University of the West of England
BRISTOL

Copyright Notice

Library Digitisation Service

Development of a Proximity-Based Mobile Mesh Networking Framework

by

Justin Lewis Salmon

A project submitted in partial fulfilment for the degree of

Bachelor of Science
in
Computer Science

in the
Faculty of Environment and Technology
Department of Computer Science and Creative Technologies

March 2014

For Diana,
and her infinite patience

Abstract

Peer-to-peer proximity-based wireless networking can provide improved spatial and temporal semantics over traditional wireless topologies that rely on static network infrastructure. It can potentially enable unique new classes of independent proximity-based applications. However, the difficulties of setting up such peer-to-peer connections has thus far been a development barrier. There is currently a need for an abstraction tool to allow application developers to exploit the potential advantages of such networks with minimal knowledge of the underlying connectivity. We present Proxima, a framework and API for the Android platform, which employs ad-hoc device-to-device connections and proactive mesh routing for a decentralised topology with solely proximity-based rich content dissemination. The framework is designed to be developer- and user-friendly with minimal configuration effort, lightweight, reusable and hardware independent. We have further developed a real-life application, named TuneSpy, based around sharing music with local peers. TuneSpy makes novel use of proximity-based functionality, demonstrates the ease of writing proximity-based applications with Proxima, and has served as a multi-device testing platform for the Proxima framework.

Keywords: *mesh networking; ad-hoc; proximity-based; peer-to-peer; android.*

Acknowledgements

First and foremost, I would like to extend my sincerest gratitude to my supervisor Dr Rong Yang for her enduring support and encouragement throughout the project, and indeed throughout the past four years at UWE. I would like to thank Rong for inspiring and motivating me to publish my work, and for her highly helpful contributions and invaluable feedback throughout. Any success this project earns is a great reflection of her guidance.

To my beautiful Diana, without whose patience, toleration and endless support this project would most certainly have suffered, I offer my eternal gratitude. It is not often in one's life when a single piece of work dominates that life so completely, and I am hugely grateful to her for staying by my side so firmly throughout the past year.

I would also like to thank my friend Francesco Velotti for his unremitting academic encouragement, and for introducing me to LaTeX, without which this report would not be as gratifying to me as it turned out to be.

Finally, special thanks go to my mother Hazel for her continuing moral support throughout, and for her indispensable help with proofreading.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growing prevalence of computing devices in our everyday lives has fundamentally shaped and changed the way we communicate with one another. We are now able to communicate over vast distances, effectively instantaneously, through the Internet. In recent years, these devices have become increasingly advanced both in terms of processing power and hardware sophistication.

Moreover, these everyday computing devices are also becoming increasingly mobile. As of February 2012, 46% of adults in the US owned a smartphone (Kim *et al.*, 2014) and this figure is on a firm upward trend. Smartphones and other smart devices are generally capable of wireless Internet access via Wi-Fi and/or cellular connections such as 3G and EDGE.

Of course, there are countless benefits associated with this capability. But there are certain situations where this functionality is not possible. On an aeroplane or a train for example; or in regions stricken by natural disaster or electrical malfunction. This is due to the dependence upon immovable networking infrastructure such as Wi-Fi access points or cellular network towers.

In addition, there are also certain scenarios where this functionality is not necessary. The traditional infrastructure-based paradigm has a key shortcoming, from the perspective of *information publication*. To share information with one another, mobile device users must generally publish the information to a centralised server, from which the recipients may subsequently retrieve it. Sometimes, this behaviour is undesirable, for the following reasons:

Spatial dependency. The information may be spatially dependent, i.e. it may have a limited localised audience and may only be relevant within a specific localised

area. Publishing the information for the entire Internet to see may be unnecessary and/or redundant.

Temporal dependency. The information may be temporally dependent, i.e. only relevant within a short time frame of its creation. It may have *ephemeral* value, there being thus no use for it except for right at the very moment.

Put more succinctly, distance-independent infrastructure-based communication has a lack of semantics relating to notions of spatial and temporal locality, or lacks a sense of *"here-and-now"* (Bostanipour *et al.*, 2012).

This project aims to explore a different type of networking paradigm, whereby reliance is solely upon mobile users being within proximity of one another, in a peer-to-peer orientation. Publication of information is also solely to proximal neighbours, and thus does not rely on additional networking infrastructure. We refer to this paradigm as the *proximity-based* paradigm throughout the remainder of the report. We explore the technologies and protocols that might be employed on mobile devices to create distance-dependent networks that make novel use of spatial and temporal semantics. Surely, with the advanced hardware sitting in our pockets, this type of communication paradigm should be possible.

1.1 The *proximity-based* paradigm

Originally, human communication required proximity. One had to be close enough to another person for vocal communication to be effective. Computers, on the other hand, did not originally have any concept of proximity. For two computers to communicate, all that was required was a network cable to connect them. Of course, the length of such a network cable defines the maximum communication distance of a wired connection, but as Wide Area Networks (WANs) such as the Internet quickly grew to span intercontinental distances, this detail became largely irrelevant.

With the advent of wireless communication, the notion of proximity was introduced to the computing world. For a computer to be able to communicate with others wirelessly, it had to be within operable range of some transceiver that was capable of accepting messages from the computer and forwarding them to their intended recipients, as well as delivering messages intended for the computer.

These days, most of us are familiar with this paradigm. Our laptops must be within range of our wireless home router to be able to browse the Web, and our mobile phones must be within range of a cellular base station in order to receive text messages and phone calls from friends and family. However, this does not bring us back to the original human communication paradigm. This means of communication, although proximity-based, is not truly peer-to-peer. It is limited by the necessity of having static infrastructure in place for computing devices to connect to, and is therefore an *indirect* form of communication. It is analogous to two humans requiring an intermediate message-passing human (such as a translator) in order to talk to one another.

Additionally, the static infrastructure requirement renders any device outside of the operable range of this infrastructure completely useless for communication purposes. All smartphone users are familiar with this frustrating limitation; not being able to use your home Wi-Fi in the garden for example, or not being able to receive cellular network coverage in remote (or sometimes not-so-remote) locations.

1.1.1 The train journey scenario

To illustrate the idea of the proximity-based paradigm, let us imagine a scenario where one might benefit from purely localised, ephemeral, decentralised information dissemination. Consider, for example, a hypothetical train journey. Two people, Alice and Bob, are both passengers on this journey. Bob happens to be using his smartphone to listen to a track by his favourite music artist through a pair of headphones. Alice, who also has a smartphone, notices Bob, and begins to wonder what he might be listening to. She wishes that there were some way for her to "tune-in" to what Bob is currently listening to (assuming that Bob has pre-authorised this type of sharing behaviour), as if Bob were able to "broadcast" his current track to the people around him. Assuming that this train does not have a Wi-Fi access point available, how is Alice able to listen to Bob? How might their smartphones be able to connect directly to and share data with each other, simply based on the fact that they are in proximity to each other? The advanced capabilities of their devices should surely allow this behaviour.

The *here-and-now* nature of this encounter is clear. Alice must be spatially close to Bob, and the information she is able to retrieve is temporally dependent, i.e. Bob will only be listening to a particular track for a certain period of time; after that time, the

information is irrelevant and possibly unobtainable. Such is the ephemeral nature of the proximity-based paradigm.

1.1.2 The earthquake scenario

The train journey scenario illustrates a situation where dependence upon static infrastructure is not required. Let us now imagine a scenario where this dependence would not be physically possible. Consider a hypothetical situation, where electrical power has been cut off due to damage from a recent earthquake. It would be necessary for disaster relief workers to communicate in the local area, to coordinate the relief operation. However, wireless Internet access would not be possible in this situation, as Wi-Fi access points and cellular network towers would not have access to mains electricity, and may even have been damaged. It would be hugely beneficial if it were possible to communicate wirelessly in this situation for coordination purposes. Since it is likely that a large percentage of people in the area would have access to smartphones or other devices, would it be possible to connect these devices together based on proximity, to form some kind of ad-hoc network? Would it also be possible for each device to act as a relay for other devices, thereby increasing the overall connection range? This would allow relief workers to pass information throughout the disaster-stricken area, and could potentially save lives that might otherwise be lost.

1.1.3 Potential benefits

The concept of purely localised, ephemeral, decentralised information dissemination has several apparent advantages. The ability to sense and connect with those around us allows purely real-time and real-space communication and collaboration. The lack of infrastructure dependence also enables communication in a far wider range of locations.

There are unique potentials for applications that might necessitate or otherwise exploit these *here-and-now* semantics, such as social, multimedia, crisis management, and gaming. There are some scenarios where dissemination onto the wider Internet is not desired, not useful, or simply not necessary due to the inherent importance of the temporal and spatial aspects.

Aside from the unique semantic qualities, what other benefits do we acquire from this paradigm? The following list summarises the potential advantages that have been identified so far:

- **Peer-to-peer**: since no additional infrastructure is needed, such as Wi-Fi access points or cellular network towers, communication is directly between devices.
- **Increased mobility**: the lack of infrastructure dependence makes communication possible in a far more diverse range of locations (practically anywhere except inside a Faraday cage).
- **Unexplored applications**: new types of cooperative, proximity-based applications might be possible.

1.2 The problem

As it stands today, there is no ready-to-use solution that provides the aforementioned functionality. There is no reusable platform on which application developers can build that can make use of proximity-based peer-to-peer connections in the way that we have envisioned above.

This is in part due to the complexity and difficulty of setting up such connections. Traditional infrastructure networks are generally configured using a set of well-known protocols such as DHCP to assign new devices unique IP addresses, and DNS to convert between IP addresses and human-readable hostnames. These mechanisms are generally automatic, and do not require the user to intervene in the process. Peer-to-peer networks do not have standard precedents for these mechanisms, although many have been proposed. This is also in part due to the various underlying technologies that could potentially be used.

This project aims to tackle these issues, using modern tools to create a platform that is capable of operation regardless of infrastructure availability, and reaches as close to zero-configuration as possible. This will allow future application developers to exploit the potential benefits of the proximity-based paradigm. The specific goals and objectives for this project are described in detail in Chapter 5.

1.3 Current state-of-the-art

The concept of proximity-based mobile computing is not new. Bluetooth, for example, is a widely used technology for short-range device-to-device communication[1]. It has been

[1]Bluetooth is discussed in greater detail in Chapter 2.

around for many years and has gone through many standards. It does not, however, meet our operable range and speed requirements, due to the complexity and data volume requirements of the envisioned applications. It is simply too slow and short-range.

There are a number of other existing solutions for proximity-based networking, however none meet all of the requirements outlined in the previous section. These existing solutions, along with the reasons for their unsuitability, are discussed in detail in Chapter 4.

1.4 Making it happen

For our train journey scenario to be successful, what exactly do we need in terms of technology? How can we actually achieve a true proximity-based paradigm? To answer this question, we must break the problem down into its constituent parts, and analyse the practicality of each. Here, we will make an initial informal analysis, before properly formalising the requirements in Chapter 6. The following is a brief outline of some of the decisions that must be made in terms of technology for a successful realisation of our paradigm:

Communication mechanism. Devices must be able to sense each other without any additional infrastructure. The chosen mechanism should be fast and have an acceptable operational range.

Network topology. Devices should be able to communicate with multiple other devices simultaneously, so the chosen network topology should accommodate this.

Routing. Ideally, any device should be able to reach any other device, possibly by routing messages through intermediate devices.

Automatic connection and configuration. If two devices come within proximity of one another, they should be able to sense and connect to one another automatically, without any manual setup procedure. The user should have to configure as little as possible. Processes such as IP address assignment should happen automatically.

Node identification. Each device should be uniquely identifiable to other devices with a human-readable handle.

1.5 Report structure

This section outlines the over-arching structure of the report. The remainder of the report is structured into five main sections:

- **Research**: Chapter 2, Chapter 3 and Chapter 4 form the background research section of the report.
- **Requirements and scope**: Chapter 5 and Chapter 6 form the requirements analysis and project scope section of the report.
- **Design**: Chapter 7 and Chapter 8 form the software design and testing section of the report.
- **Implementation**: Chapter 9 and Chapter 10 form the implementation section of the report.
- **Evaluation**: Chapter 11 forms the evaluation and reflection section of the report.

Table 1.1 gives a chapter breakdown of the report, with a brief outline for each chapter for ease of reference.

Chapter	Description
Introduction	Chapter 1 sets the scene for the remainder of the report. The core problems to be tackled are described.
Background concepts	Chapter 2 provides an introduction to the background concepts necessary for the understanding of the remainder of the report. The reader is directed to more in-depth material where pertinent.
Android	Chapter 3 describes the fundamentals required for an understanding of the Android operating system, and also details how proximity-based functionality might be implemented on Android.
Related work	Chapter 4 attempts to evaluate some existing work related to the problems identified in Chapter 1 and discusses their suitability and relevance the project.

Project scope	Chapter 5 describes the scope of the project, in terms of hypotheses and objectives. Details of the project management and development methodology approaches and timeline are also given.
Requirements analysis	Chapter 6 identifies and formalises the functional and non-functional requirements of the software to be developed.
Design	Chapter 7 outlines the design process that was undergone to create the software and illustrates it with standard UML diagrams where appropriate.
Testing	Chapter 8 gives details of the testing specification for the project, as well as details related to testing in general on Android systems.
Implementation	Chapter 9 discusses specific implementation details related to the development of the software.
Example application	Chapter 10 describes the secondary project deliverable *TuneSpy*, a non-trivial example application written using the core proximity-based framework.
Conclusion	Chapter 11 evaluates the whole process and attempts to draw some appropriate conclusions about the post-implementation state of affairs. Some possible areas for future development are also discussed.

Table 1.1: Chapter breakdown.

Chapter 2

Background concepts

In this chapter, we will first discuss some existing wireless networking technologies and relate their suitability back to the vision for the proximity-based paradigm, which we described in Chapter 1. We will identify an appropriate technology for the underlying connectivity mechanism, called *ad-hoc Wi-Fi*. We will then explain the concepts of *mesh networking*, specifically Mobile Ad-Hoc Network (MANET)s. Following this, we go on to explain how information routing occurs in MANETs, focusing specifically on the Optimized Link State Routing (OLSR) protocol. Finally, we discuss some issues and challenges that face MANET users and developers, taken from the viewpoint of the project goal of developing a framework for proximity-based computing.

2.1 Traditional wireless networking

This section gives a basic overview of the traditional means by which mobile devices have used to communicate.

2.1.1 Wi-Fi

Wi-Fi (Wireless Fidelity) is a mode of wireless network operation standardised by the IEEE, and first defined in the 802.11b standard. It is otherwise known as the Basic Service Set (BSS) when a single access point is involved, or the Extended Service Set (ESS) for multiple access points. It is also commonly known as *infrastructure mode* Wi-Fi. See Figure 2.1 for an illustration of a typical infrastructure Wi-Fi network.

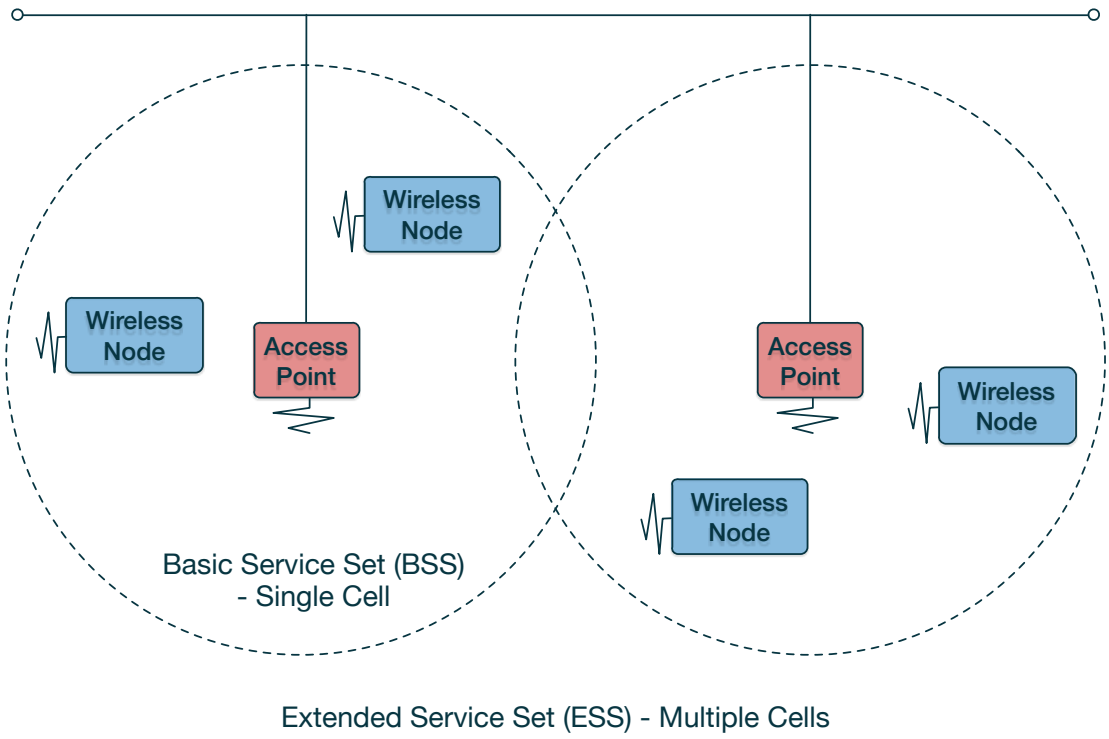


FIGURE 2.1: A typical infrastructure Wi-Fi network.

Infrastructure Wi-Fi relies upon the mobile device user being within range of a fixed wireless network access point (AP). These access points can be found in a wide variety of locations, including most modern households, coffee shops, university campuses, restaurants, etc. This is the wireless networking paradigm that people are most familiar with. However, in places that do not offer an access point, Wi-Fi connectivity is not available.

2.1.2 Cellular networks

Cellular networks are used generally to make phonecalls and send/receive SMS messages, although in recent years they have also been used for Internet access via technologies such as 3G, LTE, GPRS, and EDGE. They are composed of fixed network infrastructure towers. As a mobile user moves around, a handover procedure transfers the user to the nearest base station. These networks generally contain hundreds or thousands of wirelessly interconnected network towers with periodic hard links to Internet backbones. The towers are generally very expensive. Similar to infrastructure Wi-Fi, if the mobile device user is not within range of a cellular network tower, then connectivity will not be available.

2.1.3 Summary

These traditional networking technologies go against our requirement for lack of infrastructure dependence. So, we need to investigate different technologies that do not have this dependency and will allow us to connect devices directly together.

2.2 Peer-to-peer networking

In this section, we describe the available technologies that allow us to connect devices in a peer-to-peer fashion. This is exactly the type of connectivity mechanism that we require for the proximity-based paradigm.

2.2.1 Bluetooth

The Bluetooth technology, standardised by the Bluetooth Special Interest Group (Bluetooth SIG, 2014) is a relatively old technology, having been around since 1994. It has gone through many standards in its lifetime. It uses short-range radio waves to connect devices directly together. It is available natively on most modern mobile phones, tablets, laptops, and other electronic devices. However, the operable range of Bluetooth is generally quite short, reaching a typical maximum range of 30 metres. It is also somewhat slow in terms of transfer speed, reaching a typical maximum of 3Mbps. Due to the way the Bluetooth protocol works, it is only capable of connecting up to 7 devices together in any single network.

Although the Bluetooth mechanism initially seems promising, the aforementioned limitations make it an unsuitable choice for our purposes.

2.2.2 NFC

NFC is used for very short-range, low-bandwidth communications. It uses magnetic induction to create a low power radio-wave field which the NFC hardware can detect and use to transfer small amounts of data. It is used for things like contactless payment and pairing of devices, and is available on a very limited range of mobile devices. It can be used to pair devices for Wi-Fi Direct, instead of needing to enter passwords. Although a promising technology with several potential use cases, the range and bandwidth limitations mean that NFC will not be suitable for our requirements and thus we will not consider it further.

2.2.3 Wi-Fi Direct

Wi-Fi Direct is a s a relatively new standard proposed by the WiFi Alliance (WiFi Alliance, 2014) and is aimed at creating point-to-point connections between mobile devices. The main idea of Wi-Fi Direct is to create a Wi-Fi access point on the mobile device itself, via software, to achieve connectivity without a real access point.

However, Wi-Fi Direct is not a truly peer-to-peer technology. Devices negotiate roles; one assumes the access point (AP) role, otherwise known as the Group Owner (GO). The remaining devices connect to the GO as if it were a regular Wi-Fi AP. This makes it a hierarchical network, which limits the potential network topologies and hinders scalability. It is also not capable of multi-hop routing.

WiFi Direct is available on Android devices running version 4.0 and above, and is referred to as the Wi-Fi P2P API. It provides a robust interface allowing people to develop mobile applications that make use of peer-to-peer device connections. However, not all 4.0 devices contain the necessary special hardware that Wi-Fi Direct requires. Additionally, there are still a large number of devices running Android versions lower than 4.0 which are not compatible with the Wi-Fi P2P API. See Section 3.1 for detailed Android version distributions.

In summary, Wi-Fi Direct is hardware-dependent, is not capable of multi-hop routing, and requires a manual setup procedure. These issues go against our initial aims.

2.2.4 Ad-hoc Wi-Fi

Ad-hoc Wi-Fi, otherwise known as the Independent Basic Service Set (IBSS), is an alternative operation mode allowing connection without an access point. It is also standardised by the IEEE and was first defined in the same 802.11b standard as regular infrastructure Wi-Fi, and is hence a relatively old technology. Unlimited devices can be connected at the same speed and range as regular infrastructure Wi-Fi. See Figure 2.2 for an illustration of a typical ad-hoc Wi-Fi network configuration.

Support for ad-hoc mode varies between mobile devices. All devices that support infrastructure Wi-Fi mode should support ad-hoc mode, in theory, since they are defined in the same standard. Strangely, support for ad-hoc mode on Android devices is disabled by default. Modifications must be made to the device to re-enable it. These modifications are discussed in detail in Chapter 3.

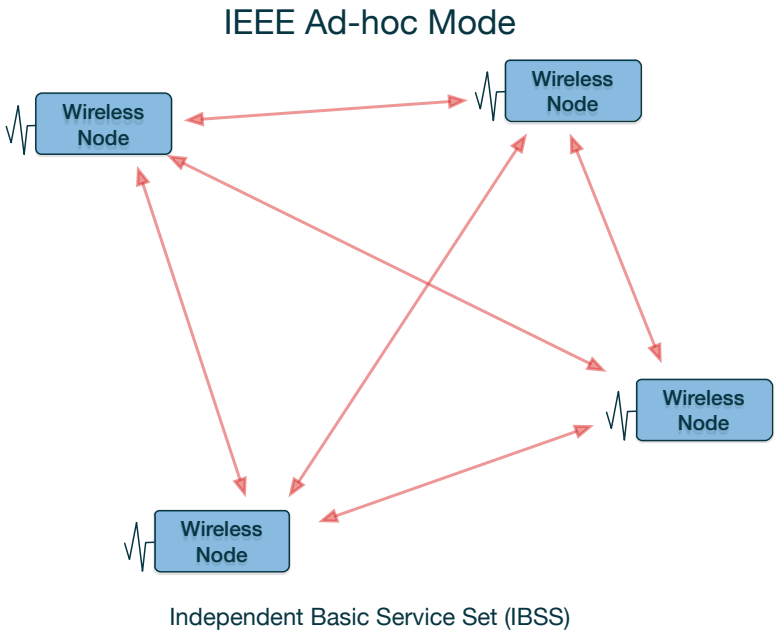


FIGURE 2.2: A typical ad-hoc Wi-Fi network.

