Android devices in version 4.0 (API level 14). Android 4.0 was released 19 October 2011, which makes Wi-Fi Direct support for Android devices fairly new. Legacy devices will not be able to run this system if Wi-Fi Direct is used as communication technology. A choice was made to support both Wi-Fi Direct and Bluetooth, with Wi-Fi Direct as the main focus. Bluetooth is a much older technology, enabling legacy devices to use the system. The choice of communication technologies satisfies the requirements FR3-FR6, and hopefully also FR10.

When it comes to synchronization strategy, the initial thought was to use the peer-to-peer strategy with broadcasting. The reason was that this approach is the least resource demanding, since it requires the least amount of network traffic. After a failed experiment of sending broadcast messages over Wi-Fi Direct, it was decided that the client-server strategy would be a better fit. Not only does it result in a less complex architecture, it also seems a better fit for both Wi-Fi Direct and Bluetooth. Both technologies have a group owner or a master device that could also act as a server, since it is known to all the other devices in the network. The proposed physical architecture is shown in Figure 4.1. Chosen synchronization strategy satisfies the requirements FR3 and FR4.

Consider the earthquake scenario. If UbiShare had such infrastructure-less synchronization feature, the rescue crew members in close proximity could continue to use UbiShare even after the internet infrastructure is wiped out by the earthquake. By placing the crew member equipped with the server device in the middle, the system could potentially cover an area of $\pi * (200m)^2 \approx 125000m^2$, considering Wi-Fi Directs theoretical maximum range. It is, however, highly unlikely that today's mobile devices can achieve such coverage, but future devices might.

Chapter 5

Development

This chapter presents the thesis' contribution to solve the problem; the design and implementation of a proof-of-concept, infrastructure-less data synchronization system.

5.1 System Architecture

This section presents the architecture of the proof-of-concept system.

5.1.1 Approach

The development of the proof-of-concept system was quite experimental. Using unfamiliar technologies, the approach was to make the first draft of the architecture as minimalistic as possible. The main focus was to get core components up and running, and then to gradually extend these with the required functionality. There was also a focus on making the components as generic as possible, since multiple communication technologies were to be supported. The system was designed to have the same behavior independent of which communication technology is used. A class diagram of the initial architecture is shown in Figure 5.1.

UbiShare uses sync adapters to synchronize the data with an online storage service [3]. These sync adapters are run by the Android operation system, and only if an internet connection is present. The whole purpose of the proof-of-concept system was to be able to synchronize data without an internet connection, but it was still desired to have

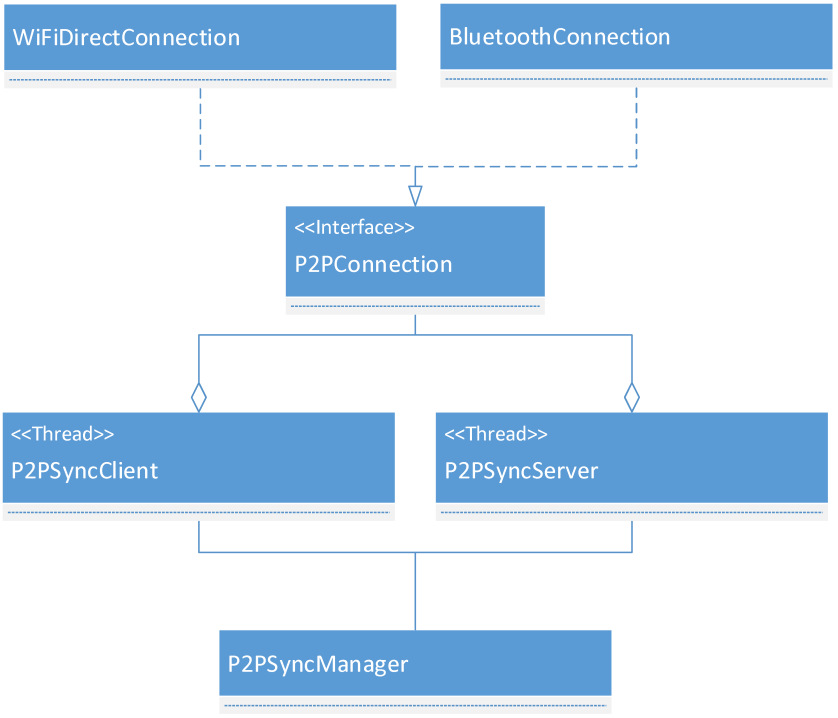

FIGURE 5.1: The initial architecture of the system contains only core components of the system.

a similar behavior as of using sync adapters. This way, UbiShare will have the same behavior independent on which synchronization technology is used. A solution was to create two internal threads; `P2PSyncServer` and `P2PSyncClient`. These threads were to mimic the behavior of sync adapters.

In order to encapsulate which communication technology is used, a generic `P2PConnection` interface was proposed. This interface forces the functionality of the different communication technologies to be the same, hence the synchronization services would have the same behavior independent of communication technology.

The `P2PSyncManager` class is the API of the peer-to-peer synchronization. This class has the functionality to find and connect to other peers, and starts the appropriate synchronization service when a group is formed. `P2PSyncManager` is the glue that holds the different components of the peer-to-peer synchronization system together.

5.1.2 Resulting Architecture

The initial architecture proved to be a bit too minimalistic, and a few components was added in order to get the system to work with minimal functionality. One of these components was a connection listener – `P2PConnectionListener`. The synchronization server needed some way of accepting incoming peer-to-peer connections from client peers, and since the chosen synchronization strategy was *Server Push*, the client peers also required a connection listener to accept connections from the server in order to receive change notifications. A `P2PConnectionListener` is required at the initialization of both `P2PSyncServer` and `P2PSyncClient`. Again, in order to support different communication technologies, the `P2PConnectionListener` needed to be generic and encapsulate the underlying technology. A class diagram of the classes related to network communication are shown in Figure A.2, Appendix A.

Another problem that arose when using the initial architecture, was the lifetime of the two threads `P2PSyncServer` and `P2PSyncClient`. If these threads are spawned in any of the applications activities, they would be terminated when the operating system decided that the activity was inactive and could be destroyed. In order to get the synchronization to work with this architecture, the activity that spawned the threads needed to be visible at all times. This meant that the device could not be used to anything other than running the synchronization. Without the possibility to close the synchronization application and use other applications to fill the database with data, there would not be any data to synchronize. This defeated the purpose of the system completely. The solution to the problem was to create two services; `P2PSyncServerService` and `P2PSyncClientService`. These services would spawn the synchronization threads and run in the background, outliving the applications user interface. The new architecture is presented in Figure A.4.

`P2PSyncManager` handles the initiation of peer discovery and connection. These operations were quite different for Wi-Fi Direct and Bluetooth, and having the implementation of both communication technologies cramped up in a single class resulted in an unnecessary complex architecture. In the final architecture, the `P2PSyncManager` class is made abstract, providing a clean interface independent on which communication technology is used. `WiFiDirectSyncManager` and `BluetoothSyncManager` contain the

specific implementations of the different communication technologies. This is illustrated in Figure A.1.

Since the system that was to be developed was only a proof-of-concept, and was to contain only a subset of the functionality of a complete system, there was a focus on making the architecture as modular as possible. This way, the proof-of-concept system could easily be extended with new functionality and formed into a complete version. If a future communication technology proved to be better suited than the current ones, this technology could be added without affecting the behavior of system, due to the modularity of the system. Class diagrams of the final architecture are presented in Appendix A.

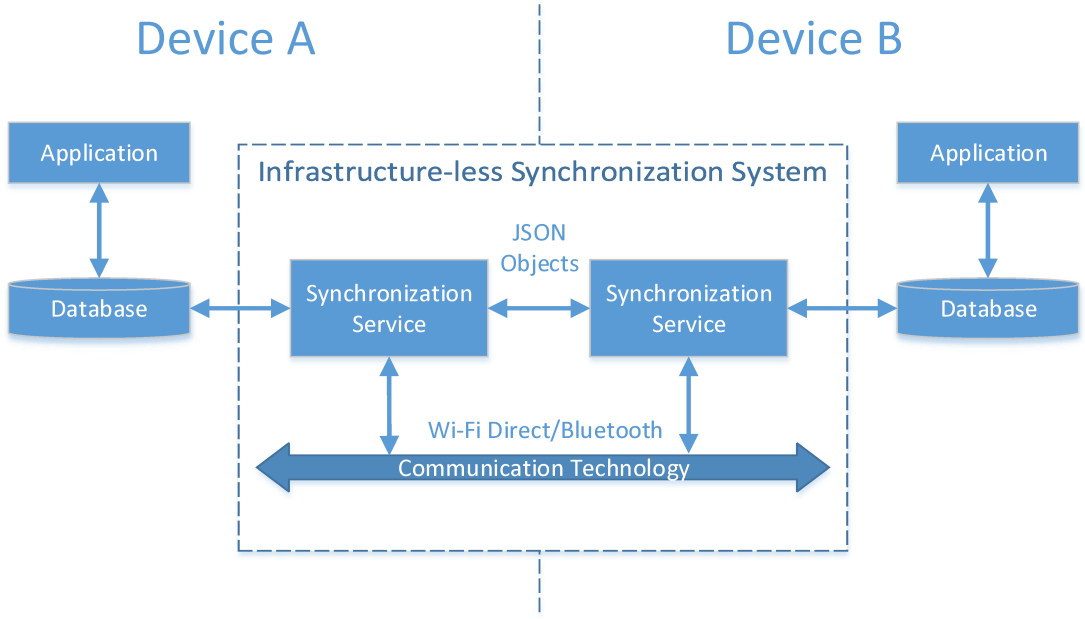

FIGURE 5.2: The components of the infrastructure-less synchronization system.

Figure 5.2 shows the interaction of the different components of the system. Application built on top of the synchronization platform reads and writes data to the local database. When the synchronization service detects that data in the local database has been changed, it triggers a synchronization. The new or modified database entries are then converted into entity objects. These entity objects contains properties that reflect the columns in the database tables. Then, the synchronization service serializes the entity objects into the JSON-format and sends them to another device, in our case the server device. The server device then de-serializes the JSON-formatted objects back into entity

objects and inserts them into the local database. The application on the server device can now read and manipulate the new changes.

5.2 Implementation

This section presents the implementation of the proof-of-concept system.

5.2.1 P2PSyncManager

The `P2PSyncManager` is the single point of entry of the peer-to-peer synchronization system. All operations of the system, such as discovering and connecting to a peer, are accessible through this class. These operations are somewhat different for Bluetooth and Wi-Fi Direct, and having both implementations within the same class resulted in an unnecessary complex implementation. The solution was to make `P2PSyncManager` abstract, and separate the specific implementations of the two communication technologies in two subclasses – `BluetoothSyncManager` and `WiFiDirectSyncManager`. In order to encapsulate which of the two implementations were used, and to make a cleaner API, the factory method pattern was used. This way, programmers can specify the behavior of the `P2PSyncManager` through parameters, and do not need to understand the potentially complex class hierarchy. The only way to obtain an instance of `P2PSyncManager` is to call the `getSyncManager(Context, IP2PChangeListener, ConnectionType)` method (Listing 5.1). To initialize a `P2PSyncManager` using Wi-Fi Direct, this method should be called with the parameter `ConnectionType.WIFI_DIRECT`.

Operations using the Bluetooth or Wi-Fi Direct framework are all asynchronous. Android reports the result of these operations by broadcasting intents. These broadcasts are received by `BluetoothBroadcastReceiver` and `WiFiDirectBroadcastReceiver`, who notifies the `P2PSyncManager` of changes related to the communication technologies. In order to receive notifications of the different events of the peer-to-peer synchronization system, such as the discovery of peers or the change of connection status, a `IP2PChangeListener` needs to be supplied to the `P2PSyncManager`. This listener will be used to report the results of the asynchronous operations.

```
1  public static P2PSyncManager getSyncManager(
2          Context context,
3          IP2PChangeListener listener,
4          ConnectionType connectionType) {
5      P2PSyncManager syncManager = null;
6
7      if (connectionType == ConnectionType.WIFI_DIRECT)
8          syncManager = new WiFiDirectSyncManager(context, listener);
9      else if (connectionType == ConnectionType.BLUETOOTH)
10         syncManager = new BluetoothSyncManager(context, listener);
11     else
12         throw new IllegalArgumentException(
13             "Unknown connection type: " + connectionType);
14
15     syncManager.initialize();
16
17     return syncManager;
18 }
```

Listing 5.1: `P2PSyncManager` – Factory Method Pattern

After a successful connection to another peer, the `P2PSyncManager` starts either the `P2PSyncServerService` or the `P2PSyncClientService`. Which of the services is started depends on the synchronization role of the peer. If the peer is the group owner in a Wi-Fi Direct network or the master in a Bluetooth piconet, the `P2PSyncServerService` is started. Otherwise, the peer is a client peer or a slave, hence the `P2PSyncClientService` is started. Section 5.2.2 describes the implementation of these services in further detail. The sequence of discovering and connecting to peers is presented in Figure A.5, Appendix A.

5.2.2 Synchronization Services

The synchronization services are the main components of the system. These services handles the actual data synchronization. `P2PSyncServerService` is run on the server peer, and `P2PSyncClientService` is run on client peers. The services are local, which means that they run in the same process alongside the rest of the system. Since they are

local, the interaction between the application and the services are greatly simplified. By using the `LocalServiceBinder`, you can obtain the actual instance of the service, and make direct calls to its methods. Both services implement the `ISyncService` interface. This interface specifies methods to stop the services.

```
1  public void onReceive(Context context, Intent intent) {
2      String action = intent.getAction();
3
4      if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION
5              .equals(action)) {
6          int state = intent.getIntExtra(
7              WifiP2pManager.EXTRA_WIFI_STATE, -1);
8
9          if (state == WifiP2pManager.WIFI_P2P_STATE_DISABLED)
10             stopSyncService();
11     } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION
12             .equals(action)) {
13         NetworkInfo networkInfo = (NetworkInfo)
14             intent.getParcelableExtra(
15                 WifiP2pManager.EXTRA_NETWORK_INFO);
16
17         if (!networkInfo.isConnected())
18             stopSyncService();
19     }
20 }
```

LISTING 5.2: `WiFiDirectServiceBroadcastReceiver` – When Wi-Fi Direct is turned off or the connection to the synchronization group is lost, the synchronization service is stopped.