2.2.5 Comparison

Having identified the possible technologies, we must now decide which one we are going to use to fulfil our initial vision of the proximity-based paradigm. Table 2.1 provides a comparison of the aforementioned technologies.

	Bluetooth	**Wi-Fi Direct**	**Ad-hoc Wi-Fi**	**NFC**
Topology	Peer-to-peer, up to 7 devices	Device-to-device, hierarchical groups	Peer-to-peer, unlimited devices	Peer-to-peer, one-to-one
Speed	3Mbps typical	Up to 250Mbps	Up to 54Mbps	0.424Mbps
Range	30ft typical	Up to 150ft	Up to 150ft	4in
Android Support	Native	Only on newer devices	Modification required	Only on newer devices

TABLE 2.1: Comparison of P2P wireless technologies.

Realistically, the decision is between ad-hoc Wi-Fi and Wi-Fi Direct. These are the only solutions that satisfy our range and speed requirements. But do they satisfy our remaining requirements? Table 2.2 summarises the benefits and drawbacks of the two technologies.

Benefits of Wi-Fi Direct	**Benefits of ad-hoc Wi-Fi**
• Easy to temporarily connect a small number of devices • Encryption and security built-in • Service discovery built-in	• True peer-to-peer connectivity • Capable of handling dynamic topology changes • Possible to create large mesh networks • Simple protocol
Drawbacks of Wi-Fi Direct	**Drawbacks of ad-hoc Wi-Fi**
• Hierarchical topology, not truly peer-to-peer • Responds poorly to topology changes • Complex protocol	• Difficult to use out-of-the-box • Security and encryption not built-in • Service discovery not built-in

TABLE 2.2: Comparison of Wi-Fi Direct and ad-hoc Wi-Fi.

The proposed solution is to use Wi-Fi ad-hoc mode. The need to make non-standard modifications to devices is considered acceptable for the scope of this project. The advantages of ad-hoc connectivity outweigh the tediousness of the modifications. Since we shall be making these modifications at the framework level, we remove the need for application developers to repeat this process.

2.2.6 Next steps

So, we have a mechanism that gives us the peer-to-peer connectivity of Bluetooth and the speed and operable range of Wi-Fi. However, this provides only the basic connectivity. More software is needed for it to be useful for complex applications. Going back to our initial paradigm, we identified the need to be able to route messages through other devices, in order to reach as many devices as possible. This can be achieved through mesh networking, described in the next section.

2.3 Mesh networking

This section describes the basic aspects of mesh networking. A mesh network is a group of nodes arranged in a mesh topology. Nodes in a mesh network can be mobile devices, laptops or other wireless devices. Although mesh networks are considered a type of ad-hoc network. they offer several advantageous features over basic ad-hoc networks:

Decentralisation. Since mesh networks are purely peer-to-peer, there is no concept of centralisation or hierarchical structure. They offer a purely decentralised architecture.

Multi-hop routing. Each node in a mesh network participates in routing information to other nodes, thus every node in the mesh is capable of communicating with every other node. This is known as *multi-hop* routing and is described in detail in Section 2.4.

Self-healing. Mesh networks are tolerant to failure. If an intermediate node goes offline, other nodes will still be able to communicate either directly or through other intermediate nodes.

Gateway nodes. Some nodes in the mesh may act as *gateways*, providing Internet access to the rest of the nodes.

Mesh networks are generally wireless. See Figure 2.3 for an illustration of a typical wireless mesh network. Each node is capable of connecting directly to the other nodes that are within range of it.

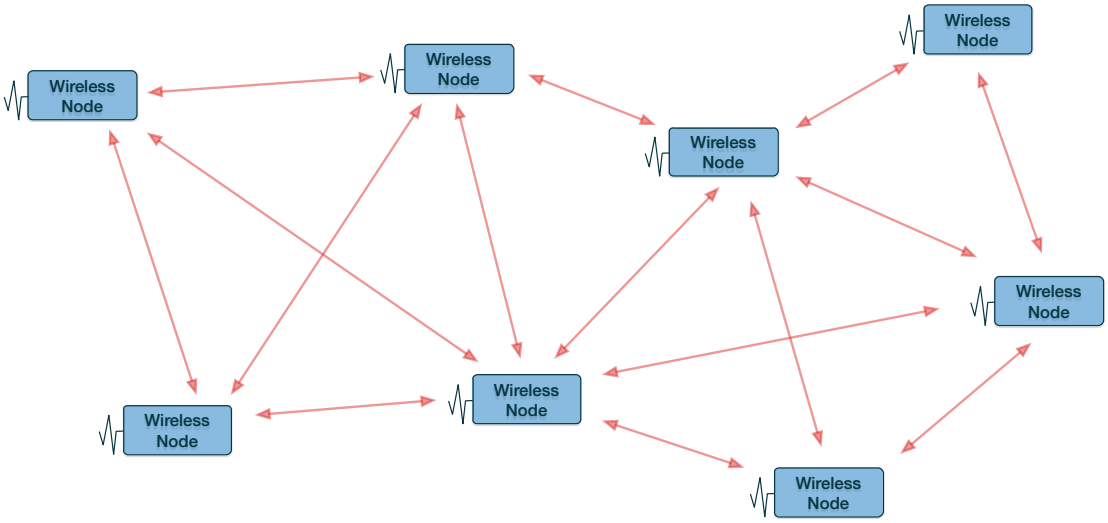


FIGURE 2.3: A typical mesh network.

2.3.1 IWMNs

IWMNs are wireless mesh networks with fixed nodes. These can be rapidly deployed in emergency situations for local communication. The cost of the required networking equipment is generally quite high. They can use a variety of communication technologies, such as WiMax, ZigBee or radio links. They can have a very wide range, in the order of thousands of square miles.

2.3.2 MANETs

Mobile Ad-Hoc Networks (MANETs) are mesh networks comprised of mobile nodes. They can be deployed just as rapidly as IWMNs, without the additional infrastructure cost. However, they must also deal with the added complexity introduced by the mobility of each node. The rest of this chapter focuses on MANETs, since we know that we will be dealing solely with mobile devices.

The Internet Engineering Task Force (IETF) has recently set up a working group for MANETs. The working group mandate states the following:

> "To standardize IP routing protocol functionality suitable for wireless routing applications within both static and dynamic topologies. The fundamental design issues are that the wireless link interfaces have some unique routing interface characteristics and that node topologies within a wireless routing region may experience increased dynamics due to node motion or other factors."
>
> (IETF, 2014)

The working group has proposed a variety of routing protocols, some of which have been accepted as Requests for Comments (RFCs), most notably OLSR and Ad-Hoc On-Demand Distance Vector Routing (AODV). We discuss both of these protocols in the following section.

2.4 Routing in MANETs

As mentioned earlier, mesh networks are capable of multi-hop routing. This is enabled by the routing protocol software in use on the network. In the case of MANETs, the routing requirements are somewhat different, due to the dynamic nature of the devices.

There are a vast number of different routing protocols and algorithms, some with multiple implementations. They can be broadly split into three categories: *proactive*, *reactive* and *hybrid* protocols. Each category is discussed in turn in the following sections.

2.4.1 Proactive (table-driven) routing

Proactive routing protocols are based on the principle of maintaining information about the entire network on every device. Each device maintains lists of neighbouring nodes,

along with known routes to those nodes. Periodic distribution of routes over the network keeps other nodes updated.

The main advantage of this approach is that there is relatively little delay when looking up routing information. However, proactive protocols have the disadvantageous overhead of needing to store routing tables about the whole network, some of which may be redundant. Proactive networks are also less reactive to node failures and reorganisations, due to the delay in propagating topology changes throughout the network.

The most widely-used and well-known proactive protocol is the Optimized Link State Routing (OLSR) protocol. It was originally defined in RFC 3626 (Clausen *et al.*, 2003) by the IETF. OLSR has multiple implementations; the most well known of which is the *olsrd* implementation (Tonnesen *et al.*, 2008), which contains specific optimisations for mobile devices. Section 2.5 describes OLSR in detail.

2.4.2 Reactive (on-demand) routing

Reactive protocols seek out routes on-demand and do not store full network topology information. When a node needs to discover a route to a neighbour node, it floods the network with route-request messages. Intermediate nodes pass on the route request messages, until one reaches the destination. The destination node then sends a route-reply message back through the network, and the source node stores the route that the route-reply message took. Routes remain valid for as long as the source and destination nodes are transmitting data. Once transmission stops, the route is deleted.

While this approach is beneficial in terms of storage overhead, it can potentially lead to network overload due to excessive flooding, and can be slow to find routes due to the initial route-finding delay.

The most recognised reactive routing protocol is the Ad-Hoc On-Demand Distance Vector (AODV) protocol, defined by the IETF in RFC 3561 (Perkins *et al.*, 2003).

2.4.3 Hybrid routing

Hybrid protocols aim to combine the advantages of the two previously mentioned protocol types. An example of a hybrid protocol is the Zone Routing Protocol (ZRP) (Haas, 1997). While theoretically interesting, hybrid protocols have not gained much traction in practical environments.

2.4.4 Summary

Routing in ad-hoc networks is a subject of considerable active research. Having investigated the available routing protocols and their respective advantages and disadvantages, we must now decide which protocol we should use, bearing in mind our initial requirements for the proximity-based paradigm. A good in-depth study of the various routing protocol types is given in (Campista *et al.*, 2008).

Since the envisaged networks will be relatively small, we will use a proactive routing protocol, as the storage overhead will be minimal. It makes sense to choose a well-defined and well-tested protocol with proven deployments. Hence, we will use the OLSR protocol, since it is the leading proactive protocol and has been specifically designed and optimised for MANETs. The next section describes the architecture and operation of OLSR in detail.

2.5 Optimized Link State Routing (OLSR)

The OLSR protocol is defined in the Request for Comments (RFC) 3626 (Clausen *et al.*, 2003). It is a proactive, table-driven protocol that features a unique traffic flooding optimisation named Multi-Point Relaying (MPR). MPR is explained in Section 2.5.4. Throughout the remainder of this report, we will focus uniquely on the OLSR protocol when mentioning routing in mesh networks.

2.5.1 Message types

OLSR defines four core message types, which are used to control the network and distribute routing information. An OLSR message is wrapped in an OLSR packet, and each OLSR packet can contain one or more OLSR messages. The four control message types are as follows:

HELLO. HELLO messages are used for first-hop neighbour sensing, and as a basis for a node to select its MPR candidates. They are transmitted to all neighbours in order to establish one-hop connections.

TC (Topology Control). TC messages are used to transmit topological information throughout the network, and are used in conjunction with MPR.

MID (Multiple Interface Declaration). MID messages are used by nodes who are using multiple network interfaces, and list all IP addresses in use on a given node.

HNA (Host or Network Announcement). HNA messages are used by nodes who wish to announce that Internet connectivity is available through itself on another interface.

2.5.2 Packet structure

All OLSR packets share the same generic superstructure. These packets can contain one or more OLSR messages, and are then embedded in UDP datagrams for network transmission. Figure 2.4 describes this generic packet structure.

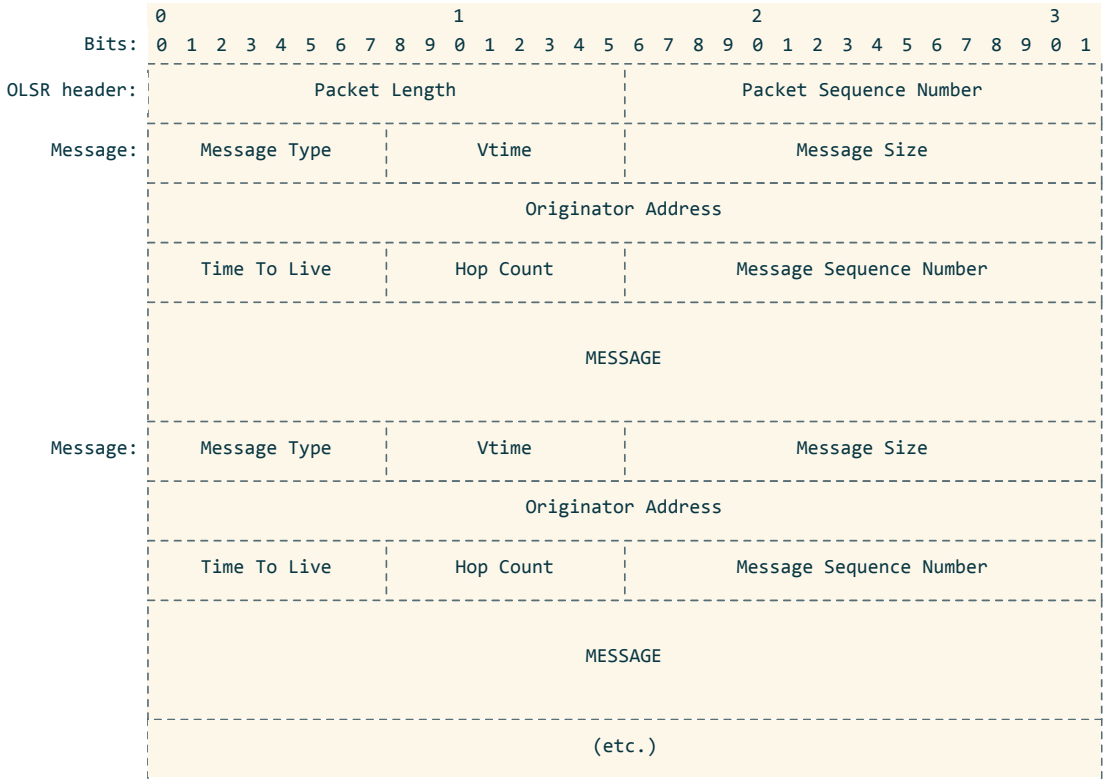


FIGURE 2.4: Format of a generic OLSR packet.

2.5.3 Neighbour discovery

In this section, the mechanism by which OLSR detects neighbours is described. The general idea is to disseminate messages from a node across the network in order to receive replies from neighbours. These messages will contain information about the node itself, as well as information that the node currently has about other nodes.

This is achieved via a special type of message, called a `HELLO` message. The structure of a `HELLO` message is given in Figure 2.5.

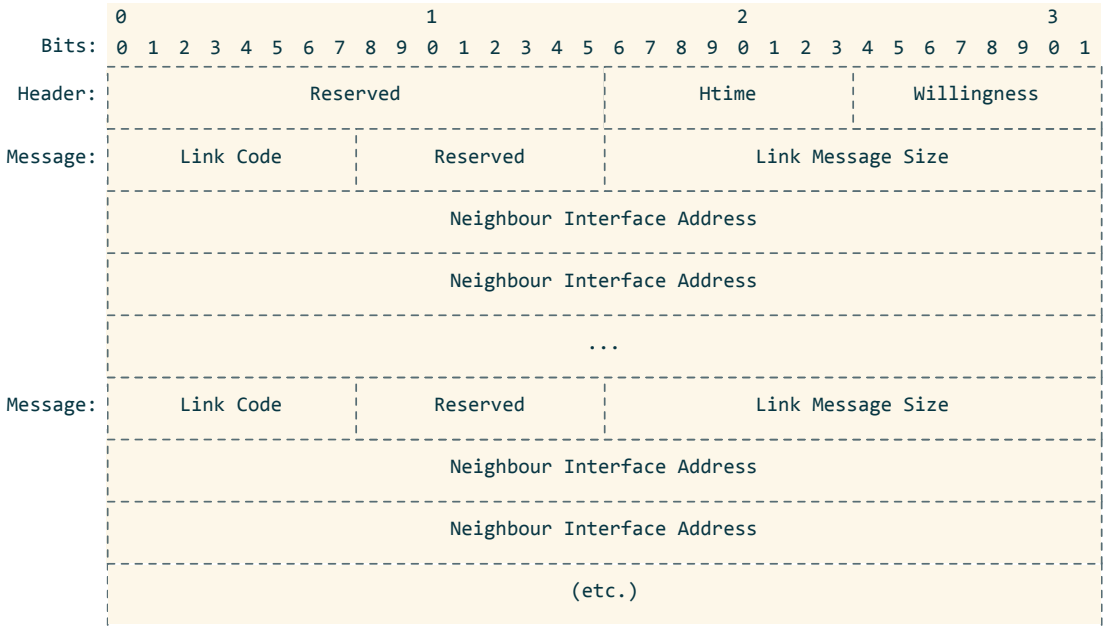


FIGURE 2.5: Format of an OLSR HELLO message.

The message exchange procedure for neighbour discovery is shown in Figure 2.6. This is a somewhat simplified version. Essentially, node `A` sends a `HELLO` message with an empty body. Upon receiving this message, node `B` checks to see if it finds its own address in the message. It cannot, of course, so node `B` registers node `A` as a so-called *asymmetric neighbour*. Node `B` then constructs and sends a new `HELLO` message which includes the address of node `A`, specifying it as an asymmetric neighbour.

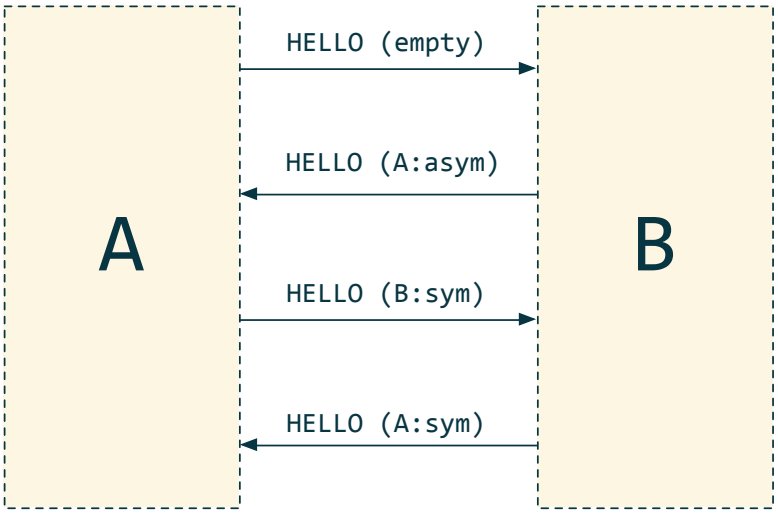


FIGURE 2.6: OLSR neighbour discovery sequence.

When node A receives this, it sees its own address and registers node B as a so-called *symmetric neighbour*. Finally, node A sends a HELLO message back to node B, containing the address of node B. Thus node B receives this and sets node A as a symmetric neighbour.

The 8-bit link code field, shown in Figure 2.7, is used to designate the state of the link between the sender and receiver and also includes the neighbour type information (i.e. symmetric or asymmetric) for the proceeding list of neighbour interface addresses.

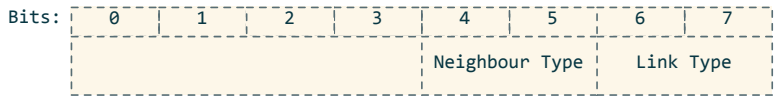


FIGURE 2.7: Format of the OLSR Link Code field.

Neighbour discovery in OLSR is described in greater detail in (Khorov *et al.*, 2012).

2.5.4 Multi-Point Relaying (MPR)

In large networks with many nodes, proactive routing protocols can potentially cause excessive network flooding when forwarding packets to other nodes. OLSR includes a feature to help dissipate this problem, called *multi-point relaying (MPR)*. The main idea behind MPR is to forward packets to a subset of neighbour nodes instead of all neighbour nodes. Figure 2.8 shows an illustration of classical flooding compared with MPR flooding.

The left-hand diagram shows the source (green) node transmitting a message to all its one-hop neighbours via classical flooding. Each neighbour then forwards the message to its own one-hop neighbour (but not back to the original source). This process continues until all nodes have received the message. It is clear from the figure that much of the retransmissions are redundant.

The right hand diagram shows the same source node transmitting a message via MPR. The source node calculates a subset of neighbour nodes (red nodes) to act as multi-point relays. The MPR set is calculated as a subset of symmetric neighbour nodes, chosen so that all two-hop neighbours can be reached through an MPR. Therefore, every two-hop node can be reached through an MPR.

The classical flooding technique in this example requires 24 retransmissions to diffuse a message up to 3 hops. The MPR technique requires only 11 retransmissions, representing a significant reduction.

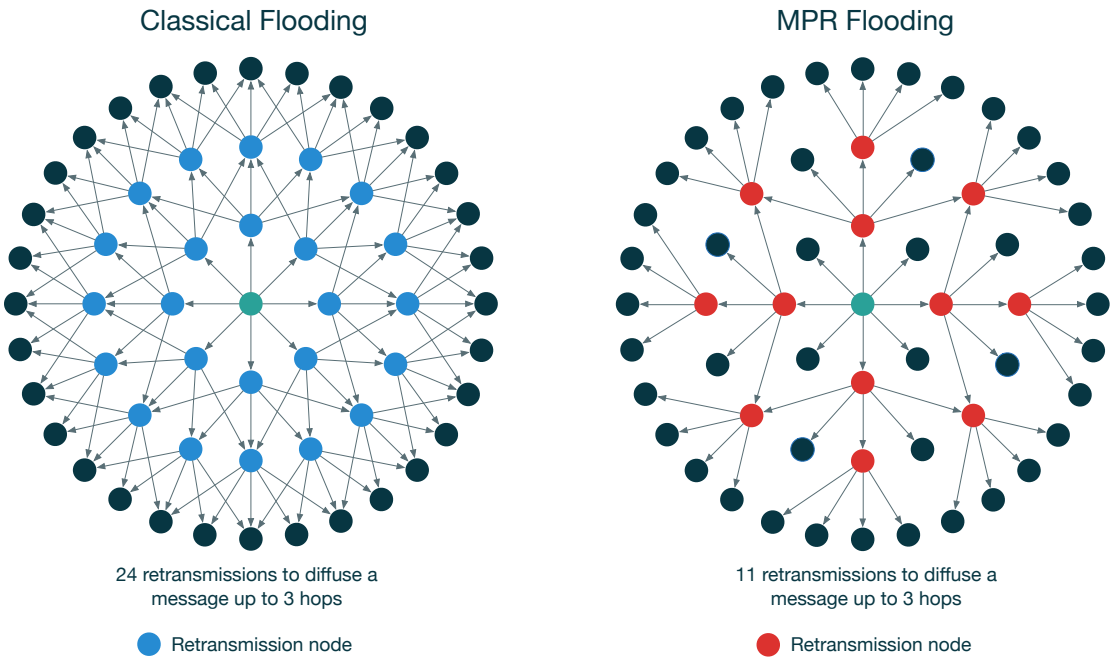


FIGURE 2.8: Comparison of classical flooding with MPR flooding.

2.5.5 Information repositories

As a table-driven protocol, OLSR maintains a variety of tables representing the current state of the network topology. These tables are referred to as *information repositories*. These repositories are updated upon receiving and processing control messages from other nodes. The information repositories are the following:

Link Set. The link set repository maintains the state of interface links between one-hop neighbours.

Neighbour Set. The neighbour set repository records all one-hop neighbours. The information is updated dynamically, based upon the contents of the link set. Asymmetric and symmetric neighbours are both recorded.