In order to receive notifications of changes from the Bluetooth and Wi-Fi Direct framework, the services had to implement a couple of broadcast receivers. `BluetoothServiceBroadcastReceiver` and `WiFiDirectServiceBroadcastReceiver` are used to receive broadcasts from the Bluetooth and Wi-Fi Direct framework, respectively. They both are subclasses of `ServiceBroadcastReceiver`, which provides a clean abstraction of the two specific implementations. When a broadcast receiver gets notified

that the peer-to-peer network has dissolved, the synchronization service could be gracefully terminated (Listing 5.2). This way, there is no need for a manual termination when losing the connection to the synchronization group.

The services was implemented to have the same behavior independent of which communication technology is used. This was done by using generic classes, `P2PConnection` and `P2PConnectionListener`, when performing network related operations. In order to get this to work as intended, the `P2PSyncManager` has to initialize these generic classes with specific implementations, depending on which communication technology is used, and pass them as arguments when starting the synchronization services. When the `WiFiDirectSyncManager` starts a synchronization service (e.g., `P2PSyncClientService`), it initializes a `WiFiDirectConnection` and a `WiFiDirectConnectionListener`, and passes these as arguments to the service (Listing 5.3). If it was the `BluetoothSyncManager`, Bluetooth related implementations would be initialized. This way, the synchronization services have the same behavior independent of communication technology.

```
1  Intent intent = new Intent(mContext, P2PSyncClientService.class);
2
3  intent.putExtra(
4      P2PSyncClientService.EXTRA_CONNECTION,
5      new WiFiDirectConnection(groupOwnerAddress));
6  intent.putExtra(
7      P2PSyncClientService.EXTRA_LISTENER,
8      new WiFiDirectConnectionListener(
9          P2PConstants.WIFI_DIRECT_CLIENT_PORT));
10
11 mContext.startService(intent);
```

LISTING 5.3: Starting `P2PSyncClientService` using Wi-Fi Direct as communication technology.

Both services are designed to be able to run multiple synchronization threads. This way, the system could run synchronization threads using both Bluetooth and Wi-Fi Direct simultaneously, which can be useful if the synchronization group is to support both communication technologies. The synchronization threads are described in further detail in the following sections.

5.2.2.1 P2PSyncServer

The `P2PSyncServer` is the thread running the synchronization server. This thread accepts incoming connections from client peers and handles their notifications of change. The handling of change notifications consists of applying the changes locally on the server peer and pushing the notification to all the client peers, except the peer that originally sent the notification. This sequence is presented in Figure A.6, Appendix A.

In order to push notifications to client peers, the synchronization server needs to have a register of active client peers and how to connect to them. This is solved by requiring the client peers to send a *handshake* request when connecting to the server for the first time. When receiving a handshake request, the server can add the client peer to the register, and bring it up to speed by sending a response that contains all the data that has been synchronized so far. The sequence of the handshake request is presented in Figure A.7.

```
1  private void handleHandshake(Request request) throws Exception {
2      mHandshakeLock.lock(LockType.HANDSHAKE);
3
4      Peer peer = null;
5      if ((peer = getPeer(request.getUniqueId())) == null) {
6          peer = Peer.getPeer(request.getUniqueId(), mConnection);
7          addPeer(peer);
8      }
9      peer.setActive(true);
10
11     sendEntities(Entity.getAllEntities(
12         mContext.getContentResolver()));
13
14     mHandshakeLock.unlock(LockType.HANDSHAKE);
15 }
```

LISTING 5.4: When handling a handshake request, the `HandshakeLock` is used to ensure that changes will not happen during the handshake.

When synchronizing data among several devices, there is a chance that a client peer makes changes while the server is handling a handshake request. This would result in

the client peer, sending the handshake request, not receiving the changes made during the period of the handshake. To avoid this scenario, a `HandshakeLock` was implemented. The synchronization server uses this lock to ensure that the handling of change notifications are postponed to after currently active handshake requests are handled, or vice versa, that handshakes are handled after currently active change notifications (Listing 5.4).

The peer running the synchronization server, the server peer, is not only accepting connections and handling requests, it can also make changes to the data, just like the client peers. To check if any changes have been made, the `P2PSyncServer` uses an `UpdatePoller`. The `UpdatePoller` continuously checks the local database and sends a notification when it detects any changes. When the `P2PSyncServer` receives this notification it pushes the notification to all the available client peers.

5.2.2.2 P2PSyncClient

The `P2PSyncClient` is the thread that runs on the client peers. This thread sends local changes to the server and listens for change notifications sent by other peers. When connecting to the synchronization server for the first time, a handshake request is sent. After a successful handshake request, the client peer has all the data that has been synchronized before she joined the synchronization group, and can start manipulating it. An `UpdatePoller` is started to keep track of local changes. When the `P2PSyncClient` is notified of local changes, an *update* request, containing all the local changes, is sent to the server (Listing 5.5).

In order to receive push notifications from the server, `P2PSyncClinet` has a separate thread accepting connections from the server. This thread applies the received changes to the local database.

When a client peer has been offline for a while and decides to rejoin the synchronization group, there might be changes that have been made while the peer was offline. In order to get these changes, the rejoining peer needs to send a new handshake request. This lets the server know that the peer is active and ready to receive change notifications, and all the data of the synchronization group is sent as a response.

```
1  private void sendEntities(
2          Collection<Entity> entities) throws IOException {
3      if (entities.size() > 0) {
4          Request request = new Request(
5              mUniqueId, RequestType.UPDATE);
6          request.setUpdatedEntities(entities);
7
8          try {
9              mServerConnection.connect();
10             mServerConnection.send(request);
11         } finally {
12             mServerConnection.close();
13         }
14     }
15 }
```

Listing 5.5: When local changes have been made, an update request containing the changes is sent to the server.

Chapter 6

Evaluation

This chapter presents the evaluation of the proof-of-concept system.

6.1 Communication Technology

A series of tests were performed to evaluate the communication technology of the proof-of-concept system. The test setup consisted of the following devices:

- HTC One X (Android 4.1.1)
- Samsung Galaxy Note (Android 4.0.4)
- ASUS Transformer Pad TF700T (Android 4.2.1)

Since the system is only a platform, a test application was needed to populate the database with data to be synchronized. A simple chat application was developed for this purpose. This chat application has the functionality to create communities, add members and to post activities (messages) on community feeds. This data was then synchronized among the test devices.

Using Wi-Fi Direct as communication technology was the main focus. Even though the architecture of the system was designed to support any communication technology, only Wi-Fi Direct was completely implemented into the system. The plan was to also support Bluetooth in order to compare different communication technologies, but due

to time constraints, Bluetooth support was not fully implemented. Hence, all the tests was made using Wi-Fi Direct.

6.1.1 Discovery and Connection Initiation

The discovery and connection initiation test consisted of finding other devices and connecting to them. This process was benchmarked by measuring the time between discovery initiation to a successful connection was made. As it turned out, this process was quite awkward.

Consider the device discovery process. The Galaxy Note device has a dedicated Wi-Fi Direct setting, making it possible to run Wi-Fi Direct separately from the standard Wi-Fi. The HTC One X, however, did not have a separate Wi-Fi Direct setting, but Wi-Fi Direct was enabled when the standard Wi-Fi was turned on. When discovery was started on the Galaxy Note device, no devices was found, even though the HTC device had Wi-Fi turned on. Only after initiating a device discovery on the HTC device, would this device appear on the Galaxy Note. This is probably due to a setting suspending Wi-Fi Direct on the HTC device until a call to the Wi-Fi Direct API is triggered, in order to prevent battery drainage. At first, this did not seem like a problem, since it was possible to discover the Galaxy Note device from the HTC device. The HTC device could be used to both discover and connect to the Galaxy Note device, but the problems did not end here.

While using the HTC device to discover and connect to the Galaxy Note, only a few connection attempts succeeded. After an unsuccessful attempt, it seemed impossible to connect without turning Wi-Fi Direct off and on again between attempts. The most successful strategy was to use the Galaxy Note to connect to the HTC device, but this required that the discovery process was initiated on the HTC device first in order to make it discoverable by the Galaxy Note. Even this strategy had a disappointing ratio of unsuccessful connection attempts. The duration of the discovery and connection initiation process of successful attempts was approximately 10 seconds, which is within the maximum of 30 seconds, specified by requirement FR10. However, it is a labor intensive process to establish a connection between the devices, regardless of which of the devices are used. It seems to be the immatureness of Wi-Fi Direct in mobile devices that causes the problems.

6.1.2 Range

In order to measure maximum range, a connection was established between two devices and then the connected devices were gradually moved away from each other until the connected was broken. Indoors, the connection was lost when the devices was approximately 20 meters apart, with several thick walls in between. This seemed like a reasonable range, considering the thick walls, and satisfies the requirement of a minimum range of 15 meters (FR5). However, the range outdoors was disappointing. After several tests outdoors, the connection only seemed to sustain within a radius of 25 meters, which is not far, considering that there were no obstructions in between the devices. This would only cover an area of $\pi * (25m)^2 \approx 1950m^2$ (compared to the theoretical coverage of $125000m^2$), and does not satisfy the requirement of a minimum range of 30 meters outdoors (FR5).

6.1.3 Data Transfer Rate

When testing the data transfer rate of the system, an application called Network Speed was used to measure the data traffic over the Wi-Fi Direct connection. This application monitors the network traffic in real time and presents the results in a chart. The data transferred was images with a size of a couple of megabytes. Figure 6.1 presents the results of the test runs.

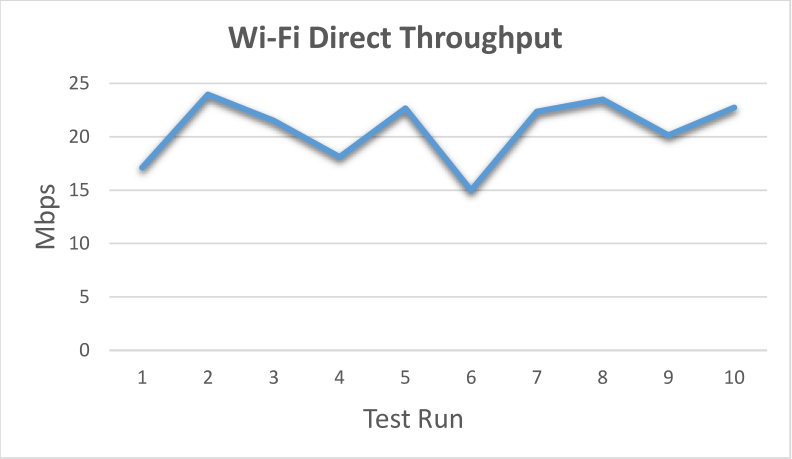

FIGURE 6.1: The results of the data transfer rate testing.

The transfer rate varied between each test run, and peaked at 24 Mbps. With an average of approximately 20 Mbps, the requirement of a minimum transfer rate of 1 Mbps (FR6) was satisfied.

6.1.4 Battery Lifetime

In order to test the battery lifetime of the system, the chat application was set up to automatically send messages. To simulate a heavy load, each device would send a message every second. The synchronization was to run until one of the devices powered off due to lack of power. By using the timestamp of the last received message, the battery lifetime could be calculated. In theory, this approach should have worked. However, it did not. After about 20 minutes, the Wi-Fi Direct connection between the devices was lost, and there is no automatic reconnect. This made it impossible to test the battery lifetime of the system. The problem isn't the synchronization system draining the battery, it's the fact that the devices could only maintain a Wi-Fi Direct connection for about 20 minutes.

6.1.5 Scalability

Most of the tests were run with only two devices, but since there most likely would be more than two people in a rescue crew, the system had to support a synchronization group of more than two devices. A third device was brought in on the action. The system proved to handle the three devices effortlessly. With *Server Push* as synchronization strategy, only the server peer would be affected by the increase of client peers. Sudden disconnects and frequent network partitioning did not cause any problems. However, if the server peer was to disconnect, it would dissolve the whole synchronization group. There is no server migration system present. The remaining client peers would then have to redo the labor intensive process of establishing a connection among them.

Due to the lack of test devices, it was not possible to test with more than three devices, hence it is unknown whether the system satisfies the requirement of supporting data synchronization among 2-8 devices (FR4). However, the only difference between a synchronization group of three and eight devices, is the amount of network traffic of the

server peer. Presumably, a synchronization group of eight devices should work with only the cost of shorter battery life of the server peer. This is, however, only speculation.

6.1.6 Summary

The implementation of Wi-Fi Direct in today's mobile devices seems very immature. With huge problems with both device discovery and connectivity, the communication technology seems unusable in the scenario of providing a communication channel between members of a rescue crew. For this system to be usable in such scenario, the connection issues needs to be fixed. Wi-Fi Direct is, however, a fairly new technology, and it is expected that its implementation in mobile devices will improve as the technology becomes more popular.

6.2 Data Synchronization

The data synchronization of the system is not as straight forward to evaluate as with communication technology. Key factors that needs to be addressed by the data synchronization system are the mobility and reduced resources of the devices. The mobility of the devices causes sudden disconnects and network partitioning, and with reduced resources it is required that the system is lightweight and energy efficient.

Sudden disconnects and network partitioning was handled quite well, as long as the server device stayed connected. If the server device lost the connection to the network, the whole synchronization group was dissolved. As discovered while testing the scalability of the system, a new server device was not appointed when the initial server device disconnected from the network. Hence, in order for the still active devices to continue synchronizing, the labor intensive process of establishing connection between the devices would have to be performed again. However, network partitioning due to disconnect of client devices did not affect the data synchronization among the still connected devices.

The proof-of-concept system developed in this thesis was to be an extension of UbiShare. UbiShare already had a data storage system in place, hence the introduction of a new storage system, such as Couchbase Mobile, would result in a complex restructuring of the system, which is outside the scope of this thesis. However, Couchbase Mobile might

be worth further investigation. Since Couchbase Mobile is a complete DBMS, it can be used the same way as the standard SQLite database. In addition to this, Couchbase Mobile has the functionality to connect to any other node running Couchbase, such as another mobile device. This could be used to synchronize the databases by using infrastructure-less wireless networks, in a peer-to-peer fashion.