2-Hop Neighbour Set. The 2-hop neighbour set repository records all nodes that can be reached via a one-hop neighbour. This can also include nodes registered in the neighbour set.

MPR Set. The MPR set repository records all neighbour nodes that have been selected as MPRs.

MPR Selector Set. The MPR selector set repository records all neighbour nodes that have selected the local node as an MPR.

Topology Information Base. The topology information base repository records all link state topology information (TC messages) received from neighbour nodes.

Duplicate Set. The duplicate set repository contains recently forwarded and processed messages in order not to duplicate those messages.

Multiple Interface Association Set. The multiple interface association set stores information about all neighbour nodes that use multiple interfaces.

Figure 2.9 illustrates these information repositories. Each input message type is processed and stored in its associated information repository. When an output message is required, the relevant repositories are consulted and used to generate the output message.

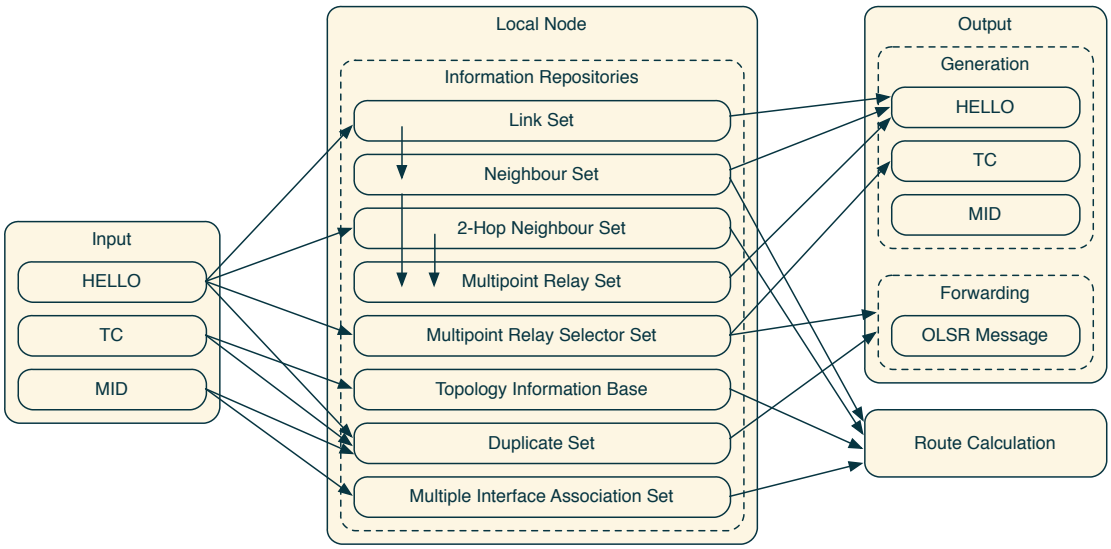


FIGURE 2.9: Relationships between information repositories in OLSR.

2.6 Issues with MANETs

MANETs are a promising technology, and are currently the subject of much research. They are exactly what we are looking for to realise our initial vision of the proximity-based paradigm. However, there are several issues that must be addressed before a framework can be built that makes the best possible use of them.

2.6.1 IP autoconfiguration

To be able to communicate with other devices in a network, one needs an IP address. In traditional Wi-Fi networks, nodes rely on a centralised server, usually an Access Point,

to automatically provide them with an IP address when they wish to connect to the network. This is usually done via the Dynamic Host Control Protocol (DHCP).

However, in the case of MANETs, this centralised mechanism is clearly not appropriate. How might IP autoconfiguration work in MANETs? Who would be in charge of assigning IP addresses to other devices? Several strategies currently exist. An in-depth survey of these strategies is given in (Bernardos *et al.*, 2008). Some of these solutions will be explored during the project implementation.

2.6.2 Name resolution

To bring an element of user-friendliness to networking, nodes are generally assigned unique human-readable names. A familiar example of this are URLs; when a user types `google.com` into their browser, their computer makes a request to a Domain Name System (DNS) server to provide it with (or *resolve*) the actual IP address associated with the URL. This IP address is then used to make a connection.

DNS is a centralised, hierarchical system comprised of many cooperating servers. Again, this centralised solution is not appropriate for MANETs. How are nodes in a MANET to resolve human-readable names for each other? Solutions exist, such as described in (Hong *et al.*, 2007, 2005). These solutions will be explored during the project implementation.

2.6.3 Service discovery

Service discovery allows devices on a network to automatically detect the *services* available on other devices on the same network. These services can be used for a multitude of purposes, such as advertising printing functionality, multimedia capability, and other data sharing services.

Service discovery is traditionally handled via protocols such as Universal Plug and Play (UPnP) and Bonjour. Again, there is no precedent for this on MANETs. Some mechanismns for service discovery in MANETs are described in (Aguilera and López-de Ipiña, 2012, Le Sommer and Mahéo, 2011, Shao *et al.*, 2009). To support and encourage content-rich applications, a solution for the service discovery problem must be implemented.

2.6.4 Geolocation

As mentioned previously, applications which make use of geographic location sharing usually rely on publishing this information to a central server. However, we are interested in sharing geolocation information on a peer-to-peer basis.

Previous approaches to GPS coordinate sharing in ad-hoc networks have tried to modify the OLSR protocol to pass GPS coordinates, such as in (Wang and Zhu, 2012). However, this approach is disruptive to the protocol and hence becomes less cross-compatible.

2.6.5 Summary

If we want to create a useful platform for proximity-based computing, these issues must be resolved. A pragmatic approach might attempt to combine these issues into one unified mechanism. In Chapter 9, implementation of solutions to these issues are described.

Chapter 3

Android

As mentioned in Chapter 1, this project is limited solely to devices running the Android operating system in the first instance. This chapter provides a high-level overview of the Android operating system. Android is a large and complex beast, so this chapter covers only the essential aspects from the viewpoint of a software developer wanting to begin writing applications for the platform. A good introductory text is provided in (Burnette, 2009).

Developed by Google Inc. and the Open Handset Alliance, the Android OS is specifically designed for touch-screen mobile devices such as smartphones and tablets that have limited memory and CPU capacity. It is based on the popular open-source Linux kernel, and is therefore open-source itself.

Android features a rich software development kit (SDK) for developers to use to create applications for the platform. These user-created applications are made available by Google from a centralised store, called the Google Play Store. The Google Play Store currently contains over 1.1 million applications[1].

3.1 Why Android?

As of the third quarter of 2013, Android held an 81% share of the smartphone/tablet market (Wilhelm, 2013) over other operating systems. This represents a 51.3% year-over-year change from the same period in 2012. See Table 3.1 for a detailed breakdown of the market shares and shipment volumes for the top four mobile operating systems.

[1] As of March 25, 2014.

Clearly, in the first instance, targeting Android devices gives us a significant potential user base.

Operating System	3Q13 Shipment Volumes	3Q13 Market Share	3Q12 Shipment Volumes	3Q12 Market Share	Year-Over-Year Change
Android	211.6	81.0%	139.9	74.9%	51.3%
iOS	33.8	12.9%	26.9	14.4%	25.6%
Windows Phone	9.5	3.6%	3.7	2.0%	156.0%
BlackBerry	4.5	1.7%	7.7	4.1%	-41.6%
Others	1.7	0.6%	8.4	4.5%	-80.1%
Total	**261.1**	**100.0%**	**186.7**	**100.0%**	**39.9%**

TABLE 3.1: Top four mobile operating systems, shipments and market share, Q3 2013 (units in millions) (Wilhelm, 2013).

Android has evolved immensely throughout its relatively short life, through several major version releases. These versions and their respective current usage distributions are shown in Table 3.2. When developing applications for Android, it is important to take these distributions into account, to know how many users one can target by supporting a particular version.

Version	Codename	Release date	API Level	Distribution
1.5	Cupcake	20 Apr 2009	3	<0.1%
1.6	Donut	05 Sep 2009	4	<0.1%
2.0/2.1	Eclair	26 Oct 2009	5	<0.1%
2.2/2.2.x	Froyo	20 May 2010	8	1.2%
2.3/2.3.x	Gingerbread	06 Dec 2010	10	19.0%
3.0/3.1/3.2/3.2.x	Honeycomb (tablet only)	22 Feb 2011	13	0.1%
4.0/4.0.x	Ice Cream Sandwich	01 Oct 2011	15	15.2%
4.1.x	Jelly Bean	09 Jul 2012	16	35.3%
4.2.x	Jelly Bean	13 Nov 2012	17	17.1%
4.3	Jelly Bean	24 Jul 2013	18	9.6%
4.4	Kit Kat	31 Oct 2013	19	2.5%

TABLE 3.2: Android versions and distribution (data from Android Market, Mar 2014).

Android versions are backwards-compatible, which means that applications written for an older version will function properly on newer versions of the OS. They are generally not forwards-compatible, however; this means that applications written for a newer version are not guaranteed to work with older versions.

3.2 Anatomy

Android allows hardware-agnostic application development due to its unique abstraction mechanism. This abstraction is organised into five segments. Figure 3.1 represents the structural anatomy of the Android OS. The following sections describe each segment of Figure 3.1.

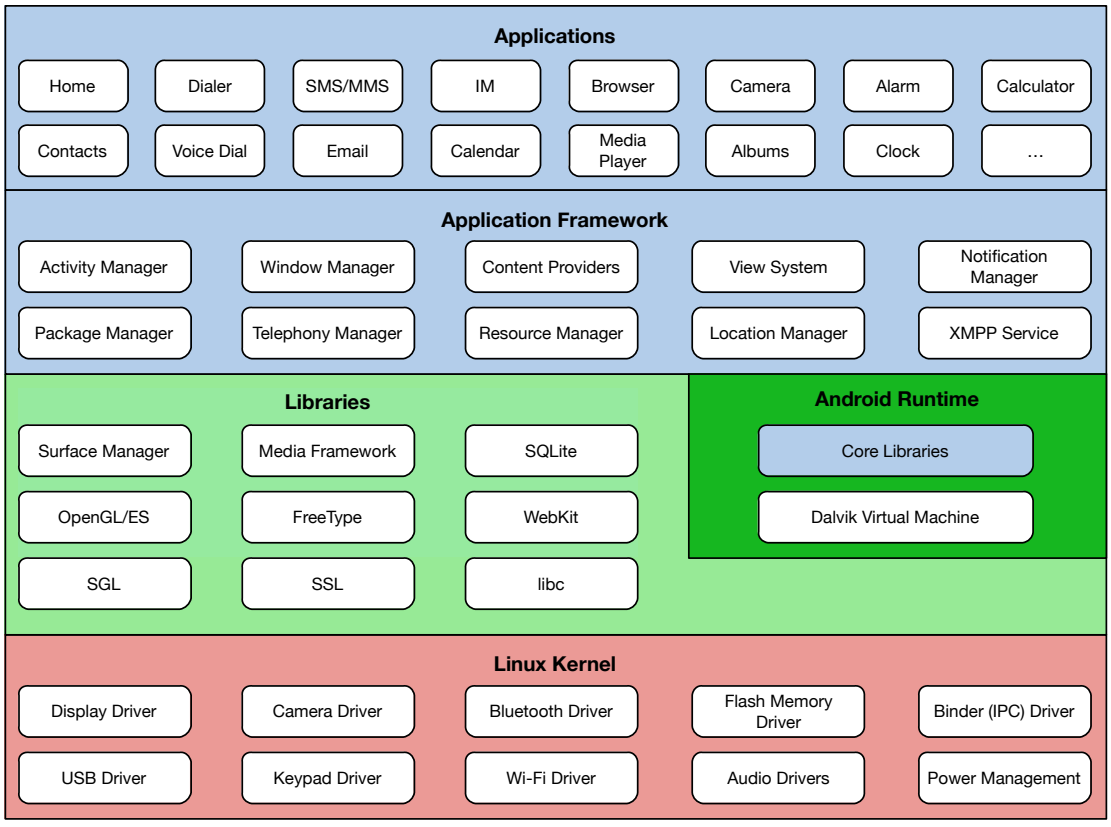


FIGURE 3.1: Android software stack.

3.2.1 Linux kernel

The Android kernel is based on a heavily modified version of the Linux 2.6 kernel. The kernel provides fundamental system functionality, such as memory management, process management and hardware drivers.

The Linux kernel core has been used for a number of reasons. Firstly, Linux is widely known to be proven and robust in terms of threading, memory and process management. Secondly, the Linux permission-based security model is well-tested and effective. Thirdly, of course, Linux is open source. The Android creators wanted a platform that could be freely modified by the open-source community.

However, it cannot be said that Android is a Linux distribution in its own right. Most notably, the differences between Android and Linux include the following:

- Lack of native windowing system (such as GNOME or KDE on Linux);
- Lack of `glibc` (GNU libc) and standard Linux utilities.

3.2.2 Libraries

On top of the kernel, Android contains a set of native libraries. These include a custom version of `libc` named `bionic`, native servers for audio, video, databases, and security libraries. Hardware abstraction is also done at this level.

3.2.3 Android runtime environment

The Android runtime environment is based on the same principles as the Java runtime environment (JRE) in that it consists of two components; a core SDK, and a virtual machine (JVM). However, the core Android SDK is a complete reimplementation of the standard Java SDK. Furthermore, Android does not use the standard JVM, but instead uses a custom virtual machine called the **Dalvik Virtual Machine**. Dalvik is optimized for CPU and memory usage on the embedded devices on which it runs. It has been written so that a device can run multiple Dalvik instances efficiently. Everything above the Android runtime is written in Java.

The Zygote process

There are some processes that are below the application level, and as such do not have their own Dalvik VM. The most important of these processes is the *Zygote*. It is launched when the system first starts up. The Zygote is essentially a "warmed-up" process with all necessary shared libraries pre-loaded. When an application is started, the Zygote is "forked", i.e. a copy is created. This results in a large performance increase, due to the fact that the shared libraries do not need to be copied.

3.2.4 Application framework

This layer provides the main pieces that compose the Android SDK. For example, the Activity Manager keeps track of the currently running activities and their state. Developers have full access to the same framework APIs as the core applications. Developers are free to benefit from the device hardware, run background services, access location information, add notifications to the status bar, set alarms, and more.

The application architecture has been designed with simplification of reuse of components in mind. Applications can publish their capabilities, which may then be made use of by other applications. This mechanism allows built-in applications, such as the email application, to be replaced by user-created versions.

3.2.5 Applications

All Android applications are written in Java, including the stock applications such as the Email, SMS and Calendar applications. Since they are written in Java, they will run on any Android device hardware-independently. Android applications are compiled into a compressed version of Java bytecode, called `dex` format.

3.3 Android applications

There are two significant differences between Android applications and standard Java applications. Firstly, each Android application runs in its own process, and is given its own Dalvik VM instance. Android is a single-user OS; this means that each application runs as a separate user, allowing sandboxing of applications while still maintaining the Linux security features. The kernel can protect files and memory for each application without additional effort.

Secondly, Android applications do not have a single point of entry. They are composed of a collection of components that are potentially reusable by other applications. Compiled code and resources are packaged into a special file, called an APK file (.apk extension). This APK file is installed on the device. An APK file holds an "application", which can have many components. These components may be:

- **Activities** and their sub-components **Fragments** which can broadly be thought of as regular processes;

- **Content Providers** which essentially provide data to other applications in a structured way;
- **Services** which act like system daemons, and can be interacted with in a similar way to regular system calls;
- **Intents** which are similar to regular system events, and;
- **Broadcast Receivers** which receive and handle those intents.

The following sections describe these different application components in detail.

3.4 Activities

An activity comprises a visual user interface for a single task. For example, an activity might display a gallery of photographs taken from the user's camera, or present a list of contacts and their telephone numbers.

An application might consist of a single activity, but most complex applications consist of several activities. Typically, one of these activities is designated as the "main" activity, i.e. the one that should be first presented to the user when the application is opened. An activity can essentially be in one of three states:

- It is **active** or **running** when it has foreground focus on the screen, and is therefore the focus for the user's actions.
- It is **paused** if it is still visible to the user but has lost focus. Another transparent activity, not covering the entire screen, lies on top of it. An activity in a paused state is alive, but in extremely low-memory situations it may be killed by the system.
- It is **stopped** if it is obscured completely by another activity. It still maintains all state information, but it is no longer visible to the user. A stopped activity will often be killed by the system when memory is tight.

See Figure 3.2 for a detailed overview of the lifecycle of an Android activity.

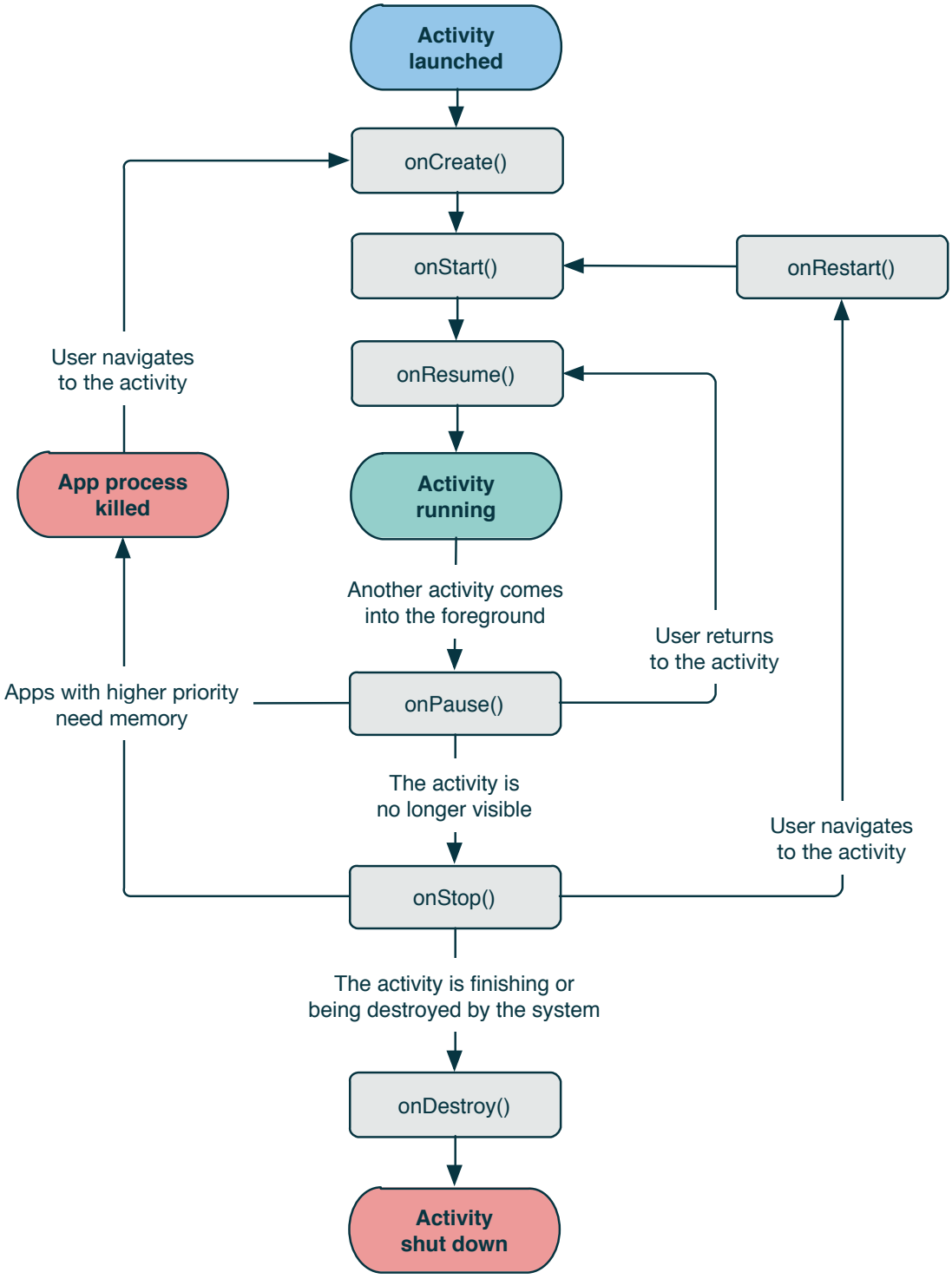


FIGURE 3.2: Android activity lifecycle.

Fragments

Fragments are designed to be modular, reusable portions of user interface which have their own lifecycle and receive their own input events. They can be thought of as "sub-activities" in that an activity can be made up of multiple fragments, and fragments cannot live independently of activities.

3.5 Services

A service does not have a visual user interface. Services run for an indefinite period of time in the background. For example, a service might play music in the background, while the user attends to other matters. Another common usage for services is fetching data over the network, or performing long-running calculations and providing the result to activities that need it when complete. There are two types of service:

Started service. A started service is acquired when an application calls the `startService()` method of a `Context` object (available to application components such as Activities). Started services run indefinitely in the background until they have finished, and do not interact with the caller. If the service is not already running, it is initially created. If the service was previously started, the call to `startService` does nothing.

Bound service. A bound service is acquired by calling the `bindService()` method of a `Context` object. Bound services are capable of performing inter-process communication (IPC) with one or more application components. The service is created when the first component binds to the service, and is destroyed when the last component unbinds from the service.

See Figure 3.3 for an overview of the lifecycle of an Android service.

3.6 Broadcast receivers

A broadcast receiver is a component that has the sole purpose of receiving and reacting to "broadcast intents". Many of these broadcasts originate from the Android system itself. For example, announcements that that the battery is low, the timezone has changed, that the user changed a language preference, or that a picture has been taken.

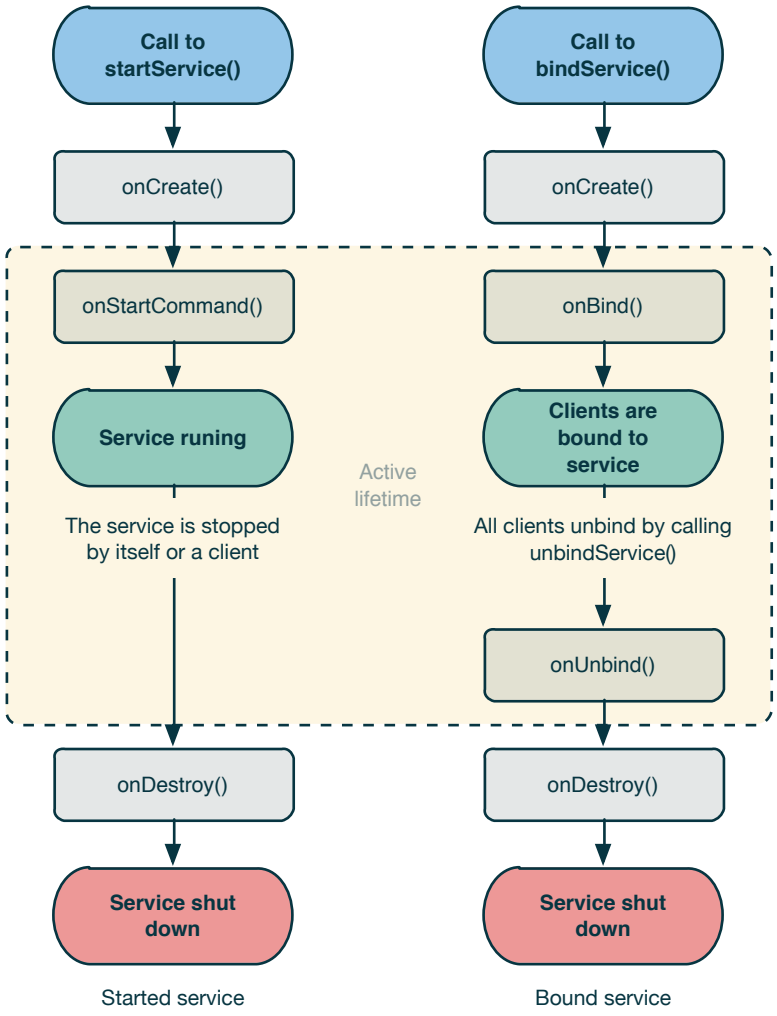


FIGURE 3.3: Android service lifecycle.