Since the battery lifetime test failed, it is hard to say how energy efficient the data synchronization system is. Considering the chosen synchronization strategy, server push, the system should be less power demanding than by using client pull. This is due to the fact that client pull would continue to request changes from the server device even though no changes have been made. Resulting in unnecessary network traffic, and since the radio antennas of the mobile devices are the most power consuming components (except for the display) [21], it is safe to assume that the client pull approach would be more resource demanding than server push. In terms of network traffic, the peer-to-peer strategy using broadcasting/multicasting would be the best choice. This is arguably also the best strategy in terms of battery lifetime, due to its low network usage. However, this strategy requires a complex architecture to ensure reliable transmission of data.

Chapter 7

Conclusion and Further Work

The purpose of the proof-of-concept system developed in this thesis was to demonstrate how data synchronization using infrastructure-less wireless networks can aid disaster management in situations where internet infrastructure is inaccessible. Even though the system only was tested by using a simple chat application, it showed great potential. However, the today's implementation of Wi-Fi Direct in mobile devices was disappointing. Using Wi-Fi Direct as communication technology in the proof-of-concept system did suffice, but the implementation of the technology would need to be greatly improved for this system to be usable in a real life scenario. With a range of only 25 meters and an ability to only sustain a connection for approximately 20 minutes, the system would perform far from satisfactory.

7.1 Further Work

The implementation of the proof-of-concept is minimal. In order to make the system usable in real life scenarios, several features needs to be implemented.

Conflict resolution is such feature. If two devices makes changes to the same data simultaneously, it would cause a conflict. The proof-of-concept system does not handle such conflicts. A conflicting change would just overwrite any previously made changes. Conflict resolution can be implemented by marking the data with revision numbers, compare these when receiving a change and send a notification if the revision numbers do not match. This does, however, require a more extensive dialog between the client

and the server. Currently, a client device does not expect a response when notifying the server of changes. The server needs to send a response which the client could use to determine whether the change notification was successful or a conflict was detected.

The requirement stating that the user should be able to choose which community is to be synchronized (FR8), and the requirement that only members of this community should be able to participate in the synchronization group (FR9), was not satisfied with the proof-of-concept system. Currently, anyone in range can request to join the synchronization group. This is a security issue, and would have to be solved before using the system in real life scenarios.

Another improvement that can be made to the system is the handling of rejoining client devices. If a device disconnects and reconnects to the synchronization group later on, it currently needs to fetch all the data previously synchronized, even data it already has. A much better solution would be to only fetch the changes made while the client was offline. This could be done by storing the timestamp of the last change received by the client server-side. When the client then reconnects to the synchronization group, the server could use this timestamp to only send changes made while the client was offline.

Bluetooth was not fully implemented into the system due to time constraints. A nice feature to have in a complete system would be to support both Wi-Fi Direct and Bluetooth. Even though Bluetooth offers slower transfer rates and a shorter range then Wi-Fi Direct, its implementation in today's mobile devices seem much more robust than the implementation of Wi-Fi Direct. Bluetooth would also add support for legacy devices and devices with stricter requirements concerning battery usage.

The robustness of the proof-of-concept system could also be improved. If an error occurs, the system would forcefully terminate and only the logs can be used to determine what went wrong. A complete implementation of the system would have to handle errors and give better feedback to the user.

Appendix A

UML Diagrams

This appendix contains UML diagrams, such as class diagrams and database ER models.

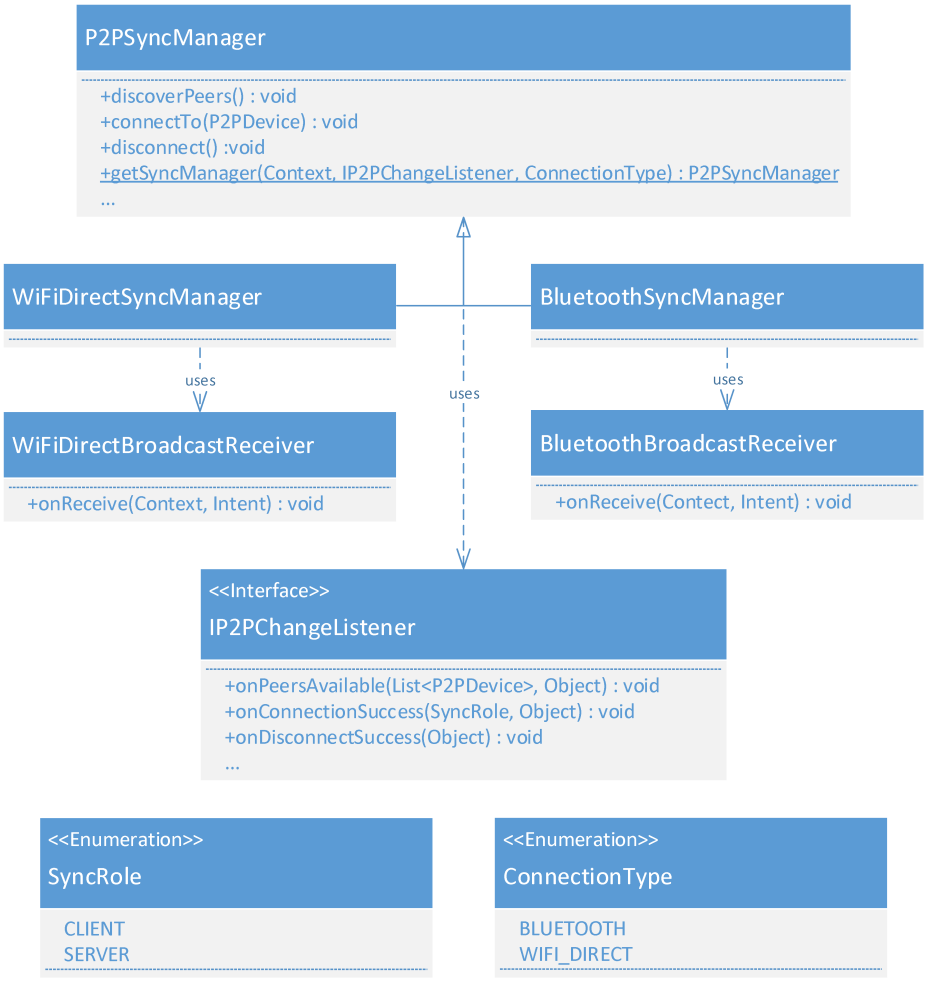

FIGURE A.1: Class diagram of the `org.societies.android.p2p` package.