Applications can also send broadcasts themselves. For example, an application might want to let other applications know that some data has finished downloading and is available for use. An application can have an arbitrary number of broadcast receivers to receive announcements it considers important.

3.7 Content providers

A content provider allows a specific set of data from an application to be accessible from other applications. The data can be stored in a database, on the file system, or in any other logical manner. Applications can retrieve and store data via a standard set of methods. Applications use `ContentResolver` objects and their associated methods to retrieve data from other applications.

3.8 Intents

Content providers communicate via direct access requests from content resolvers. Activities, services and broadcast receivers, however, communicate via asynchronous messages called `Intents`.

An intent is a message that can convey any meaning that a developer requires. For example, it might represent a request for an activity to show the user a dialog to edit some text, or it might present an image to the user. Intents that are sent in broadcast mode are those received by broadcast receivers. For example, an application might broadcast to interested parties that Wi-Fi has been turned off.

3.9 Ad-hoc mode on Android

Ad-hoc Wi-Fi mode was identified as the mechanism of choice for our proximity-based framework in Chapter 2. However, ad-hoc mode is not supported by default on Android devices; Google have chosen to disable it in their stock kernels. This section explains the possible reasons behind this decision, before going on to explain how to acquire ad-hoc mode on the devices that were used for this project.

3.9.1 Issue #82

Issue number 82 on the Android issue tracking page was the first to document the lack of ad-hoc support. It states the following:

> "The ability to support wifi ad hoc networking will allow use cases currently not supported by any non-wifi phone and even on the iPhone ad hoc networks are not officially supported. This could be a significant differentiator.
>
> The use cases I have in mind include using wifi to message friends on a plane or bus where you couldn't get a seat next to each other."
>
> (Saville, 2008)

This issue was reported in January 2008, and has been starred by 6006 people[2]. It received 1655 comments before being restricted to only allow Google developers to add comments. This is clear evidence of the pressure that Google has upon them to enable ad-hoc mode in the stock kernel.

[2] As of February 2014

It is theorised that Google has not yet responded to this issue because of some internal plan to support ad-hoc networking in a different, undisclosed way in the future.

Luckily, not all devices use the stock kernel provided by Google; some device manufacturers, such as Samsung, provide modified kernels, some of which have ad-hoc mode enabled. See Table 3.3 for an overview of the devices that were used in this project and their custom kernel requirements.

Device	*Requires custom kernel*
Samsung Galaxy S4	Yes
Samsung Galaxy Tab 10.1	No
HTC Wildfire	Yes

TABLE 3.3: Custom kernel requirements.

For those devices that do require a custom kernel, there are a number of steps that must be performed. Once the appropriate Android kernel source has been acquired, the steps are as follows:

Enable Wireless Extensions (WEXT). The wireless interface driver must have WEXT enabled in order to be able to set the interface to ad-hoc (IBSS) mode. This is done by editing the kernel build configuration before compilation (using `make menuconfig`).

Add ad-hoc mode interface support. The driver is set to ignore ad-hoc networks by default. This is fixed by adding ad-hoc mode (`NL80211_IFTYPE_ADHOC`) to the list of supported interface modes in the file `/drivers/net/wireless/bcmdhd/wl_cfg80211.c` within the kernel source, as shown in Listing 3.9.1.

```
1  static s32 wl_setup_wiphy(struct wireless_dev *wdev, struct device *sdiofunc_dev)
2  {
3      // ...
4      wdev->wiphy->interface_modes = BIT(NL80211_IFTYPE_STATION)
5          | BIT(NL80211_IFTYPE_ADHOC) /* This line added */
6          #if !(defined(WLP2P) && defined(WL_ENABLE_P2P_IF))
7          | BIT(NL80211_IFTYPE_MONITOR)
8          #endif
9          | BIT(NL80211_IFTYPE_AP);
10     // ...
11 }
```

LISTING 3.9.1: Adding ad-hoc support to the Wi-Fi driver.

Disable packet filtering. By default, the driver is set to filter UDP packets when the screen is turned off. This prevents the routing protocol from working properly. Disabling this filtering requires editing the file `/drivers/net/wireless/bcmdhd/Makefile` within the kernel source and removing the `-DPKT_FILTER_SUPPORT` flag from the `DHDCFLAGS` variable.

Once these modifications have been made to the kernel source, it can be compiled and installed on the device.

3.9.2 Obtaining root access

Regardless of the kernel modification requirements, it is always necessary to have root access to the device. This is due to the need to modify system settings, including reconfiguring the wireless interface into ad-hoc mode. There are various methods of obtaining root access, and the methods differ by device. Details of the rooting process for our devices is given in Chapter 9.

3.10 Summary

This chapter provided a high-level overview of the Android OS, in enough detail to enable a new developer to begin writing applications for the platform. In the context of our proximity-based framework, we identified the modifications that must be made to the OS. Following this, we will be ready to begin work on designing and implementing the framework.

Chapter 4

Related work

This chapter outlines some of the existing work that has been done in the project domain, and assesses the relevance and similarity to this project.

4.1 Project SPAN

Project SPAN (Thomas *et al.*, 2012) is an API and management application for MANETs. It is based around an Android application called MANET Manager, which is used to manually configure network parameters. The MANET manager provides a modular routing protocol framework and an interface for application developers to interact with the network.

The contributors to the SPAN project have done great work in the area of MANETs on Android. Huge credit goes to them for their effort. Some inspiration for this project has been taken from the work of the SPAN developers. However, while actively developed, the MANET Manager API does not provide any mechanisms for tasks such as automatic IP configuration, service discovery or name resolution. Also, it should not be necessary for application developers to require their users to install an additional application for their own to function; the functionality should exist solely as a development library.

4.2 Serval Project

The Serval Project (Gardner-Stephen and Palaniswamy, 2011) aims to bring infrastructure free mobile communication to people in need, such as during crisis and disaster

situation when vulnerable infrastructure such as cellular network towers and mains electricity is cut off. It requires devices that are capable of ad-hoc Wi-Fi to form a mesh network. It then handles the routing and connections between the nodes (devices) that make up the network. The routing protocol the app uses is custom and behaves in a similar fashion to OLSR. It is made up of two components: the Batphone and the Serval DNA. The Batphone is the user interface and is composed of a conventional Android app written in Java and XML and built using the Android SDK. The Serval DNA is the core networking, encryption and file sharing component; it is a daemon written in GNU C and built using the Android NDK.

Although the Serval Project is licensed as Open Source software there is no developer documentation available that outlines how to build custom applications utilizing the Serval DNA. Therefore, the Serval Mesh is not intended for use with custom applications that rely on a MANET, but intended to be as-is software with no easy to access APIs.

4.3 Commotion Wireless

Commotion Wireless (Reynolds *et al.*, 2011) is an open-source communication tool that uses mobile phones, computers, and other wireless devices to create decentralised mesh networks. We share the common goal of decentralisation, however we are focusing more on the creation of a framework to allow people to develop applications rather than just providing basic communication ability.

4.4 Open Garden

Open Garden (Open Garden, 2014) is another free piece of software which creates an overlay mesh network using Bluetooth and Wi-Fi connections across a range of mobile devices, from smartphones to tablets to laptops and desktops. The primary goal of Open Garden is to provide Internet connection to devices without a Wi-Fi access point. While this goal is useful, ours is more general than simply providing Internet access.

4.5 AllJoyn

AllJoyn (Qualcomm, 2014) is an open source project which provides a universal framework for Android, iOS, Windows, Mac OSX, and Linux that allows software across multiple devices to interact using a single dynamic network. The SDKs provided by the

AllJoyn project allow developers to create applications that utilize a distributed software bus. This bus is made through the cooperation of several bus daemons running on each device in the network. The bus is an implementation of the D-Bus wire-protocol, an open-source inter-process communication (IPC) system, allowing multiple, concurrently-running applications to communicate with one another.

The bus itself is formed in an ad-hoc fashion based on proximal discovery. The bus and protocol for AllJoyn are transport independent. That is, a daemon that is connected to the global bus using a Bluetooth connection may send data to a daemon connected to the global bus using a Wi-Fi connection. Applications on each device send data directly to the bus daemon. In turn, the bus daemon routes and forwards the data over the global bus to the recipient daemon(s).

However, there is a limitation to the AllJoyn project which make it a less than ideal candidate for implementing a MANET; AllJoyn currently only supports connections formed using Bluetooth or infrastructure Wi-Fi. As mentioned previously, infrastructure Wi-Fi cannot be used to create MANETs as it requires static access points. As for Bluetooth, the physical range of class 1 devices is at most 100 meters (usually 20-30 meters) and of class 2 devices 30 metres (usually 5-6 metres). This limited range makes Bluetooth networks a poor candidate for MANETs.

4.6 Location-based applications

Location-based applications are another category that have had several successes in the mobile arena. These applications have usually relied on traditional infrastructure networks such as Wi-Fi or cellular towers for timely publication of the location information to a centralised server, where the location information is determined by the users current GPS location. Some location-based applications, such as Foursquare (Foursquare, 2014), have traditionally oriented around the mobile device user arriving within proximity of a particular fixed location, while others have also oriented around the location of other nearby devices.

4.7 Overview

The findings from our field study show that the existing work on proximity-based wireless networks is either application-specific, i.e. based around a specific use case; or it is

not quite functionally adequate to meet our requirements for a useful framework for proximity-based computing.

Chapter 5

Project scope

Following the background research into how and why the proximity-based paradigm might be realised, this chapter outlines the scope of the project in terms of objectives, and gives details on the project management and prototyping decisions that were made before beginning the process of development.

5.1 Hypothesis

Is it possible to build an API framework so that application developers can make use of Proximity-Based Functionality (PBF), without an Internet connection, to build new features into their applications or create entirely new applications? Furthermore, is it possible to design the framework to function effectively on the Android platform, regardless of the physical device specifics and transport mechanics?

5.2 Project goals

The goals for this project are twofold. The primary goal is to develop a general, reusable *framework* for PBF. The secondary goal is to prove the usability and effectiveness of the framework by implementing an application that takes full advantage of the framework. These two goals are explained in greater detail below.

5.2.1 *Proxima* – A general framework for proximity-based functionality

This project aims to create a novel framework for mobile proximity-based peer-to-peer networking, which addresses the limitations of the existing solutions and provides all of

the necessary requirements for PBF. The framework will allow the development of novel applications or the modification of existing applications which may benefit from PBF. Due to the myriad of available mobile devices and their varying levels of compatibility and interoperability, we will initially build on the Android platform, whilst still aiming to support as wide a range of devices as possible.

The framework, named Proxima, will be delivered as an Application Programming Interface (API) for use with smartphones running the Android operating system. The supported Android versions will be 1.6 and above. The Android operating system is outlined in Chapter 3. The framework will abstract the device specifics and transport mechanics away from the application developer. The core functionality of the Proxima framework will be written in Java using the Android SDK.

5.2.2 *TuneSpy* – A proximity-based real-time music sharing application

The secondary goal of the project is to design and develop an Android application which both demonstrates the usage of, and serves as a testing platform for, the Proxima framework. The example application, named TuneSpy, will be a realisation of the train journey scenario described in Section 1.1.1 and will allow real-time music streaming between Android devices. The TuneSpy application is explained in detail in Chapter 10.

5.3 Tools

The client-facing portion of the Proxima framework will be written in Java using the Android Java SDK. The entire framework will be compatible with Android 1.6 (codename Donut, API level 4), and will deployed as an Android Library Project. Development will be done using the Eclipse IDE (Kepler) with the Android ADT plugin on a machine running Ubuntu 10.04 (Lucid Lynx) x86_64 with kernel version 2.6.32.

The project will be developed and tested on the following devices:

- Samsung Galaxy S4, running Android version 4.3 (Jelly Bean)
- Samsung Galaxy Tab 10.1 running Android version 3.2 (Honeycomb)
- HTC Wildfire running Android version 2.2.3 (Froyo)

5.4 Project management

This section outlines the processes and methodologies that have been/will be implemented during the course of the lifecycle of the project.

5.4.1 Development methodology

Complex team-based methodologies such as Scrum will not be appropriate for this project, due to its solo nature. Instead, a simple iterative methodology will be used whereby requirements analysis, design, implementation and testing will be done iteratively and in parallel. Figure 5.1 gives an illustration of this iterative idea.

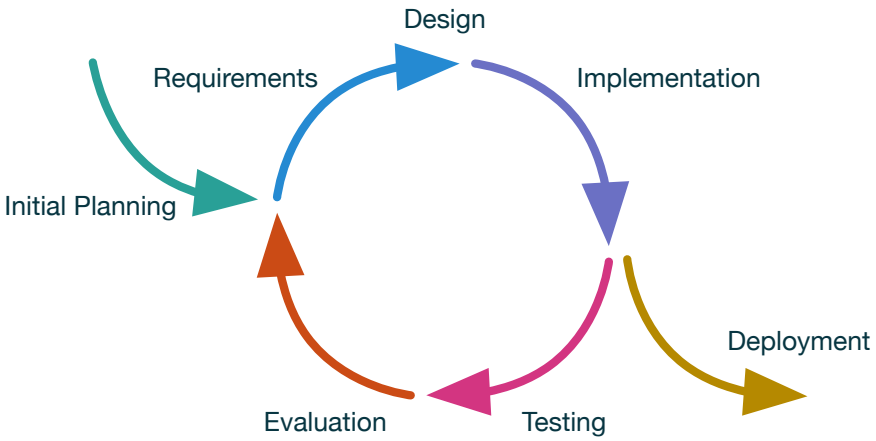


FIGURE 5.1: Illustration of an iterative development methodology.

5.4.2 Project timeline

Throughout the lifetime of the project, this has been a live document; i.e., it has changed and grown according to the requirements of the project and the understanding of the author. Table 5.1 outlines the various stages of the project and the approximate dates on which they were undertaken.

5.5 Planned iterations and prototypes

Following the initial background research and requirements analysis, development work is planned into three distinct iterations. These iterations are described in the following subsections.

Stage	**Planned period**
Initial research and feasibility study	September 2013
Initial planning and requirements analysis	October 2013
Iteration 1: design, test and develop Prototype 1	November 2013
Iteration 2: review design, test and develop Prototype 2	December 2013
Iteration 3: review design, test and develop Prototype 3	January 2014
Final development and testing, report preparation	February 2014
Complete report	March 2014

TABLE 5.1: Project timeline.

5.5.1 Iteration 1

The first iteration will lay the foundation of the implementation. The fundamental tasks such as enabling ad-hoc mode, running the OLSR routing daemon and interfacing to it from Java will be completed. This will form the first prototype.

5.5.2 Iteration 2

The second iteration will shape and form the framework into a coherent, cohesive, reusable set of components. This will comprise the second prototype. It is expected that development on the example application will begin at this point, as the client-facing API will be defined during this iteration.

5.5.3 Iteration 3

The third iteration should provide the ability to set the device name, and to retrieve the names of neighbour devices. These will be incorporated into a third prototype. If time allows, additional functionality such as GPS location dissemination and service discovery will be implemented.

Detailed descriptions of the specific implementation tasks of each of these iterations will be given in Chapter 9.

Chapter 6

Requirements analysis

In Chapter 1, we informally identified a set of requirements that a framework should have in order to realise the proximity-based paradigm. In Chapter 2, we analysed the potential technologies that could be used to achieve the required underlying network connectivity and topology, and we selected the most suitable (OLSR routing over ad-hoc Wi-Fi). Now that these decisions have been made, we can proceed with the task of designing a framework from the point of view of those who will use it.

This chapter attempts to elicit a set of formal requirements for the software in context, using standard requirements analysis techniques. Requirements analysis is the process by which customer needs are understood and documented, and express *what* is to be built but not *how* it is to be built. These requirements serve as a contract between the client application developer and the library developer. They will serve to specify the project goals and plan development iterations. They will also be used as a basis for developing test plans in Chapter 8. The process of requirements elicitation comprises the following activities, in order:

- **Stakeholder identification**. It is necessary to identify the stakeholders of the system, to know who we are actually building software for.
- **Use case analysis**. Following stakeholder identification, it is necessary to define how those stakeholders are going to use the system.
- **Functional requirements**. There may be additional system requirements that are not direct stakeholder use cases, and we must identify them.

- **Non-functional requirements**. To be able to judge the overall operation of a system as a whole, we must specify its non-functional requirements i.e. the general properties of the system as opposed to its behaviours.
- **MoSCoW prioritization**. To aid workflow management, we must prioritise the identified requirements. This is done using using the standard MoSCoW method.

The next few sections describe these steps in greater detail.

6.1 Stakeholder identification

When assessing the requirements of a system, it is crucial to identify the interested parties, or *stakeholders*, who will interact with the system. For this particular system, the *customer* is the *application developer* who would like to implement proximity-based functionality in his/her application. It seems at first that that the application developer will be the sole stakeholder, because of the fact that the system is a framework designed for developers to use in order to create their own applications and is not a product that end-users (i.e. device owners) will use directly. It is during the design process of the specific application using the framework that the *end-user* should be identified as a stakeholder.

However, since the framework will potentially be shared by multiple applications on a single device, and that device will likely require some global configuration to be set by the end-user, we must include the end-user as a stakeholder at the framework level. This will become more clear throughout the following sections.

6.2 Use case analysis

In this section, we identify the core use cases from the viewpoint of the stakeholders of the system, and provide an overview of the previously identified use cases using standard UML notation. We previously identified two stakeholders; the application developer and the end-user.

Figure 6.1 is an instance of a so-called *UML use case diagram* (Rumbaugh *et al.*, 2004). The stick figures represent stakeholders, and the enclosing box represents the system as a whole. The ovals represent the individual use cases. The `RetrieveNeighbourDetails` use case has been further subdivided into four categories; IP address,

GPS location, device name, and device service data retrieval. These actions *"extend"* the parent use case, as they are related to but functionally separate from the parent.

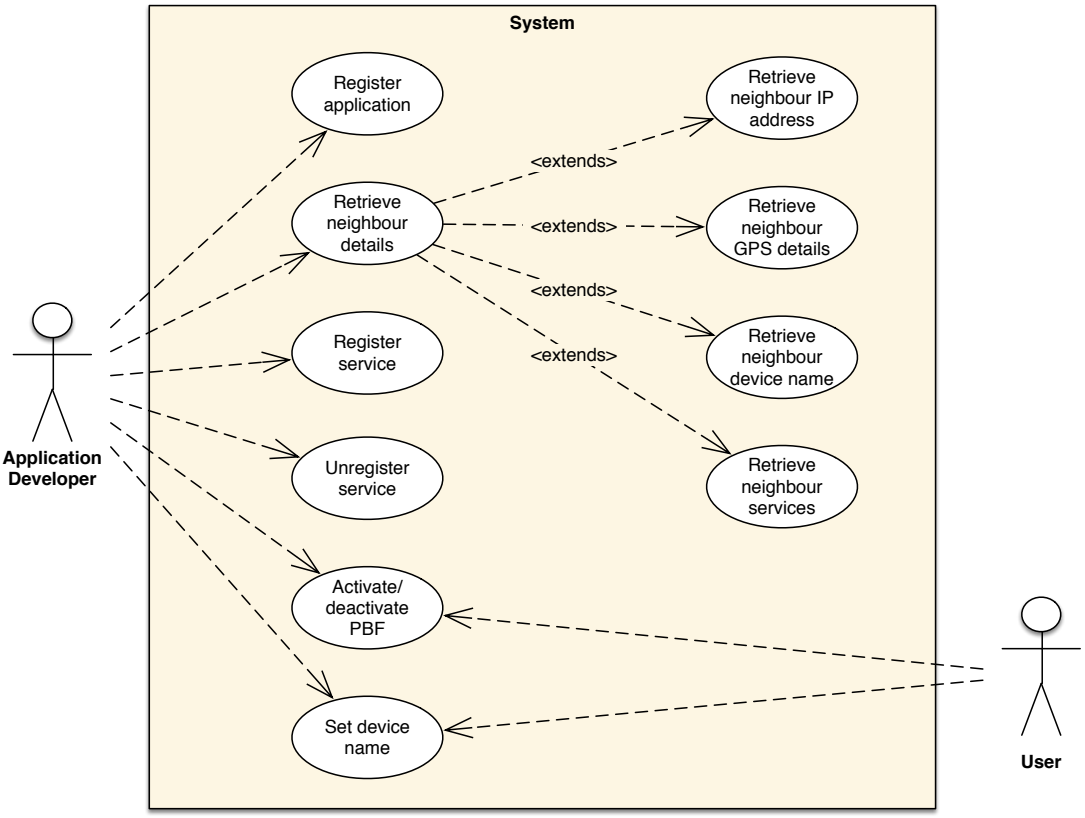


FIGURE 6.1: UML use case diagram.

Table 6.1 through Table 6.7 describe each of these use cases in detail. Each use case has one or more participating stakeholders, or *actors*.

Use Case Identifier	`RegisterApplication`
Description	The application developer must be able to register his/her application with the `Framework`, in order to receive future updates regarding neighbour status and perform other framework-related operations.
Participating Actor	`ApplicationDeveloper`
Entry condition	The device must have root access enabled.
Flow of Events	1. `ApplicationDeveloper` requests to register an application with the `Framework`. 2. `Framework` performs necessary registration tasks. 3. `Framework` notifies `ApplicationDeveloper` upon successful registration.
Alternative Flow	If root access is not enabled on the device, `Framework` notifies `ApplicationDeveloper` of the registration failure.
Exit Condition	The application is registered with the framework and is ready for further interaction.

TABLE 6.1: Use case `RegisterApplication`.

Use Case Identifier	`ActivatePBF`
Description	The application developer must be able to request `Framework` to activate proximity-based functionality (PBF). The `User` may have some interaction or control over this activation.
Participating Actor	`ApplicationDeveloper`, `User`
Entry condition	The device must have root access enabled.
Flow of Events	1. `User` (possibly) requests to activate PBF via the client application. 2. `ApplicationDeveloper` requests `Framework` to activate PBF. 3. `Framework` performs necessary setup and activation tasks. 4. `Framework` notifies `ApplicationDeveloper` upon successful PFB activation.
Alternative Flow	If root access is not enabled on the device, or if some other error occurs, `Framework` notifies `ApplicationDeveloper` of the activation failure.
Exit Condition	PBF is activated and ready to be used.

TABLE 6.2: Use case `ActivatePBF`.

Use Case Identifier	`DeactivatePBF`
Description	The application developer must be able to request `Framework` to deactivate proximity-based functionality (PBF). The `User` may have some interaction or control over this deactivation. Upon deactivation, the previous device Wi-Fi state should be restored.
Participating Actor	`ApplicationDeveloper`, `User`
Entry condition	The device must have PBF activated.
Flow of Events	1. `User` (possibly) requests to deactivate PBF via the client application. 2. `ApplicationDeveloper` requests `Framework` to deactivate PBF. 3. `Framework` performs necessary deactivation tasks and restores the previous device Wi-Fi state. 4. `Framework` notifies `ApplicationDeveloper` upon successful PFB deactivation.
Alternative Flow	If PBF is not active, or if some other error occurs, `Framework` notifies `ApplicationDeveloper` of the activation failure.
Exit Condition	PBF is deactivated and can no longer be used.

TABLE 6.3: Use case `DeactivatePBF`.

Use Case Identifier	`RegisterService`
Description	The application developer must be able to register one or more unique local services for use by neighbour applications.
Participating Actor	`ApplicationDeveloper`
Entry condition	The application must have been previously registered with the framework.
Flow of Events	1. `ApplicationDeveloper` requests to register a service. 2. `Framework` registers the service. 3. `Framework` notifies `ApplicationDeveloper` upon successful registration.
Alternative Flow	If the application has not been previously registered, `Framework` notifies `ApplicationDeveloper` of the failure.
Exit Condition	The service is successfully registered and is available to neighbours.

TABLE 6.4: Use case `RegisterService`.

Use Case Identifier	`UnregisterService`
Description	The application developer must be able to unregister a previously registered service.
Participating Actor	`ApplicationDeveloper`
Entry condition	The application must have been previously registered with the framework.
Flow of Events	1. `ApplicationDeveloper` requests to unregister a service. 2. `Framework` unregisters the service. 3. `Framework` notifies `ApplicationDeveloper` upon successful service unregistration.
Alternative Flow	If the application has not been previously registered, or the service has not been previously registered, `Framework` notifies `ApplicationDeveloper` of the failure.
Exit Condition	The service is successfully unregistered and is no longer available to neighbours.

TABLE 6.5: Use case `UnregisterService`.

Use Case Identifier	`RetrieveNeighbourDetails`
Description	The application developer must be able to request the `Framework` for details of the current neighbours at any time. These details will include the device name and IP address, and GPS coordinates. Since it may take some time to retrieve the neighbour details, this operation should happen asynchronously, i.e. `ApplicationDeveloper` will be able to perform other tasks while waiting for `Framework` to respond to the request.
Participating Actor	`ApplicationDeveloper`
Entry condition	The application must have been previously registered with the framework.
Flow of Events	1. `ApplicationDeveloper` requests neighbour details. 2. `Framework` retrieves neighbour details. 3. `Framework` notifies `ApplicationDeveloper` upon successful retrieval and returns the neighbour details.
Alternative Flow	If the application has not been previously registered, `Framework` notifies `ApplicationDeveloper` of the retrieval failure.
Exit Condition	None.

TABLE 6.6: Use case `RetrieveNeighbourDetails`.

Use Case Identifier	`SetDeviceName`
Description	The user of a framework application must be able to specify a desired name for their device, which will be visible to neighbours. The user should be able to specify this name from any framework application, therefore the application developer must defer this functionality to the framework itself.
Participating Actor	`User`, `ApplicationDeveloper`
Entry condition	The application must have been previously registered with the framework.
Flow of Events	1. `User` requests to set the device name within an application. 2. `ApplicationDeveloper` requests `Framework` to set the device name. 3. `Framework` displays an input box to `User` where he/she may enter the desired device name. 4. `User` submits the desired device name. 5. `Framework` notifies `ApplicationDeveloper` that the device name change was successful.
Alternative Flow	If `User` decides to cancel the device name change operation, `Framework` will notify `ApplicationDeveloper` of this.
Exit Condition	The device name is successfully changed and will be visible to neighbours.

TABLE 6.7: Use case `SetDeviceName`.

Use Case Identifier	`RetrieveNeighbourServices`
Description	The application developer must be able to request the `Framework` for details of the services currently available on the network at any time.
Participating Actor	`ApplicationDeveloper`
Entry condition	The application must have been previously registered with the framework.
Flow of Events	1. `ApplicationDeveloper` requests service details. 2. `Framework` retrieves service details from neighbour devices. 3. `Framework` notifies `ApplicationDeveloper` upon successful retrieval and returns the service details of each neighbour device.
Alternative Flow	If the application has not been previously registered, `Framework` notifies `ApplicationDeveloper` of the retrieval failure.
Exit Condition	None.

TABLE 6.8: Use case `RetrieveNeighbourServices`.

6.3 Functional requirements

There are some behaviours that the system is required to perform, which cannot be classed as use cases in the formal sense because stakeholders are not directly involved with their operation. They are lower-level requirements, and can be considered somewhat like "building blocks" for the higher-level use cases. These functional requirements are summarised as follows:

Automatic connection. The system should have the ability to automatically detect and connect to other devices in the network without a manual setup procedure like Bluetooth.

Automatic configuration. The system should configure itself on the network automatically. This involves automatically negotiating IP addresses within the network, amongst other configuration tasks. In other words, there should be as close to zero user configuration as possible.

Security. Ideally, connections between devices should use some form of authentication and encryption mechanism.

6.4 Non-functional requirements

The completed system should meet the following non-functional requirements:

Compatibility. The system should support Android version 1.6 and above. This covers 99.9% of active devices (see Table 3.2).

Speed. The system should be capable of transferring data between devices at a high enough speed to allow complex high-bandwidth applications to function adequately.

Range. The system should be able to detect and connect to devices within a wide proximity.

Scalability. The system should scale and work well with large numbers of networked devices, and also with large numbers of client applications simultaneously on a single device.

Deployability. The system should be lightweight; it should have minimal overhead for the application developer, in terms of size and runtime cost. This will make it easier for developers to deploy the system into their applications.

Reliability. The system should behave the same on different devices in a reliable and predictable manner.

Well-documented. The system should have a well-documented API, and also provide detailed additional documentation about how to use the system, including any caveats.

Open source. The system should be freely available for others to use and modify.

6.5 MoSCoW task prioritisation

MoSCoW (DSDM, 2014) is a simple but effective task prioritisation method. It identifies four priority levels; *MUST* (these tasks must be completed, otherwise the system is considered a failure), *SHOULD* (these tasks should be completed, if at all possible), *COULD* (these tasks could be completed, if it does not affect anything else), and *WONT* (these tasks will not be completed this time, but they may be in the future). Table 6.9 is a set of MoSCoW priorities for the use cases, functional and non-functional requirements of the system.

Task	*Priority*
Register application	**MUST**
Activate/deactivate PBF	**MUST**
Retrieve neighbour device name & IP address	**MUST**
Set the device name	**MUST**
Connect to other devices automatically	**SHOULD**
Automatically configure network	**SHOULD**
Register/unregister local services for discovery	**COULD**
Retrieve neighbour services	**COULD**
Retrieve neighbour GPS coordinates	**COULD**
Implement authentication and encryption	**WONT**

Table 6.9: MoSCoW task prioritisation.

Chapter 7

Design

This chapter describes the analysis and design process that was undertaken during the lifecycle of the project. As mentioned in Chapter 5, an agile iterative development methodology has been used throughout the project, therefore the design documents in this chapter have evolved and been refined iteratively as both the project and the author's understanding have progressed.

The specific design approach used is the so-called Object-Oriented Analysis and Design (OOAD) approach (Rumbaugh, 2003). This approach is comprised of several stages:

- Analyse and identify the classes of objects in the system;
- Organise the objects into a logical structure and define their static relationships;
- Define the dynamic interaction between the objects;
- Define the dynamic interaction between multiple system instances.

Following the first iteration of this process, a prototype was developed and tested. A total of two prototypes were created before the final version, ergo three iterations of the design/code/test sequence were completed. Details of these prototypes were previously given in Chapter 5. Testing specifications are given in Chapter 8.

7.1 Class identification

In this section, we identify the candidate classes of our system. This is done via textual analysis of the use cases that were identified in Chapter 6.

Class-Responsibility-Collaboration (CRC) (Beck and Cunningham, 1989) is a technique for identifying the candidate classes of an object-oriented system. It involves the creation of CRC *cards*, each of which represent a single candidate class. A CRC card is composed of three sections:

- **Class**: The name of the candidate class. This name should reflect the essential purpose of the class.
- **Responsibilities**: The stereotypical behaviour of the class. The information that the class is responsible for maintaining, and the functions that the class performs.
- **Collaborations**: Other classes in the system that this class will interact with.

There follows a single CRC card for each class that has been identified on the first design iteration. Note that the final class model, given in Figure A.7, differs slightly from the classes given here due to the refinement of the structure over the course of the development iterations.

Class Name	`ProximityManager`
Responsibilities	• Handle requests from client applications. • Manage client application connections to the background service. • This class should be the main entry point into the framework API.
Collaborations	• `ProximityService` • `Channel`

Table 7.1: CRC card for the `ProximityManager` class.

Class Name	`ProximityService`
Responsibilities	• Applications communicate with this class via `Channel` connections. • Respond to proximity-based operation requests such as neighbour and service discovery requests.
Collaborations	• `Channel` • `RoutingHelper` • `MetadataService` • `NativeHelper`

Table 7.2: CRC card for the `ProximityService` class.

Class Name	`Channel`
Responsibilities	• Maintain a single connection to the `ProximityService`. This design choice defers the maintenance of the service connection to the user, thereby simplifying thread safety within the framework. • Handle communication with the `ProximityService`. • Notify client applications of service responses by invoking callbacks.
Collaborations	• `ProximityManager` • `ProximityService`

Table 7.3: CRC card for the `Channel` class.

Class Name	`MetadataService`
Responsibilities	• Maintain up-to-date metadata about this device to be served by `MetadataServer`.
Collaborations	• `ProximityService` • `MetadataServer` • `LocationHelper`

Table 7.4: CRC card for the `MetadataService` class.

Class Name	`MetadataServer`
Responsibilities	• Respond to external HTTP metadata requests from other devices.
Collaborations	• `MetadataService`

Table 7.5: CRC card for the `MetadataServer` class.

Class Name	`NativeHelper`
Responsibilities	• Perform native operations on the underlying system, such as executing binaries and modifying file permissions.
Collaborations	• `ProximityService`

Table 7.6: CRC card for the `NativeHelper` class.

Class Name	`LocationHelper`
Responsibilities	• Maintain information about the current GPS location of this device. • Return current GPS location on request.
Collaborations	• `MetadataService`

TABLE 7.7: CRC card for the `LocationHelper` class.

Class Name	`RoutingHelper`
Responsibilities	• Start, stop and interact with the native routing protocol.
Collaborations	• `ProximityService` • `RoutingConfiguration`

TABLE 7.8: CRC card for the `RoutingHelper` class.

Class Name	`RoutingConfiguration`
Responsibilities	• Manage the configuration file for the native routing protocol.
Collaborations	• `RoutingHelper`

TABLE 7.9: CRC card for the `RoutingConfiguration` class.

7.2 Class relationships

We have previously identified the classes that will make up our system, their responsibilities, and the classes they should collaborate with. In this section, we define more rigidly the static relationships between collaborating classes.

The relationships between the system classes are illustrated in standard UML class notation in Figure 7.1. This is a compact form of a class diagram, containing a subset of classes with no attributes or methods, designed for clarity. A full class diagram, complete with attributes, methods and additional classes is given in Appendix A (Figure A.7). It can be seen that the `ProximityManager` is a singleton (designated by the `<<singleton>>` stereotype), and manages zero or more `Channel` objects. These channels "bind" to the background `ProximityService` object. This is the Android "service

binding" as described in Section 3.5.

It can also be seen that the ProximityService is composed of several helper classes, and also the `MetadataService` which is used to communicate with and retrieve metadata about remote devices.

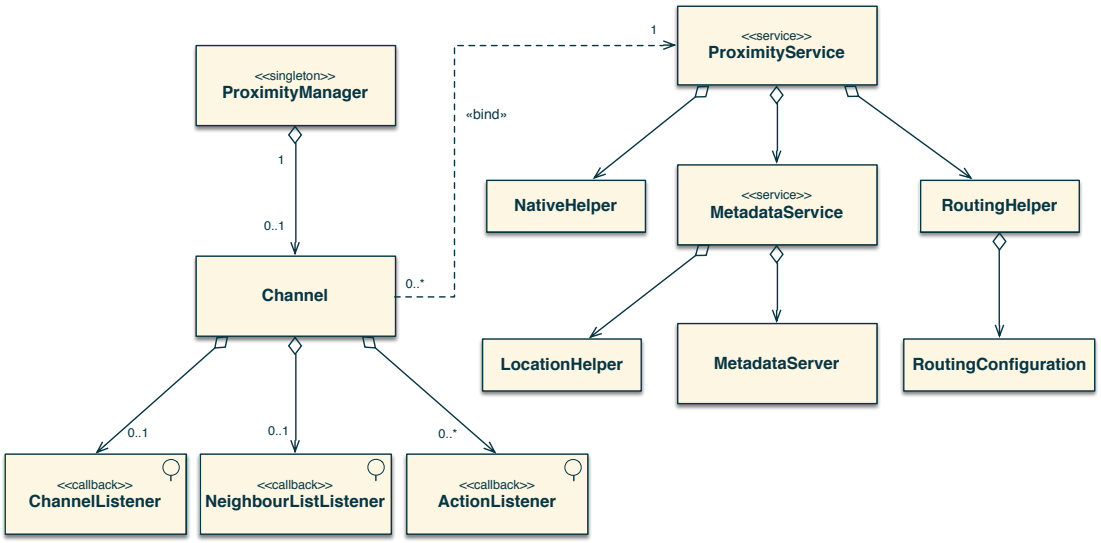


FIGURE 7.1: UML class diagram.

7.3 Package relationships

To get a higher-level, more abstract representation of the system it is important to separate distinct components into *packages* and define the relationships between them. See Figure 7.2 for an illustration of these relationships in standard UML package diagram notation.

The system is composed of three internal packages and two external packages. Each internal package resides within an outer package, named `org.proxima`, following standard Java package notation format. The packages are as follows:

- `org.proxima.client`: Contains client API classes. These are the classes that the client will interact with directly. These include the `ProximityManager`, `Channel`, and related client callback classes.
- `org.proxima.service`: Contains core framework service classes. These include the `ProximityService`, `MetadataService` and `NativeHelper` classes.

- `org.proxima.routing`: Contains classes related to and interacting with the underlying routing protocol. These include the `RoutingHelper` and `RoutingConfiguration` classes.
- `fi.iki.elonen`: This package contains the external NanoHTTPD library (Hawke, 2012), which is used by the `MetadataServer` to handle HTTP requests.
- `org.codehaus.jackson`: This package contains the external Jackson JSON processing library (Codehaus, 2014), which is used by the `ProximityService` and `MetadataServer` to serialize/deserialize JSON messages passed between nodes over HTTP during metadata requests.

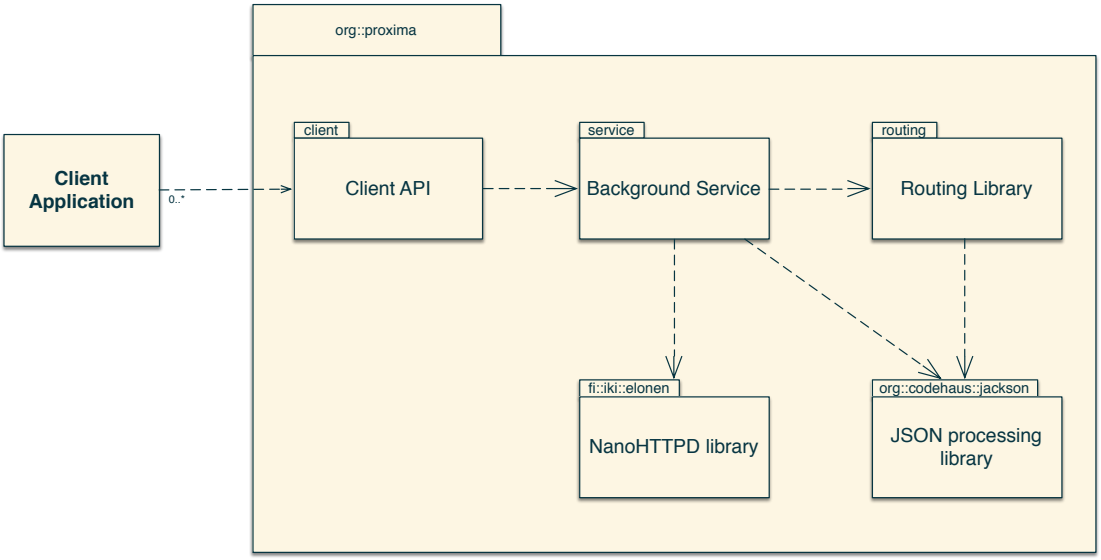


FIGURE 7.2: UML package diagram.

7.4 System interaction

To get some kind of scope of how the system will actually work, it is useful to first model the interactions between internal components in various scenarios. Following this, the external interaction between devices using the system must be defined.

7.4.1 Internal interaction

We can model internal interaction scenarios using standard UML interaction diagrams. When deciding which interactions to model, the *use cases* identified in Chapter 6 will

provide a starting point. Here, we will model only a single use case for brevity. The full set of use case interaction diagrams is given in Appendix A.

Figure 7.3 shows the system interaction corresponding to the `RegisterApplication` use case. Each vertical line represents the lifetime of the object instance that it is connected to. The leftmost object represents a client application that is using the system.

Notice the half-head arrows; these signify *asynchronous* method calls. The client application calls the `initialize()` method of the `ProximityManager` asynchronously. The `ProximityManager` then requests a `Channel` to connect and bind to the `ProximityService`. Upon successful connection, the client application is registered with the framework and the `onChannelConnected()` method of the client's `ChannelListener` is called.

This asynchronous method call with client-supplied callback listener is the core pattern of interaction with the framework, and can be seen with all of the remaining interaction diagrams in Appendix A.

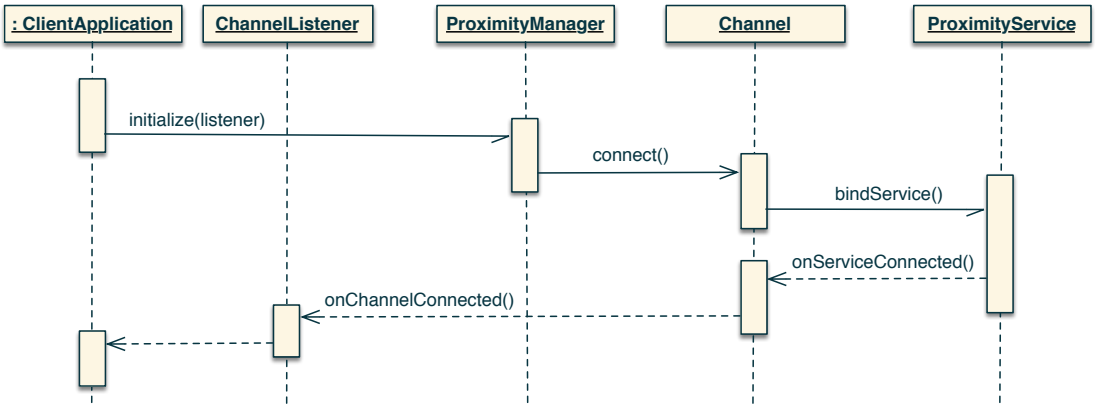


FIGURE 7.3: UML sequence diagram illustrating the `RegisterApplication` use case.

7.4.2 External interaction

Since this is essentially a distributed platform composed of multiple interacting devices, it will be useful to model the interactions between nodes to gain an idea of the external behaviour of the system.

Figure 7.4 shoes this internal interaction. Three devices are shown, within the boundary of the Proxima MANET (the dashed outline). On each device, the Proxima framework can be seen on top of the `olsrd` daemon (as described in Chapter 2) and the native Android system (as described in Chapter 3). Communication between devices is

via the OLSR protocol between the `olsrd` daemons and via HTTP between the Proxima layers themselves. Two Proxima-based applications are shown on each device.

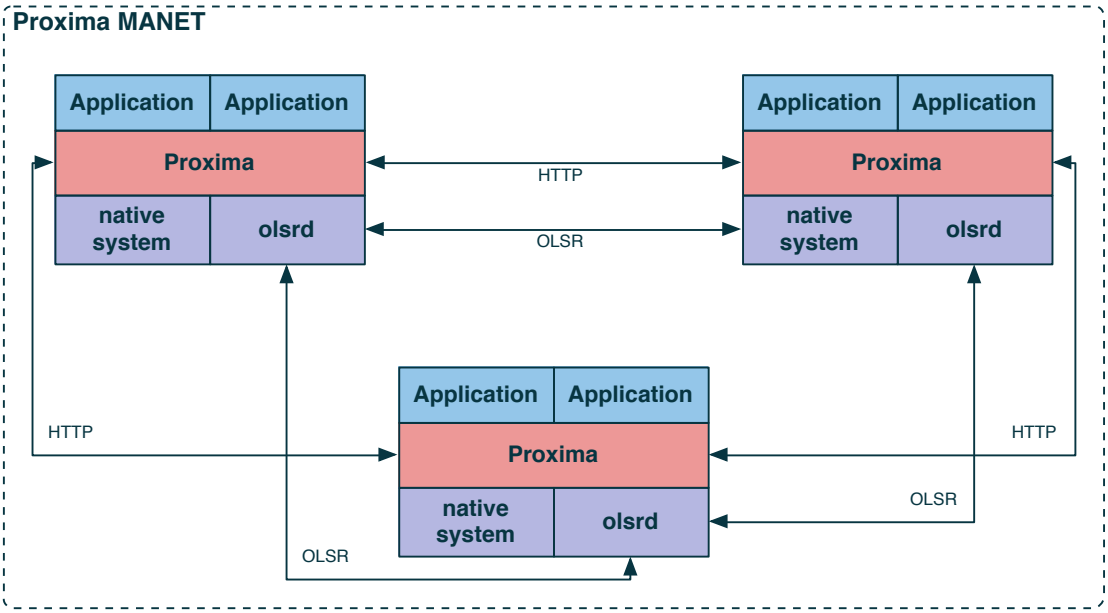


FIGURE 7.4: External interaction between nodes over OLSR and HTTP.

Chapter 8

Testing

In this chapter, we describe the process of testing the project software. Testing is crucial to ensure that the completed software behaves as it should, and also serves as a means of proving the validity of the hypotheses and claims that have been made about the system.

8.1 Tools and resources

The Android SDK provides a comprehensive testing framework and powerful tools that help test every aspect of applications at every level (Google, Inc., 2014). These tools include instrumentation and base classes for testing Activities, Services, Content Providers, and more. Android test cases are based on the popular JUnit 3 (JUnit, 2004).

The Android testing framework also makes it possible to create *mock* instances of classes, which are fake objects that can be injected into test cases in order to isolate them from the rest of the system.

As mentioned in Chapter 5, we have 3 devices available for development. Testing will be done on all three devices.

8.2 Test plan

Android test cases are arranged into projects. The core `Proxima` project will have a separate sister-project for tests, called `ProximaTest`. This is necessary since the Proxima framework exists as a library project and hence cannot be compiled directly into an APK file (see Chapter 3 for an explanation of Android library projects). Testing will be done on three levels:

Unit testing. Otherwise known as white-box testing, unit tests are the lowest-level, smallest type of test. Unit tests cover individual methods independently, in isolation from the rest of the system. The `ProximaTest` project will contain unit tests for each class method, where appropriate. Detailed specifications of these unit tests are not given in this report for brevity.