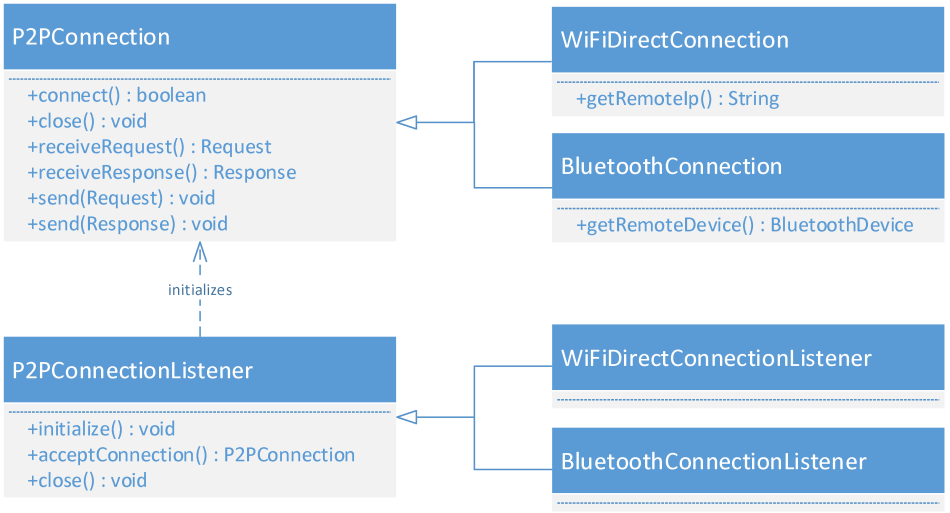

FIGURE A.2: Class diagram of the `org.societies.android.p2p.net` package.

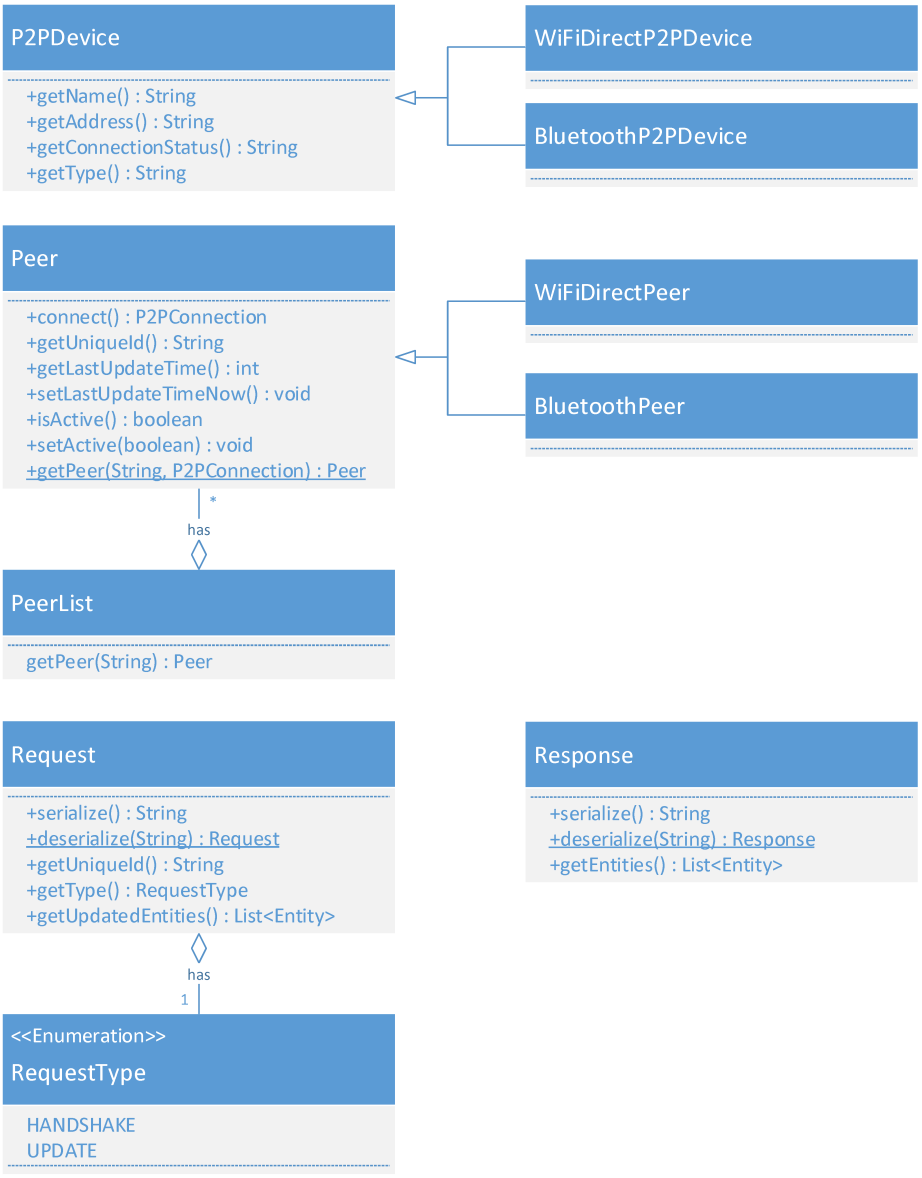

FIGURE A.3: Class diagram of the `org.societies.android.p2p.entity` package.

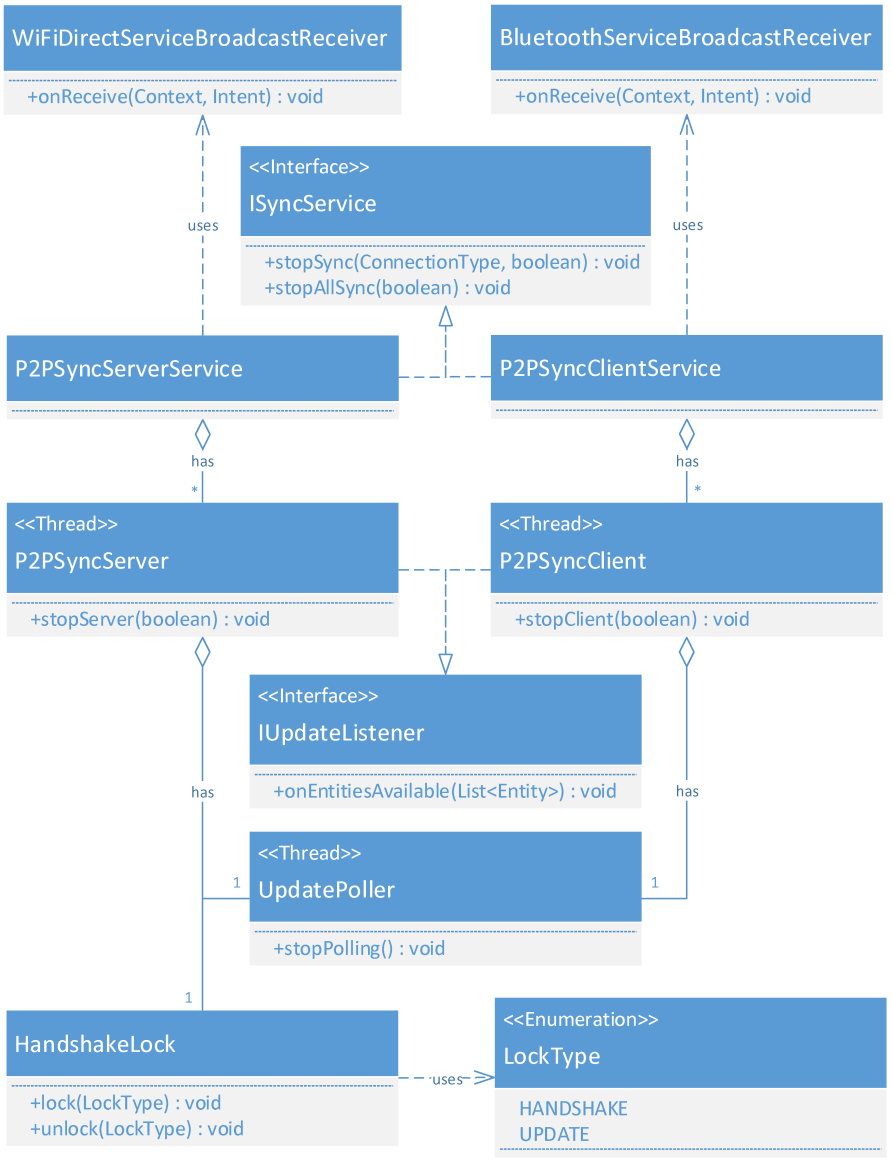

FIGURE A.4: Class diagram of the `org.societies.android.p2p.service` package.