Functional testing. Otherwise known as black-box testing or integration testing, functional tests are designed to test multiple system components as a whole. Each of the previously identified use cases (from Chapter 6), which correspond to the public API methods of the `ProximityManager` class, will be used to create a single functional test case within the `ProximaTest` project. Detailed specifications of these functional tests are given in Section 8.3.

High-order system testing. System tests are the highest level type of test. The example application *TuneSpy* (see Chapter 10) will be used for high-order multi-device system testing with real data.

8.3 Test specification

This section documents the black-box functional tests that map directly on to the functional requirements, and hence the public API methods, of the system.

The tests will be run on each development device independently. The devices will not interact directly at this stage, but use mock data to simulate responses from other devices.

Since all of the public API methods are asynchronous, we will need a mechanism that allows us to test in an asynchronous manner. This can be achieved using a special class, `CountDownLatch`, which acts somewhat like a condition variable and allows one thread of execution to wait for another to finish processing. Listing 8.3.1 gives an example of using `CountDownLatch` to test the registration of a client application via the `ProximityManager.initialize()` method.

```
public void testRegisterApplication()
{
  ProximityManager proximityManager = ProximityManager.getInstance();
  final CountDownLatch lock = new CountDownLatch(1);
  int marker = 0;

  Channel c = proximityManager.initialize(getActivity(), new ChannelListener()
  {
      @Override
      public void onChannelConnected()
      {
          Log.d(getName, "Channel connected");
          marker = 1;

          // Release the lock
          lock.countDown();
      }

      @Override
      public void onChannelDisconnected()
      {
          Log.d(getName(), "Channel disconnected");
          lock.countDown();
      }
  });

  assertNotNull(c);

  // Block and wait for lock.countDown() to be called
  lock.await();
  assertEquals(1, marker);
}
```

Listing 8.3.1: Running an asynchronous test using a `CountDownLatch`.

Table 8.1 through Table 8.7 give detailed descriptions of the black-box test associated with each public API method.

Test Case Identifier	`RegisterApplication`
Description	Register an application with the Proxima framework.
Input data	A `ChannelListener` implementation for the framework to use to notify of channel connection/disconnection.
Preconditions	The device must have root access available and ad-hoc mode enabled.
Flow of events	1. The `initialize()` method is called and returns a `Channel` instance; 2. The test case waits for the `onChannelConnected()` method of the `ChannelListener` to be called asynchronously.
Pass/fail criteria	The `onChannelConnected()` method is called within 2 seconds.

Table 8.1: Test case `RegisterApplication`.

Test Case Identifier	`ActivatePBF`
Description	Activate proximity-based functionality (PBF).
Input data	An `ActionListener` implementation for the framework to use to notify of activation success/failure.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `discoverNeighbours()` method is called; 2. The test case waits for the `onSuccess()` method of the `ActionListener` to be called asynchronously.
Pass/fail criteria	The `onSuccess()` method is called within 10 seconds.

Table 8.2: Test case `ActivatePBF`.

Test Case Identifier	`RetrieveNeighbourDetails`
Description	Retrieve the list of current neighbours.
Input data	A `NeighbourListListener` implementation for the framework to use to notify when the neighbour list is available.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `requestNeighbours()` method is called; 2. The test case waits for the `onNeighboursAvailable()` method of the `NeighbourListListener` to be called asynchronously.
Pass/fail criteria	The `onNeighboursAvailable()` method is called within 5 seconds.

Table 8.3: Test case `RetrieveNeighbourDetails`.

Test Case Identifier	`RetrieveNeighbourServices`
Description	Retrieve the list of current services available on the network.
Input data	A `ServiceResponseListener` implementation for the framework to use to notify when the service list is available.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `requestServices()` method is called; 2. The test case waits for the `onServicesAvailable()` method of the `ServiceResponseListener` to be called asynchronously.
Pass/fail criteria	The `onServicesAvailable()` method is called within 5 seconds.

TABLE 8.4: Test case `RetrieveNeighbourServices`.

Test Case Identifier	`RegisterService`
Description	Register a local service on the network.
Input data	An `ActionListener` implementation for the framework to use to notify on registration success/failure.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `registerLocalService()` method is called with an appropriate `ServiceInfo` instance; 2. The test case waits for the `onSuccess()` method of the `ActionListener` to be called asynchronously.
Pass/fail criteria	The `onSuccess()` method is called within 1 second.

TABLE 8.5: Test case `RegisterService`.

Test Case Identifier	`UnregisterService`
Description	Unregister a local service from the network.
Input data	An `ActionListener` implementation for the framework to use to notify on unregistration success/failure.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `unregisterLocalService()` method is called with the corresponding `ServiceInfo` instance that was previously registered; 2. The test case waits for the `onSuccess()` method of the `ActionListener` to be called asynchronously.
Pass/fail criteria	The `onSuccess()` method is called within 1 second.

TABLE 8.6: Test case `UnregisterService`.

Test Case Identifier	`SetDeviceName`
Description	Change the name of the device.
Input data	An `ActionListener` implementation for the framework to use to notify on name change success/failure.
Preconditions	A pre-bound `Channel` instance must be available.
Flow of events	1. The `setDeviceName()` method is called; 2. The test case waits for `EditDeviceNameActivity` to show a dialog prompting the user to set the device name; 3. The dialog is filled out and the "Ok" button is "pressed" programmatically; 4. The test case waits for the `onSuccess()` method of the `ActionListener` to be called asynchronously.
Pass/fail criteria	The dialog is displayed, and the `onSuccess()` method is called within 1 second of the dialog being submitted.

TABLE 8.7: Test case `SetDeviceName`.

Chapter 9

Implementation

This chapter details the process of implementation of the Proxima framework. As mentioned in Chapter 5, the implementation has been planned into three iterations of the agile methodology. Each iteration is broken down into several stages. The next few sections describe these iterations and the work that was completed within them, stage-by-stage.

9.1 Iteration 1

By this point, the underlying communication technology has been selected (802.11 ad-hoc mode), the network topology has been chosen (MANET), and the routing protocol has been chosen (OLSR). The first iteration focuses on getting these elements to work on Android.

9.1.1 Stage 1 – Gain root access

To be able to execute compiled C code and to gain access to the networking hardware on Android, it is necessary to gain root access to the device. The method for accomplishing this varies by device and manufacturer. Generally, all that is necessary is to replace the `su` program with a modified version that does not have the default behaviour of denying regular users root access. This is achieved by exploiting one or more security holes in the OS, which allow the system bootloader to be unlocked. Custom firmware can then be uploaded to the device, which usually includes various utilities, the `su` binary itself, and an application called *SuperUser* that is used to manage root access requests.

Many device rooters also choose to install a completely new version of Android at this point. There is a large community of developers who create these custom OS images, known as *ROMs*.

Manufacturers have tried to prevent rooting from being possible, but there is usually still some possible exploit vector. Google-branded phones, such as the Nexus series, make it intentionally easy to unlock the bootloader. It is as simple as running the `fastboot` program with a simple command while connected to a computer in bootloader mode. In most cases, rooting a device will void the manufacturer warranty. Table 9.1 shows the ROMs and rooting methods used for our devices.

Device	**Root method**	**ROM**
Samsung Galaxy Tab	heimdall	Stock
Samsung Galaxy S4	heimdall	CyanogenMod 10.1
HTC Widfire	HBOOT/S-Off	Cyanogenmod 9

Table 9.1: Rooting methods.

9.1.2 Stage 2 – Obtain ad-hoc mode

As mentioned previously, ad-hoc mode is not enabled by default in most stock Android devices. Generally, the stock kernels are shipped with device drivers for the wireless chipset that do not support ad-hoc mode. Modified versions of these drivers must be installed on the device that have ad-hoc support compiled-in.

Device	**Chipset**	**Driver**
Samsung Galaxy Tab 10.1	bcm4330	bcmdhd_samsung
Samsung Galaxy S4	bcm4335	bcmdhd
HTC Widfire	bcm4329	bcm4329

Table 9.2: Wireless chipsets and drivers.

Most Android devices use chipsets and drivers made by Broadcom (bcm*). The modifications that must be made to the drivers were previously described in Section 3.9.

Once enabled, a series of commands using the `ifconfig` and `iwconfig` programs are made which request the wireless driver to switch to ad-hoc mode and set various network parameters such as IP address, netmask and channel frequency. Stock android

kernels do not usually come with these programs, so precompiled versions for Android were obtained.

For a device with interface `eth0` and IP address 192.168.1.2, the necessary commands are shown in Listing 9.1.1. The commands were executed in a shell on the device itself. The shell is obtained by running the command `adb shell` on the host machine while the device is plugged in via USB. The commands were executed from the directory `/data/local/tmp/org.proxima/bin`, as this is the system directory where the precompiled binaries were placed on the device. Most of the commands produce no output, which indicates success.

```
shell@android:/\$ su
root@android:/ # ./wifi load
WIFI: driver loaded

root@android:/ # ./ifconfig eth0 192.168.1.2 netmask 255.255.255.0
root@android:/ # ./iwconfig eth0 mode ad-hoc
root@android:/ # ./iwconfig eth0 essid ProximaMesh
root@android:/ # ./iwconfig eth0 channel 1
root@android:/ # ./ifconfig eth0 up
root@android:/ # ./iwconfig eth0 commit
root@android:/ # echo 1 > /proc/sys/net/ipv4/ip_forward

root@android:/ # ./ifconfig eth0
eth0: ip 192.168.1.2 mask 255.255.255.0 flags [up broadcast running multicast]

root@android:/ # ./iwconfig eth0
eth0      IEEE 802.11-DS  ESSID:"ProximaMesh"  Nickname:""
          Mode:Ad-Hoc  Frequency:2.412 GHz  Cell: EA:C4:43:B7:C4:80
          Bit Rate=65 Mb/s   Tx-Power:32 dBm
          Retry min limit:7   RTS thr:off   Fragment thr:off
          Power Managementmode:All packets received
          Link Quality=5/5  Signal level=-57 dBm  Noise level=-98 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

Listing 9.1.1: Switching the network interface to ad-hoc mode.

At this point, the ad-hoc network was tested with two devices, using the `ping` and `tcpdump` programs to verify and monitor correct network operation.

There was a problem with the S4. IBSS coalescing. Due to ambiguity in 802.11b spec. Fix: copy `bcmdhd_ibss.bin` from stock ROM and copy to `/system/etc/wifi`. Then restart the wifi module with the IBSS firmware, as shown in Listing 9.1.2.

After this, the S4 will cooperate and coalesce IBSSIDs with the other devices.

```
root@android:/ # rmmod dhd.ko
root@android:/ # insmod /system/lib/modules/dhd.ko \
                  firmware_path=/system/etc/wifi/bcmdhd_ibss.bin \
                  nvram_path=/system/etc/wifi/nvram_net.txt
root@android:/ #
```

Listing 9.1.2: Injecting the IBSS firmware fix into the Wi-Fi driver.

9.1.3 Stage 3 – Compiling and running `olsrd`

First, it is necessary to obtain the Android Native Development Kit (NDK), as it provides the necessary compilation tools to compile code for the Android platform. Next, obtain source code from `http://olsr.org`. Extract the source archive and run the following command to compile the executable and plugins:

```
make OS=android NDK_BASE=/opt/android-ndk build_all
```

where `NDK_BASE` is the root directory of the Android NDK. Initially, olsrd version 0.6.5.5 was used. However, there was a problem with the jsoninfo plugin in this version; the plugin would generate invalid JSON output. This bug was fixed in the master branch of the source repository, but had not yet been packaged into a release. So the repository was cloned and everything was rebuilt.

Before running `olsrd`, it is necessary to set the `LD_LIBRARY_PATH` environment variable so that the plugins can be found and dynamically loaded by the system. The jsoninfo plugin is placed in the`/data/local/tmp/org.proxima/lib` directory. Listing 9.1.3 shows the necessary commands.

```
root@android:/ # LD_LIBRARY_PATH=/data/local/org.proxima/lib:$LD_LIBRARY_PATH$
root@android:/ # ./olsrd -f  ../conf/olsrd.conf -i eth0 -d 2

 *** olsr.org - pre-0.6.7-git_5bfc121-hash_a7a967ef1abe0769624fdef7f227c8c9 ***
 Build date: 2013-11-04 18:37:30 on vortex
 http://www.olsr.org

 <output clipped for brevity>
```

Listing 9.1.3: Running `olsrd`.

Here, you can see the custom `olsrd` build version and date. `vortex` is the author's development machine. Now we have `olsrd` running with the `jsoninfo` plugin, we can use it to retrieve routing information. The `jsoninfo` plugin listens on port 9090. Listing 9.1.4 shows the output of the plugin via the `nc` tool.

```
root@android:/ # nc localhost:9090
{
    "links": [{
        "localIP": "192.168.1.2",
        "remoteIP": "192.168.1.7",
        "validityTime": 37941,
        "linkQuality": 1.000,
        "neighborLinkQuality": 1.000,
        "linkCost": 1024
    }]
    ,
    "neighbors": [{
        "ipv4Address": "192.168.1.7",
        "symmetric": true,
        "multiPointRelay": false,
        "multiPointRelaySelector": false,
        "willingness": 3,
        "twoHopNeighbors": [],
        "twoHopNeighborCount": 0
    }]
}
 <output clipped for brevity>
```

Listing 9.1.4: Obtaining routing info from `jsoninfo` plugin via `nc`.

9.1.4 Stage 4 – Native system interaction from Java code

Now that we have our ad-hoc network with OLSR routing up and running manually, it is time to create an interface between Java applications and the native system commands, in order to be able to automate the previous stages from Java code.

Previously we have executed system commands directly from a shell on the device. In Java, it is possible to execute shell commands using the built-in `Runtime` class. A brief sample of this method is shown in Listing 9.1.5. The `Runtime` class also provides methods to get the standard output and error streams from the process. The `NativeHelper` class encapsulates this native system interaction within Proxima.

```
public static int runShellCommand(String command)
{
    try
    {
        Process process = Runtime.getRuntime().exec(command);
        return process.waitFor();
    }
    catch (Exception e)
    {
        Log.e(TAG, "Error running command " + command, e);
        return -1;
    }
}
```

Listing 9.1.5: Running a shell command from Java.

The olsrd implementation features a plugin system. The jsoninfo plugin (Tonnesen *et al.*, 2008) is used to interact with the olsrd daemon. It listens on the localhost interface on port 9090 and takes a URL of the following form:

```
http://127.0.0.1:9090/<command>
```

where `<command>` is the command to be given to the plugin. Many commands are available. For example, to obtain a list of all reachable devices in the network, the `links` command is used. The Jackson JSON library (Codehaus, 2014) is used to parse the output from this command and create Java objects representing the neighbouring devices.

A configuration file packaged with the framework specifies the plugins to be used and wireless interfaces to be used.

9.2 Iteration 2

Now that we are able to access ad-hoc mode and run OLSR from Java code, it is time to start shaping the framework itself. The second iteration focuses on building a solid back-end service and a well-defined client facing API.

9.2.1 Stage 5 – Create Android library project

The framework is intended to be deployed as a single, stand-alone artifact that application developers can simply download and use with minimal configuration. This is achieved by delivering the framework as an Android *library project*. Library projects have the same structure as regular Android applications, except for the fact that they cannot be compiled into APK files and hence cannot be installed as applications in their own right.

To include the Proxima framework as a dependency to a standard Android project, all that is required is to add the following two lines to the `project.properties` file of the project:

```
android.library.reference.1="</path/to/Proxima>"
manifestmerger.enabled=true
```

where `/path/to/Proxima` is the path to the root directory of the Proxima library project. The second line instructs the build system to merge the Android manifest files

of the library project and the client application project. This prevents the client from needing to duplicate the manifest information, such as access permissions.

9.2.2 Stage 6 – Implement background services

In Chapter 3, the concept of Android background services was discussed. Two services are used in the implementation; the `ProximityService` and the `MetadataService`.

- `ProximityService` – Runs as a *bound* service in a separate process. Client applications bind to the service via calls to `ProximityManager.initialize()`, which returns a `Channel` object. This object holds the service binding and is passed to the `ProximityManager` for all other API calls.

- `MetadataService` – Runs as a *started* service in a separate process. Local client applications do not interact with this service directly, but the `ProximityService` sends updates to the service when changes occur in the network, such as new neighbours arriving in proximity. The `MetadataService` runs an embedded HTTP server using the NanoHTTPD library. It is this server to which neighbours connect to retrieve metadata about a particular device, such as its name, GPS location, and available services.

The metadata service

The metadata service functions as an HTTP server and is responsible for serving information about its parent node, such as GPS coordinates and human-readable node identifiers. The metadata service also acts as a service discovery layer. Client applications are able to register their services, and can even specify custom per-application metadata to be served.

See Figure 9.1 for a high-level overview of the architecture of the Proxima framework, shown as a UML deployment diagram. Figure 9.1 shows the Proxima framework being used by multiple clients on multiple devices. It shows the communication between two metadata services, and the underlying OLSR daemon communication mechanism.

9.2.3 Stage 7 – Define client API

The `ProximityManager` is the class that clients interact with and hence forms the client API. The methods exposed by the `ProximityManager` map directly on to the use cases

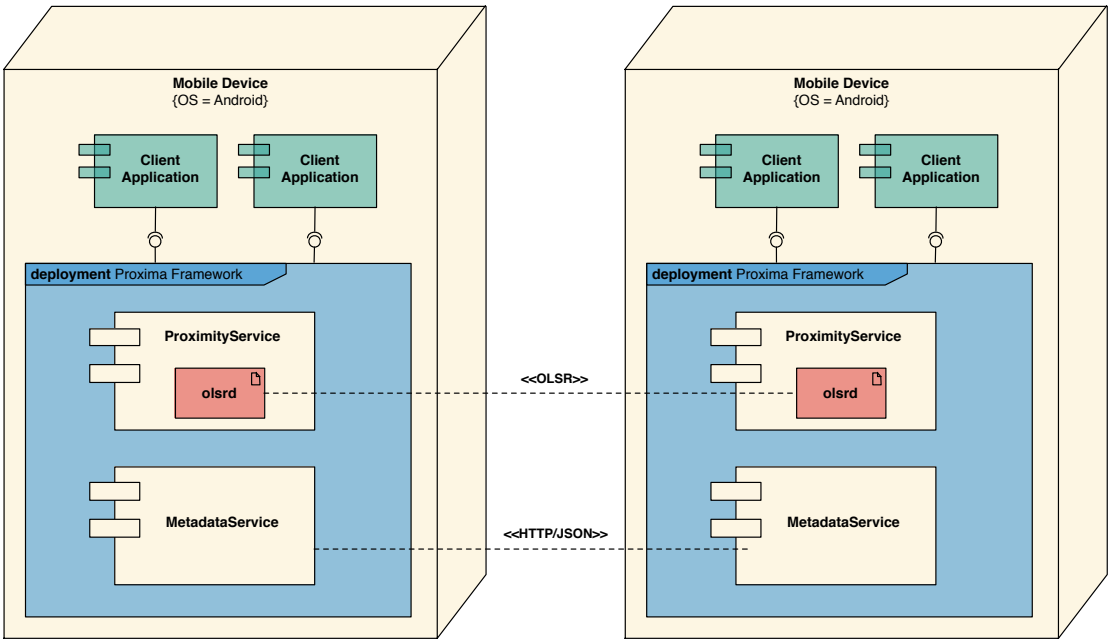


FIGURE 9.1: UML deployment diagram illustrating the Proxima architecture.

defined in Chapter 6. The `ProximityManager` should be a singleton, and all methods should be asynchronous, due to the fact they may take a long time. This is done by providing callback interfaces. Clients are responsible for keeping a Channel object, this is to ease thread safety within the framework by removing the need to manage all the service connections and listeners.

9.2.4 Stage 8 – Implement IP autoconfiguration

IP autoconfiguration is currently implemented in a very rudimentary way. The subnet ID is simply randomly generated (i.e. 192.168.1.x where x is randomly generated from 1-254). This works fine for small networks, but a collision resolution strategy will be needed for larger networks.

9.3 Iteration 3

By this point, we have a solid framework in place. The third iteration focuses on enriching the functionality of the framework. Excess time was available to implement name resolution, service discovery and geolocation distribution.

9.3.1 Stage 9 – Implement name resolution

DNS-like name resolution was initially handled using a combination of the olsrd nameservice plugin and the dnsmasq program. The olsrd plugin modified the /etc/hosts and /etc/resolv.conf files and sent a SIGHUP signal to the dnsmasq process when there was a change in local devices. Each node was then effectively a DNS server; it would resolve its own hostname when asked, and kept a cache of hostnames for other nodes. Other such solutions exist, such as described in (Hong *et al.*, 2007, 2005), but the functionality was already in the OLSR implementation and is open-source and extendable.

However, this solution was found to be too brittle. For example, if a node changed IP address, then it would take a while for each node to update its DNS cache, causing inconsistencies. Additionally, this solution only allows names composed of the small subset of characters allowed by the DNS protocol, which does not include Unicode characters or even whitespace characters.

Following these discoveries, name resolution has been absorbed into the metadata service. Upon calling the `setDeviceName()` method of the `ProximityManager`, a simple dialog is displayed in which the user can edit the device name. This device name is then passed to the metadata service and will be discoverable by neighbour nodes.

9.3.2 Stage 10 – Implement geolocation distribution

The metadata service uses the Android location API to register and receive updates periodically about GPS location changes.This geolocation information is then distributed to neighbours on-demand via the metadata service.

9.3.3 Stage 11 – Implement service discovery

Initially, implementation of service discovery was deferred (assigned "WONT" MoSCoW priority) but time was found towards the end of the project to add this important feature.

UPnP-like service discovery could easily be implemented using an extension to the olsrd nameservice plugin, and having additional framework API methods for registering/searching/unregistering services. However, to reduce dependency on OLSR, it was decided to incorporate service discovery into the metadata service.

Clients can create a `ServiceInfo` object describing their service and pass it into the API. It will then be registered by the metadata service and will be available for discovery.

9.4 Proxima – A developer viewpoint

In this section we describe the Proxima framework from an application developer's viewpoint, in terms of how simple it is to integrate proximity-based functionality into an Android application. We then describe some of the low-level detail and design decisions that were made during development.

The Proxima framework provides a fully asynchronous, thread-safe API and can be used by multiple client applications on a single device. It acts like a "neighbours and resources finder" server. It is not necessary for API users to worry about how to discover neighbours and what services are available; the framework will take care of these low-level details. Communication with the framework is done using asynchronous method calls with user-supplied callback functions. Users are sent broadcast intents when changes occur, such as a new device coming into proximity.

Registering a client application and retrieving a list of neighbours is demonstrated in Listing 9.4.1. The client-facing API takes inspiration from the Android Wi-Fi P2P API. This is a simplified version; the calls to `initialize()` and `discoverNeighbours()` both take nullable callback arguments to notify the user of success or failure. These calls are usually placed into the `onCreate()` method of an Android activity.

```
1  // Obtain a reference to the ProximityManager.
2  ProximityManager mProximityManager = ProximityManager.getInstance();
3
4  // Initialize a new channel to the framework. This must be the first
5  // call into the API.
6  Channel mChannel = mProximityManager.initialize(this);
7
8  // Begin the neighbour discovery procedure (activate proximity-based
9  // functionality).
10 mProximityManager.discoverNeighbours(mChannel);
11
12 // Retrieve the list of neighbours.
13 mProximityManager.retrieveNeighbours(mChannel, new NeighbourListListener()
14 {
15         @Override
16         public void onNeighboursAvailable(List<ProximityDevice> neighbours)
17         {
18              // Process the list of neighbours.
19         }
20 });
```

Listing 9.4.1: Example usage of the Proxima API.

The framework also sends periodic broadcasts to all registered client applications when there is a neighbour change, so clients should implement a broadcast receiver to

listen for these specific broadcasts. Neighbours can be retrieved at any point once the discovery process has begun.

Each neighbour in the retrieved list represents a device detected by the routing daemon. Prior to passing this list to the user via the callback, the framework queries the metadata service on each neighbour to retrieve information such as the user-specified device name, the device GPS coordinates and the available services (Proxima applications) provided by the device.

In summary, as a mobile application developer, all that is necessary to use the Proxima framework is to:

- Register the application with the framework;
- Request the framework to begin the neighbour discovery process;
- Retrieve the currently available list of neighbours, on-demand.

The framework will also send notifications of neighbourhood changes to you via broadcast intents should you wish to register a broadcast receiver. It is very simple and easy to use. For a detailed description of how to use the framework, plus how to use the advances features, see Appendix B.

9.5 Challenges faced with implementation

There were many challenges faced with implementation. The most difficult aspect was enabling ad-hoc Wi-Fi mode in the kernel. Each device has its own idiosyncrasies and issues, and there is little to no documentation available on the wireless drivers themselves. The IBSS coalescing issue on the Galaxy S4 (see Section 9.1.2) was completely unexpected, but was eventually solved after 6 months of laborious effort.

Since the framework is fully asynchronous and comprises multiple threads of execution, debugging was also a significant challenge. There is no simple way to do this. Testing proved to be invaluable in this aspect, to ensure that changes did not break the core functionality.

The core framework accounted for approximately 80% of the total implementation time. The TuneSpy application was started at the end of the second iteration, and accounted for the remaining 20% of development time.

Chapter 10

Example application – *TuneSpy*

This chapter presents *TuneSpy*, a significant real-world application built on top of the Proxima framework. TuneSpy allows users in proximity to stream their current music track to one other on a purely local, ephemeral, peer-to-peer basis with absolutely no reliance upon static networking infrastructure. TuneSpy also includes a fully-featured music player and browser. The functionality of TuneSpy was the initial inspiration for this project, and for the scenario described in Section 1.1.1.

Although the main focus of this project is the development of a reusable framework for proximity-based communication, it is important to prove the effectiveness of the framework by implementing a real application on top of it. This chapter gives a condensed overview of the requirements analysis, design, implementation and testing of the TuneSpy application. The application was developed in parallel with the core framework, and they therefore grew and evolved together. However, it should be made clear that the two are completely separate entities.

10.1 Requirements

TuneSpy has the following functional requirements:

Local music playback. The application must allow the user to listen to the music that is stored locally on their device, in a familiar way. This functionality should mirror that of standard mobile audio player applications. It should allow the user to browse and play list of local tracks, have a shuffle mode, have a repeat mode, etc.

Displaying list of nearby devices. The application must maintain and display a list of devices currently in proximity that are currently using the application. This list should also display information about the track that each nearby device is playing, including track name, artist, and album artwork.

Remote music playback. Upon selecting a neighbour from the list, the application should connect and play the track that the nearby user is listening to in real time. This is the "spying" element, although by using the application for local playback the user is consenting to their music being "spied on".

10.2 Design

This section gives brief details of the design process for TuneSpy, focusing primarily on the user interface aspect.

The user interface should consist of three main components, organised into tabs, and implemented as Fragments; the *playback* tab, the *local media* tab and the *nearby devices* tab.

Playback tab. This should display the currently playing track, be it local or remote. The album artwork should be displayed prominently in the centre of the screen, with the track name, artist name, and playback controls at the bottom.

Local media tab. In the first instance, this should display a list of songs on the local device. Clicking on a song should transfer the user to the playback tab and begin playing the selected track.

Nearby devices tab. This is the tab that displays the nearby devices and their current tracks. Upon clicking on an item, the user should be transferred to the playback tab and the remote track should begin playing. The remote artwork, track name, artist name etc. should be displayed in the same manner as local tracks.

Figure 10.1 shows the conceptual user interface design mockups that were created as a starting point for the TuneSpy UI. These mockups were created using the powerful OmniGraffle design suite (`http://www.omnigroup.com/omniGraffle`). Each tab was mocked up before implementing as a Fragment component using the Eclipse Android interface builder.

FIGURE 10.1: TuneSpy UI mockups.

10.3 Implementation

This section gives brief details of the implementation of TuneSpy.

10.3.1 Playback tab

The playback tab includes all the essential functionality for an audio player. It allows track playing/pausing, skipping, seeking, repeating and shuffling. It also allows background audio playback, i.e. the user can navigate away from the application or turn off the screen and playback will continue.

10.3.2 Local media tab

Particular attention was given to the responsiveness of the user interface. Mobile devices are limited in terms of the amount of memory available to a single application, therefore an interface that must display a large number of bitmap images such as the local media tab must take care to manage resources properly. For large user media libraries, it would simply not be possible to load every image bitmap into memory on application startup.

Thus, the local media tab uses a technique called least recently used (LRU) caching. An LRU cache is a fixed-size cache. When an item in the cache is accessed, it is moved to the front of an internal queue. When the cache becomes full, new items are added

to the front of the queue. The item at the end of the queue, i.e. the least recently used item, is discarded. This results in a smooth, responsive scrolling experience for the user.

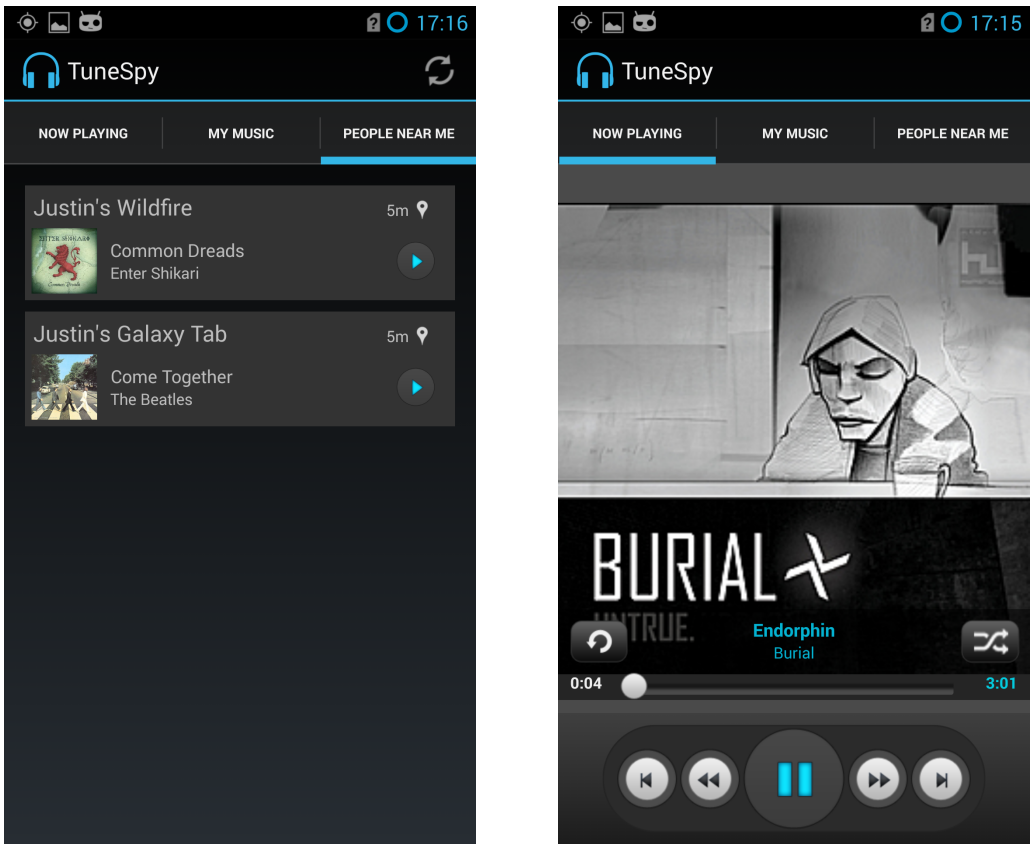
10.3.3 Nearby devices tab

The application uses the Proxima framework to broadcast metadata about itself to surrounding neighbours. This is done via the service discovery feature of the Proxima framework. Custom metadata including the current track name, artist and album artwork imagery is registered with and distributed on-demand by the framework itself.

Upon request from a neighbour, the application will directly stream the actual track data. This is done via an additional embedded HTTP server that provides progressive streaming capabilities.

10.4 Results

Figure 10.2 shows two screenshots of the TuneSpy application. Figure 10.2b shows the nearby devices tab, and Figure 10.2a shows the audio playback tab.



(A) Display of local neighbours (B) Main audio player screen

FIGURE 10.2: Early screenshots of TuneSpy sample application.

Chapter 11

Conclusion

This chapter aims to evaluate and reflect upon the project goals, identifies potential improvements and future research areas, and discusses learning outcomes.

11.1 Reflection on the project goals

As mentioned in Chapter 5, the goals for this project were twofold; the development of a reusable framework for proximity-based functionality, and the implementation of a real-world application based on the framework. These goals are appraised in the following sections.

11.1.1 The Proxima framework

The Proxima framework is a novel and useful abstraction tool for mobile peer-to-peer proximity-based networking, following the initial vision of the proximity-based paradigm. We believe that our work is a significant contribution to the peer-to-peer networking community. This belief is held for the following reasons.

Firstly, a framework such as this, with its unique semantics of purely local information dissemination, allows application developers to create entirely new classes of unique proximity-based applications, or incorporate this functionality into existing applications. Based on our current research, there is no other system available on the market with the flexibility, ease of use, generality and integrated functionality of the Proxima framework.

Secondly, Proxima provides a zero-configuration interface. All important network configuration tasks are carried out automatically by the framework. The only aspect that needs to be configured is the device name (similar in principle to the Bluetooth

device name) which can be achieved programmatically with a simple API call. This ease of configuration makes the framework easy to use for both developers and users.

Thirdly, our separation of underlying connectivity mechanism, routing functionality and higher-level services gives us flexibility to potentially change any one of these elements in the future. Since we are currently using the OLSR protocol as a routing back-end, applications developed using the framework benefit from all the scalability and resilience of the protocol. The framework takes care of the transport mechanics, device specifics and configuration issues, leaving the application developer to focus solely on implementing their application.

Finally, the framework is very lightweight at only 6MB (including all necessary native binaries and compiled code), representing a small overhead for addition into applications as a dependency.

11.1.2 TuneSpy

TuneSpy represents the successful implementation of a real-world application which uses all the functionality of the Proxima framework. The application demonstrates just how easy it is to use the framework, and has served as an invaluable platform for functional testing.

While further refinements would be necessary to bring TuneSpy to a marketable stage, it is nonetheless an innovative, novel product. There is currently nothing like it on the market. Nothing like this actually exists, anywhere.

11.2 Evaluation of original hypothesis

With regard to the original hypothesis made in Chapter 5, the project is a success. It has been possible to create a reusable, hardware-agnostic framework for proximity-based functionality without Internet connectivity.

The success of TuneSpy also supports the hypothesis, at least partially. It will require much more extensive testing to prove the stability and reliability of the framework, as there are many Android devices that were not accessible due to obvious cost considerations.

Overall, the original hypothesis appears to be thoroughly supported by the successful completion of the two project goals.

11.3 Issues and potential improvements

The most significant issue has been the lack of built-in support for ad-hoc mode on Android and hence the necessity for extensive device modifications. The modifications that had to be made to each device to enable ad-hoc mode were far more time-consuming than first anticipated. It required significant time and effort to figure out what the problem actually was. Since Android is a highly modified version of Linux, and there are very few people trying to make these modifications, there were little to no resources available as a starting point for debugging. It took a lot of trial and error to narrow down the problem before a fix could be attempted. It is not likely that native support for ad-hoc mode will be added to Android in the future.

One potential improvement would be to add an authentication and encryption mechanism to the underlying connection. This was deferred in the initial prioritisation stage. Another improvement would be to make the service discovery protocol compliant with a standardised protocol such as UPnP. This would allow interaction with devices that already use the protocol, such as printers and home media servers.

11.4 Future work

We hope in the future to extend the framework, experimenting with different IP auto-configuration strategies and implementing security mechanisms, amongst other things.

We also hope to explore alternative ad-hoc connectivity mechanisms that do not require extensive device modification but still endow us with the proximity-based semantics that we desire.

We will continue to test and support the framework on a wider variety of Android devices and larger networks. We also hope to implement a version of the framework for Apple iOS, and another version for desktop operating systems.

11.5 Learning outcomes

During the development of the core framework, much has been learnt about networking configuration, packet routing, and peer-to-peer systems. With respect to the Android aspect, the author has gained a great deal of knowledge about the low-level operating system details, especially the Android kernel. During the development of TuneSpy,

much has been learnt about responsive UI design on mobile devices, and also Android application development in general.

From a non-technical perspective, several other skills have also been enhanced. Efficient time management has been improved; weekly meetings with the project supervisor proved invaluable for setting incremental deadlines and ensuring that the project was completed with plenty of time to spare, although care had to be taken to not allow the tendency to perfectionism to hinder progress. Working under this kind of pressure was challenging, but was alleviated by breaking the problem down into manageable chunks whenever possible.

The author's ability to overcome unexpected problems has improved as a result of the project. With a large task such as this, unexpected problems are inevitable. A combination of time management, critical thinking and rational analysis allowed these problems to be overcome without negative effect on the project.

11.6 Reflection on the research approach

The initial literature survey yielded far more material than had been first anticipated, and it was challenging to make sense of the sheer volume of information. An adaptation of the survey methods given in (Dawson, 2000) was of great help in breaking down the review into stages. This consisted of defining a search scope, performing the search, evaluating the material, writing a review, and then redefining the search with a narrower breadth of focus, culminating in the completed review.

This approach required a great deal of critical evaluation of material in a relatively short space of time, usually by reading the abstract of an article and making a quick decision about its relevance. The author's skills at critical appraisal have grown considerably because of this. The use of mind maps and spider diagrams was also highly useful in the initial research stage of the project.

11.7 Reflection on the development process

Overall, the iterative development approach suited the project very well, and integrated nicely with the MoSCoW task prioritisation strategy. The clear distinction of each task allowed a subset of tasks to be completed within the timescale of each iteration. If one

negative criticism were to be made, it would be that the development process should have factored testing more heavily from the beginning.

Overall, the author's time prioritisation allowed the full completion of both core project goals, with the resulting products more than completely fulfilling the functional requirements of the project.

11.8 Closing Statement

Proximity-based mesh networking looks set to hit the public arena in a big way. Soon, everyday objects will become networked via the "Internet of Things", and mesh networking will be instrumental in making this a success. Additionally, in these times of heightened electronic surveillance, ad-hoc mesh networking could have a similar impact as the Internet itself once did in how it undermines communication control and authority.

The Proxima framework represents the leading edge in proximity-based computing. There are challenges that need to be overcome in order for this technology to become mainstream, but this proof-of-concept work is a strong step in the right direction.

Appendix A

Extended design documents

A.1 Full UML sequence diagrams

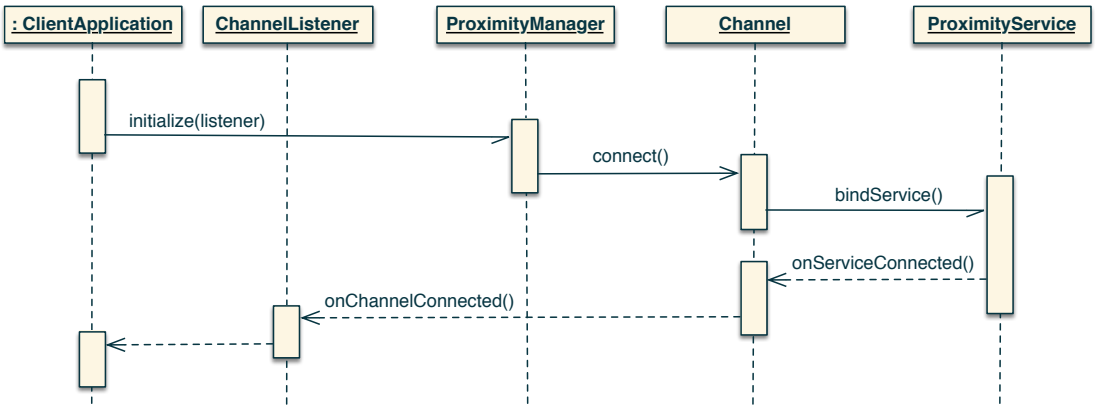


FIGURE A.1: UML sequence diagram for the `RegisterApplication` use case.

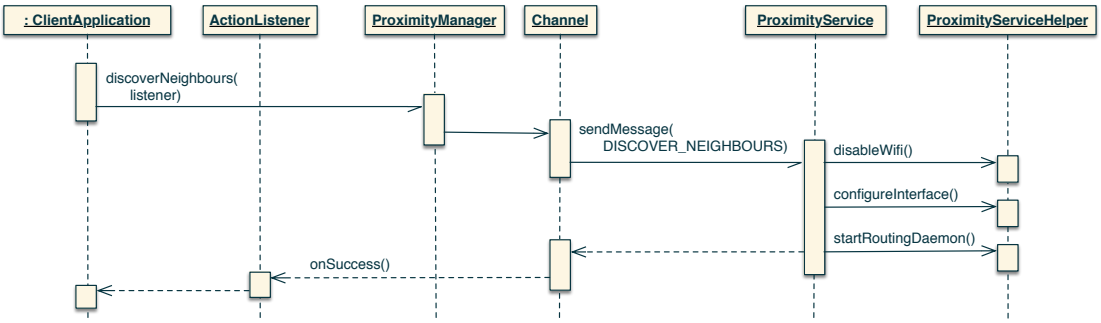


FIGURE A.2: UML sequence diagram for the `ActivatePBF` use case.

A.2 Full UML class diagram

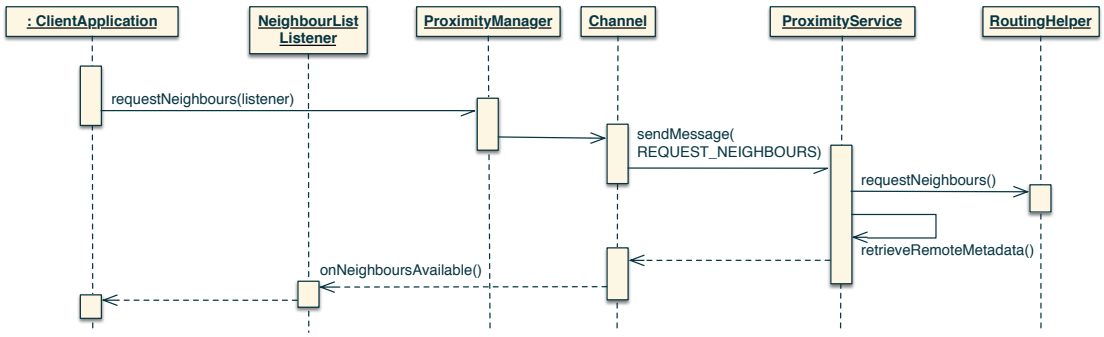


FIGURE A.3: UML sequence diagram for the `RetrieveNeighbourDetails` use case.

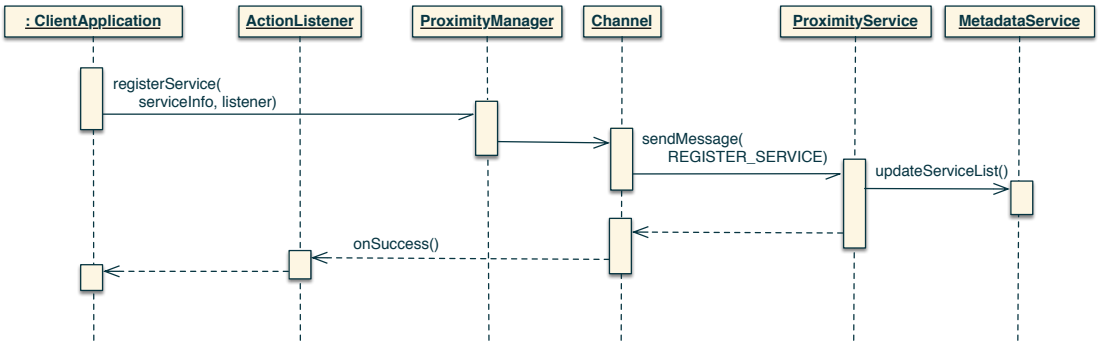


FIGURE A.4: UML sequence diagram for the `RegisterService` use case.

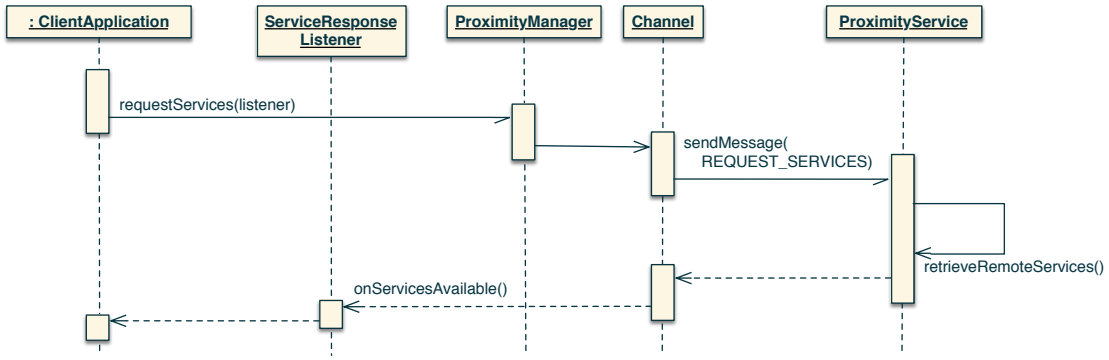


FIGURE A.5: UML sequence diagram for the `RetrieveNeighbourServices` use case.

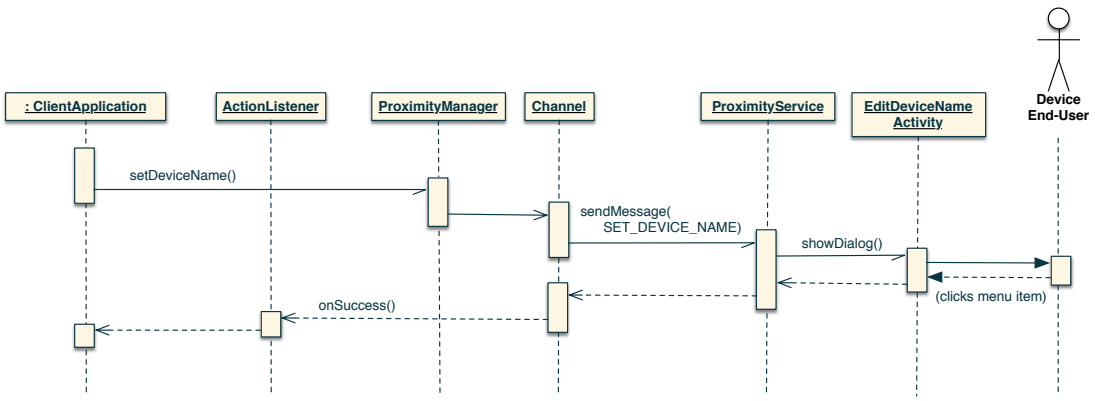


FIGURE A.6: UML sequence diagram for the `SetDeviceName` use case.

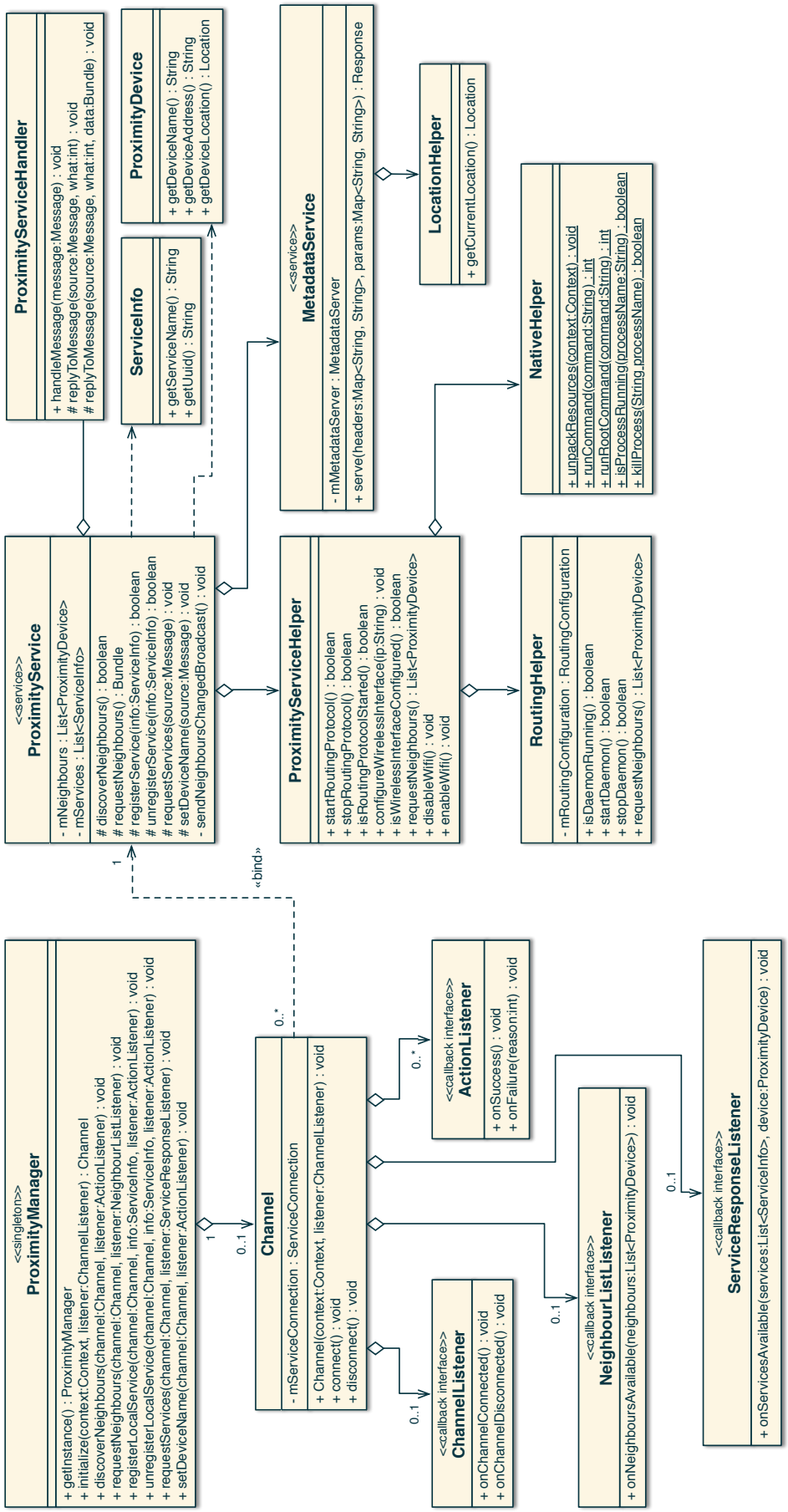


FIGURE A.7: Full UML class diagram.

Appendix B

Proxima documentation – Example usage

This appendix describes in detail how to use the Proxima framework to interface with the Proxima mesh network and discover other nearby devices.

The framework provides zero-configuration networking, i.e. no configuration is necessary to establish network connections with nearby devices. The framework also provides a fully asynchronous and easy-to-use API. Devices using the framework must have root access and must have ad-hoc Wi-Fi enabled in the kernel.

B.1 Include Proxima as a dependency

The Proxima framework is distributed as an Android Library project. This means that you must include it in your project as a dependency before you can use it. This is achieved by modifying the `project.properties` file in the root directory of your application. The following two lines shown in Listing B.1.1 must be added.