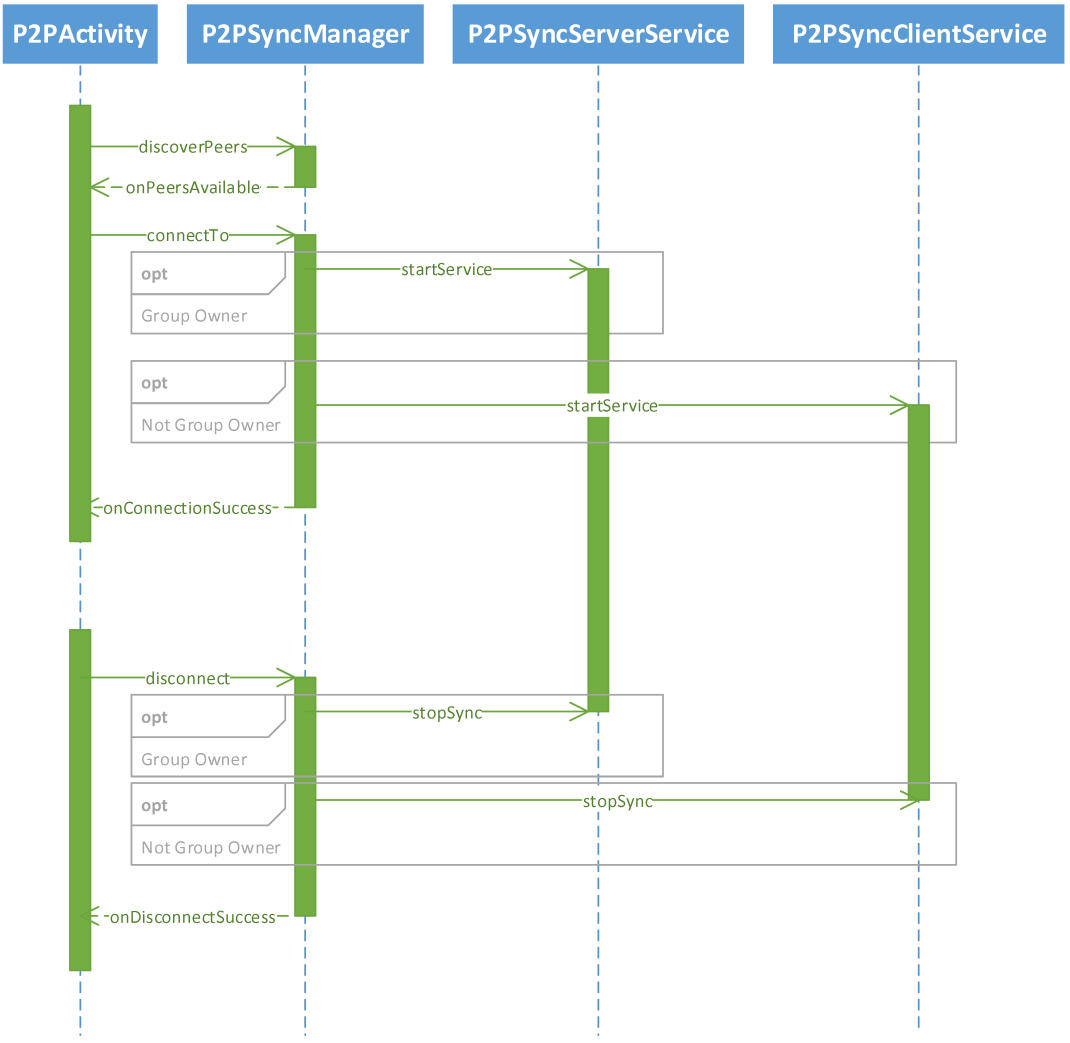

FIGURE A.5: Sequence diagram of the discovering and connecting to peers.

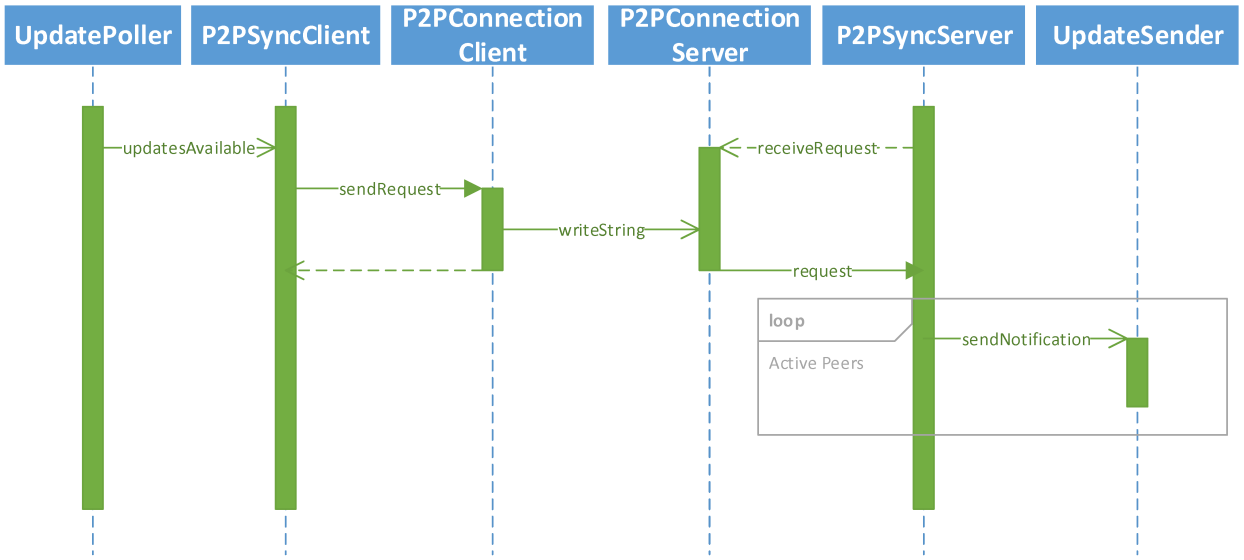

FIGURE A.6: Sequence diagram of pushing notifications.

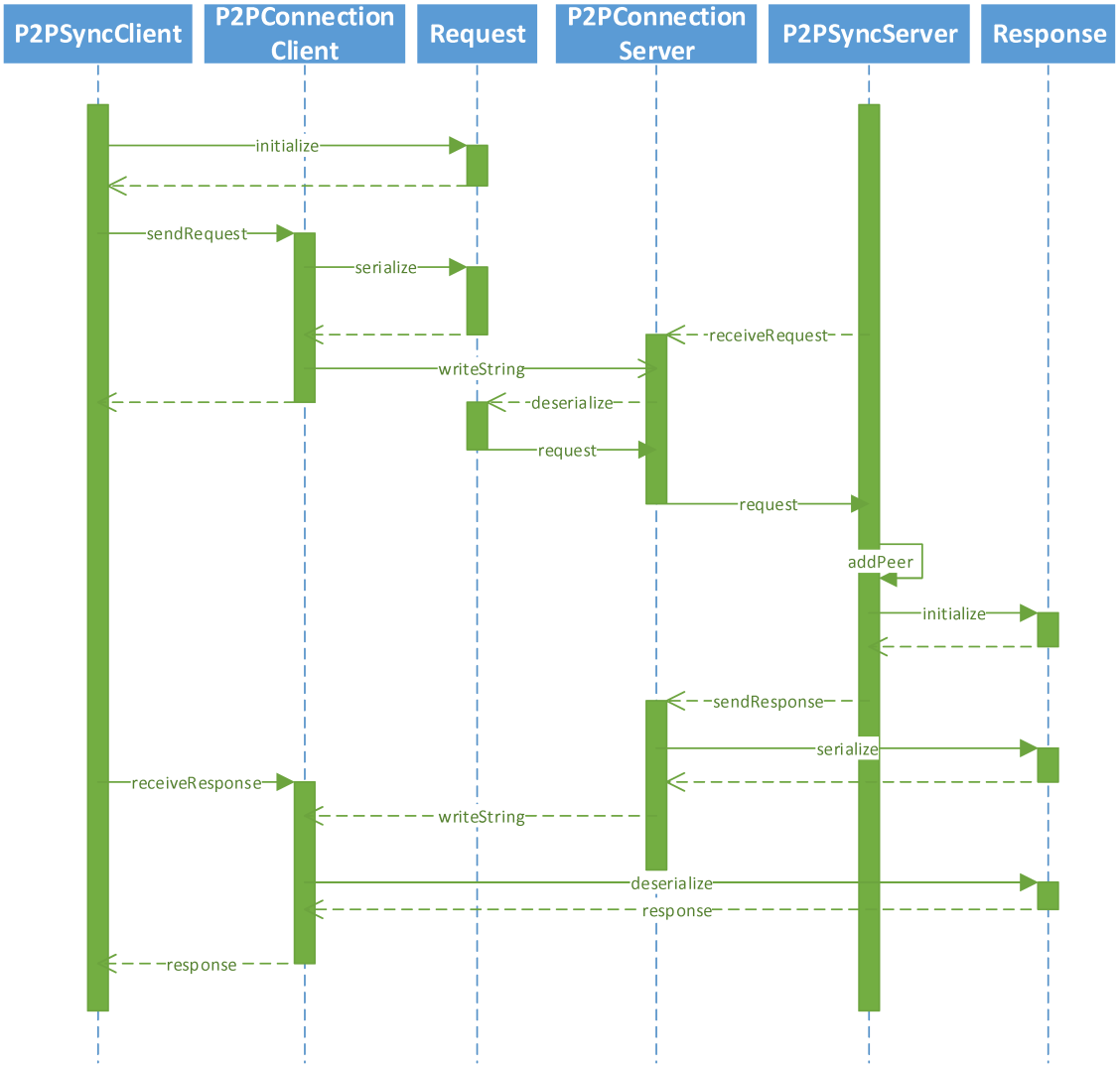

FIGURE A.7: Sequence diagram of the handshake request.

Bibliography

[1] NTNU. Tdt4501 - computer science, specialization project. 2012. URL http://www.idi.ntnu.no/emner/tdt4501/english.php.

[2] Babak A. Farshchian. Project description - ubishare: Social group management for android devices. 2012. URL http://ubicollab.org/projects/#UbiShare_Social_Group_Management_for_Android_Devices.

[3] Kato Stølen. Ubishare: Social group management for android devices. 2012.

[4] Saul Greenberg, Nicolai Marquardt, Till Ballendat, Rob Diaz-Marino, and Miaosen Wang. Proxemic interactions: the new ubicomp? *interactions*, 18(1):42–50, January 2011. ISSN 1072-5520. doi: 10.1145/1897239.1897250. URL http://doi.acm.org/10.1145/1897239.1897250.

[5] Mark Weiser. The computer for the 21st century. 1991. URL http://wiki.daimi.au.dk/pca/_files/weiser-orig.pdf.

[6] Sachin Agarwal, David Starobinski, and Ari Trachtenberg. On the scalability of data synchronization protocols for pdas and mobile devices. *Network, IEEE*, 16(4): 22–28, 2002.

[7] David Ratner Gerald J Popeky and Peter Reiher. The ward model: A replication architecture for mobile environments.

[8] Inc. Bluetooth SIG. A look at the basics of bluetooth wireless technology. 2013. URL http://www.bluetooth.com/Pages/Basics.aspx.

[9] Wikipedia. Bluetooth. 2013. URL http://en.wikipedia.org/wiki/Bluetooth.

[10] Wikipedia. Bluetooth low energy. 2013. URL http://en.wikipedia.org/wiki/Bluetooth_low_energy.

[11] Wi-Fi Alliance. Wi-fi certified wi-fi direct™: Frequently asked questions. 2010. URL http://www.wi-fi.org/sites/default/files/downloads-public/faq_20101021_Wi-Fi_Direct_FAQ.pdf.

[12] Wikipedia. Radio-frequency identification. 2013. URL http://en.wikipedia.org/wiki/Radio-frequency_identification.

[13] Wikipedia. Near field communication. 2013. URL https://en.wikipedia.org/wiki/Near_field_communication.

[14] David Murphy. 53 per cent of phones nfc-enabled by 2015, says frost. 2011. URL http://mobilemarketingmagazine.com/content/53-cent-phones-nfc-enabled-2015-says-frost.

[15] Wikipedia. Couchbase server. 2013. URL http://en.wikipedia.org/wiki/Couchbase_Server.

[16] Wikipedia. Shared nothing architecture. 2013. URL http://en.wikipedia.org/wiki/Shared-nothing_architecture.

[17] BitTorrent Inc. Bittorrent sync: Technology. 2012. URL http://labs.bittorrent.com/experiments/sync/technology.html.

[18] Wikipedia. Jxta. 2013. URL http://en.wikipedia.org/wiki/JXTA.

[19] Jérôme Verstrynge. Practical jxta ii. 2010. URL http://www.scribd.com/doc/47538921/Practical-JXTA-II.

[20] Dario Bottazzi, Antonio Corradi, and Rebecca Montanari. Enabling context-aware group collaboration in manets. In *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, pages 310–318. IEEE, 2005.

[21] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. 2010. URL http://static.usenix.org/event/usenix10/tech/full_papers/Carroll.pdf.