```
1 android.library.reference.1="</path/to/Proxima>"
2 manifestmerger.enabled=true
```

Listing B.1.1: Including the Proxima framework as a dependency to an Android application.

where `/path/to/Proxima` is the path to the root directory of the Proxima library project. The second line instructs the build system to merge the AndroidManifest.xml file of the library project with your own. This prevents you from needing to duplicate the manifest information, such as registering access permissions.

B.2 Register with the framework

The `ProximityManager` is the main class that you will be using to interact with the Proxima framework. It allows concurrent thread-safe use by multiple client applications. Listing B.2.1 shows how to obtain a reference to the `ProximityManager` and register with the framework.

```
1  private ProximityManager mProximityManager;
2  private Channel mChannel;
3  ...
4  @Override
5  protected void onCreate(Bundle savedInstanceState)
6  {
7      ...
8      // Obtain a reference to the ProximityManager
9      mProximityManager = ProximityManager.getInstance();
10
11     // This must be the first call into the API
12     mChannel = mProximityManager.initialize(this, new ChannelListener()
13     {
14         @Override
15         public void onChannelConnected()
16         {
17             // Indicates that the Channel has connected successfully
18         }
19
20         @Override
21         public void onChannelDisconnected()
22         {
23             // Called when the channel is disconnected for some reason
24         }
25     });
26     ...
27 }
```

Listing B.2.1: Registering with the ProximityManager.

The call to `initialize()` returns a `Channel` object that represents a connection to the background framework. You must keep hold of this channel, as it will be used for all future API calls. The `onChannelConnected()` method of the supplied `ChannelListener` callback will be invoked asynchronously when the channel is connected. If the channel connection breaks for some reason, the `onChannelDisconnected()` method will be called.

B.3 Register for updates

Once activated, the Proxima framework will periodically send broadcast intents to all client applications that have requested to receive them. These broadcasts will notify you

if proximity-based functionality is enabled or disabled, and also of devices entering and leaving the Proxima mesh. Should you wish to receive these broadcasts (and it is highly recommended that you do) then you must follow the steps outlined in the following subsections.

B.3.1 Create a `BroadcastReceiver`

Broadcast intents are handled by broadcast receivers. To create a broadcast receiver, extend the `BroadcastReceiver` class and override the `onReceive()` method. The `onReceive()` method will be called when a new broadcast intent is detected, and it is here that you must handle the received intents. Listing B.3.1 shows an example of a `BroadcastReceiver` implementation.

```
1  public class ExampleBroadcastReceiver extends BroadcastReceiver
2  {
3      private final ProximityManager mProximityManager;
4      private final Channel mChannel;
5      private final ExampleApp mActivity;
6
7      public ExampleBroadcastReceiver(ProximityManager manager, Channel channel,
8                                      ExampleApp activity)
9      {
10         super();
11         mProximityManager = manager;
12         mChannel = channel;
13         mActivity = activity;
14     }
15
16     @Override
17     public void onReceive(Context context, Intent intent)
18     {
19         String action = intent.getAction();
20
21         if (ProximityManager.PROXIMITY_STATE_CHANGED_ACTION.equals(action))
22         {
23             // Indicates that proximity-based functionality has been enabled
24             // or disabled.
25         }
26         else if (ProximityManager.PROXIMITY_PEERS_CHANGED_ACTION.equals(action))
27         {
28             // Indicates that there has been a change in neighbours in the mesh.
29             // Call ProximityManager.requestNeighbours() to get a list of current
30             // neighbours
31         }
32     }
33 }
```

Listing B.3.1: Creating a `BroadcastReceiver`.

B.3.2 Create an `Intent` filter

To specify which broadcast intents you wish to receive, you must create an `IntentFilter` which filters and matches certain intents. The Proxima framework sends two intent types:

`ProximityManager.PROXIMITY_STATE_CHANGED_ACTION`

This intent is broadcast whenever the state of proximity-based functionality is changed, i.e. enabled or disabled.

`ProximityManager.PROXIMITY_NeighbourS_CHANGED_ACTION`

This intent is broadcast whenever there is a change in network topology, i.e. neighbours leaving or entering the Proxima mesh network.

Listing B.3.2 shows how to create an `IntentFilter` and specify the Proxima intents to be registered.

```
1  private BroadcastReceiver mBroadcastReceiver;
2  private final IntentFilter mIntentFilter = new IntentFilter();
3  ...
4  @Override
5  public void onCreate(Bundle savedInstanceState)
6  {
7      super.onCreate(savedInstanceState);
8      setContentView(R.layout.main);
9
10     mBroadcastReceiver = new ProximityBroadcastReceiver(mProximityManager,
11                                                         mChannel, this);
12     // Indicates a change in the proximity status.
13     mIntentFilter
14             .addAction(ProximityManager.PROXIMITY_STATE_CHANGED_ACTION);
15
16     // Indicates a change in the list of available neighbours.
17     mIntentFilter
18             .addAction(ProximityManager.PROXIMITY_NeighbourS_CHANGED_ACTION);
19
20     ...
21 }
```

Listing B.3.2: Creating an `Intent` filter.

The `BroadcastReceiver` then needs to be registered with the Android system, along with the previously created `IntentFilter`. The best place to do this is in the `onResume()` method of your activity. The receiver should be unregistered in the `onPause()` method. Listing B.3.3 shows how to register and unregister the receiver.

```java
/**
 * Register the broadcast receiver with the intent values to be matched
 */
@Override
protected void onResume()
{
    super.onResume();
    registerReceiver(mBroadcastReceiver, mIntentFilter);
}

/**
 * Unregister the broadcast receiver
 */
@Override
protected void onPause()
{
    super.onPause();
    unregisterReceiver(mBroadcastReceiver);
}
```

Listing B.3.3: Registering the `BroadcastReceiver`.

B.4 Discover neighbours

Now you have registered your application with the framework, you can activate proximity-based functionality and begin the process of searching for nearby devices, termed throughout as *neighbours*.

Listing B.4.1 demonstrates how to activate PBF by calling the `discoverNeighbours()` method of the `ProximityManager`, passing in an `ActionListener` callback interface that will be invoked on success or failure of the activation attempt. Following activation, the framework will begin to send broadcast intents to your application.

```java
mProximityManager.discoverNeighbours(mChannel,
                                     new ProximityManager.ActionListener()
{
    @Override
    public void onSuccess()
    {
        // PBF was activated successfully
    }

    @Override
    public void onFailure(int reasonCode)
    {
        // PBF activation failed
    }
});
```

Listing B.4.1: Discovering neighbours.

B.5 Retrieve the list of neighbours

Now that you have activated proximity-based functionality and begun the neighbour discovery process, the framework can be queried for the current list of neighbours. Listing B.5.1 demonstrates how to retrieve the neighbour list.

```
mProximityManager.requestNeighbours(mChannel,
        new ProximityManager.NeighbourListListener()
{
    @Override
    public void onNeighboursAvailable(List<ProximityDevice> neighbours)
    {
        ...
    }
});
```

Listing B.5.1: Retrieving the list of neighbours.

The `onNeighboursAvailable()` method of the supplied `NeighbourListListener` callback interface will be invoked asynchronously when the framework is ready to supply you with the list of neighbours.

The neighbour list contains the IP address for each device. This can be used to create direct connections to neighbours via TCP sockets, or any other appropriate method.

The neighbour list also contains the device name, IP address and current GPS location for each neighbour. This information can be used in whichever way you wish. There are endless possibilities!

B.6 Summary

This appendix has hopefully demonstrated how easy it is to discover nearby devices using the Proxima framework. Some advanced features such as service discovery and implementing device name editing capabilities are not given for brevity, but can easily be interpreted by studying the framework documentation given in Appendix C.

Appendix C

Proxima documentation – API reference

This appendix contains reference documentation for the Proxima Java API. This documentation is designed for use by application developers who wish to implement proximity-based functionality within their application using the Proxima framework.

The documentation was generated from Javadoc by the Doclava documentation tool, and rendered to PDF using the wkhtmltopdf library.

package
org.proxima

Interfaces

Channel.ChannelListener	Callback interface provided with `initialize(Context, ChannelListener)`.
ProximityManager.ActionListener	Callback interface for use with various API method calls.
ProximityManager.NeighbourListListener	Callback interface for use with `requestNeighbours(Channel, NeighbourListListener)`.
ProximityManager.ServiceResponseListener	Callback interface for use with `requestServices(Channel, ServiceResponseListener)` .

Classes

Channel	This class represents a channel connection between a client application and the Proxima framework.
EditDeviceNameActivity	This activity, when started, displays a dialog that allows the user to edit the name of the device.
LocationHelper	A class to track the current device GPS location.
MetadataServer	The MetadataServer is a HTTP server that is used by neighbours to request metadata about this device.
MetadataService	The MetadataService is responsible for maintaining up-to-date information about the local device and providing that information on-demand to other devices via HTTP.
NativeTaskHelper	A class to help perform native system tasks, such as running shell commands and changing file permissions.
ProximityDevice	A class representing a single device in the proximity-based network.
ProximityManager	ProximityManager This is the class that users of this API will interact with to discover neighbours in proximity.
ProximityService	The ProximityService is the main background service that responds to requests from client applications for proximity-based functionality such as neighbour discovery and retrieval (via Channel object).
ProximityServiceHelper	ProximityServiceHelper Simple helper class to perform basic tasks related to the ProximityService, such as enabling/disabling wifi and starting/stopping the routing protocol daemon.
RoutingHelper	This class helps with operations related to the OLSR routing protocol.
ServiceInfo	A class to represent a single service on the network.

public static interface

Channel.ChannelListener

org.proxima.Channel.ChannelListener

Class Overview

Callback interface provided with `initialize(Context, ChannelListener)`.

Summary

Public Methods	
abstract void	onChannelConnected() Called when the channel is initially connected and bound to the `ProximityService`.
abstract void	onChannelDisconnected() Called when the channel is disconnected from the `ProximityService`.

Public Methods

public abstract void **onChannelConnected** ()

Called when the channel is initially connected and bound to the `ProximityService`.

public abstract void **onChannelDisconnected** ()

Called when the channel is disconnected from the `ProximityService`.

public class

Channel

extends Object

java.lang.Object
↳org.proxima.Channel

Class Overview

This class represents a channel connection between a client application and the Proxima framework. A channel is required for all framework API calls. Channel instances are obtained by calling `initialize(Context, ChannelListener)`.

Summary

Nested Classes		
interface	Channel.ChannelListener	Callback interface provided with `initialize(Context, ChannelListener)`.

Public Constructors	
	Channel(Context context, Channel.ChannelListener listener) This constructor should not be used directly.

Public Methods	
void	connect() Connect to the `ProximityService`.
void	disconnect() Disconnect from the `ProximityService`.

Protected Methods	
Object	getListener(int key) Retrieve a callback listener from the internal store.
int	putListener(Object listener) Add a callback listener to the internal store.
void	sendMessage(int what, int arg1, int arg2) Send a message to the `ProximityService`.
void	sendMessage(int what, int arg1, int arg2, Bundle data) Send a message to the `ProximityService` with an extra data bundle.

Public Constructors

public **Channel** (Context context, Channel.ChannelListener listener)

This constructor should not be used directly. Obtain a Channel instance via `initialize(Context, ChannelListener)`.

Parameters

context	the client application context
listener	the callback listener to be invoked on channel connection/disconnection.

Public Methods

public void **connect** ()

Connect to the `ProximityService`.

public void **disconnect** ()

Disconnect from the `ProximityService`.

Protected Methods

protected Object **getListener** (int key)

Retrieve a callback listener from the internal store.

Parameters

key the map index key of the listener

Returns

the callback listener object if one was found with the matching key, null otherwise

protected int **putListener** (Object listener)

Add a callback listener to the internal store.

Parameters

listener the listener to store

Returns

the map index key of the listener

protected void **sendMessage** (int what, int arg1, int arg2)

Send a message to the `ProximityService`.

Parameters

what the message subject
arg1 the first int argument
arg2 the listener key

protected void **sendMessage** (int what, int arg1, int arg2, Bundle data)

Send a message to the `ProximityService` with an extra data bundle.

Parameters

what the message subject
arg1 the first int argument
arg2 the listener key
data the extra data bundle

public class
EditDeviceNameActivity
extends Activity

java.lang.Object
↳android.content.Context
↳android.content.ContextWrapper
↳android.view.ContextThemeWrapper
↳android.app.Activity
↳org.proxima.EditDeviceNameActivity

Class Overview

This activity, when started, displays a dialog that allows the user to edit the name of the device.

Summary

Public Constructors	
	EditDeviceNameActivity()

Protected Methods	
void	onCreate(Bundle savedInstanceState) Called when the activity is first created.

Public Constructors

public **EditDeviceNameActivity** ()

Protected Methods

protected void **onCreate** (Bundle savedInstanceState)

Called when the activity is first created.

See Also

onCreate(android.os.Bundle)

public class
LocationHelper

extends Object
implements LocationListener

java.lang.Object
↳org.proxima.LocationHelper

Class Overview

A class to track the current device GPS location. GPS information can come from two sources: GPS_PROVIDER and NETWORK_PROVIDER.

Summary

Public Constructors	
	LocationHelper(Context context) Constructor.

Public Methods	
Location	getCurrentLocation() Get the current GPS location.
void	onLocationChanged(Location location) Called when the device GPS location changes.
void	onProviderDisabled(String provider) Called when a GPS provider (GPS_PROVIDER or NETWORK_PROVIDER) is disabled.
void	onProviderEnabled(String provider) Called when a GPS provider (GPS_PROVIDER or NETWORK_PROVIDER) is enabled.
void	onStatusChanged(String provider, int status, Bundle extras) Called when a provider status changes.

Public Constructors

public **LocationHelper** (Context context)

Constructor.

Parameters
context the client application context

Public Methods

public Location **getCurrentLocation** ()

Get the current GPS location.

Returns
the current GPS location

public void **onLocationChanged** (Location location)

Called when the device GPS location changes.

See Also
onLocationChanged(android.location.Location)

public void **onProviderDisabled** (String provider)

Called when a GPS provider (GPS_PROVIDER or NETWORK_PROVIDER) is disabled.

See Also

onProviderDisabled(java.lang.String)

public void **onProviderEnabled** (String provider)

Called when a GPS provider (GPS_PROVIDER or NETWORK_PROVIDER) is enabled.

See Also

onProviderEnabled(java.lang.String)

public void **onStatusChanged** (String provider, int status, Bundle extras)

Called when a provider status changes.

See Also

onStatusChanged(java.lang.String, int, android.os.Bundle)

public class
MetadataServer

extends NanoHTTPD
implements Runnable

java.lang.Object
↳fi.iki.elonen.NanoHTTPD
↳org.proxima.MetadataServer

Class Overview

The MetadataServer is a HTTP server that is used by neighbours to request metadata about this device.

Summary

Constants		
int	LISTEN_PORT	The port that the HTTP metadata server listens on.

Public Constructors	
	MetadataServer(Context context) Constructor.

Public Methods	
void	run() Start the HTTP server thread.
NanoHTTPD.Response	serve(String uri, NanoHTTPD.Method method, Map<String, String> header, Map<String, String> params, Map<String, String> files) Serve a metadata request.
void	updateServices(ArrayList<ServiceInfo> services) Update the current list of local services.

Constants

public static final int **LISTEN_PORT**

The port that the HTTP metadata server listens on.

Constant Value: 8080 (0x00001f90)

Public Constructors

public **MetadataServer** (Context context)

Constructor.

Parameters
context the client application context

Public Methods

public void **run** ()

Start the HTTP server thread.

See Also
run()

public NanoHTTPD.Response **serve** (String uri, NanoHTTPD.Method method, Map<String, String> header, Map<String, String> params, Map<String, String> files)

Serve a metadata request.

Parameters

uri	Percent-decoded URI without parameters, for example "/index.cgi"
method	"GET", "POST" etc.
header	Header entries, percent decoded
params	Parsed, percent decoded parameters from URI and, in case of POST, data.

Returns

HTTP response, see class Response for details

See Also

`serve(java.lang.String, fi.iki.elonen.NanoHTTPD.Method, java.util.Map, java.util.Map, java.util.Map)`

public void **updateServices** (ArrayList<ServiceInfo> services)

Update the current list of local services.

Parameters

services	the new service list

public class
MetadataService
extends Service

java.lang.Object
↳android.content.Context
↳android.content.ContextWrapper
↳android.app.Service
↳org.proxima.MetadataService

Class Overview

The MetadataService is responsible for maintaining up-to-date information about the local device and providing that information on-demand to other devices via HTTP.

Summary

Public Constructors	
	MetadataService()

Public Methods	
IBinder	onBind(Intent intent) Called when an activity or service calls `bindService(Intent, android.content.ServiceConnection, int)` .
void	onCreate() Called when the service is first created.
int	onStartCommand(Intent intent, int flags, int startId) Called when an activity or service calls `startService(Intent)`.

Public Constructors

public **MetadataService** ()

Public Methods

public IBinder **onBind** (Intent intent)

Called when an activity or service calls `bindService(Intent, android.content.ServiceConnection, int)` .

See Also
`onBind(android.content.Intent)`

public void **onCreate** ()

Called when the service is first created.

See Also
`onCreate()`

public int **onStartCommand** (Intent intent, int flags, int startId)

Called when an activity or service calls `startService(Intent)`.

See Also
`onStartCommand(android.content.Intent, int, int)`

public class
NativeTaskHelper
extends Object

java.lang.Object
↳org.proxima.NativeTaskHelper

Class Overview

A class to help perform native system tasks, such as running shell commands and changing file permissions.

Summary

Public Constructors	
	NativeTaskHelper()

Public Methods	
static String	chmod(Context context, String path, String mode) Change access permissions of a file or directory.
static String	getBasePath(Context context) Get the base path where the Proxima binaries and configuration files are located.
static InetAddress	getIpAddress() Get the current IP address for this device.
static boolean	isProcessRunning(String processName) Check if a particular process is running on the system.
static boolean	killProcess(Context context, String processName) Forcibly stop a running process.
static int	runCommand(String command) Execute a shell command.
static String	runCommandGetOutput(String command) Execute a shell command and get the text output that was produced by the command.
static int	runRootCommand(Context context, String command) Execute a shell command with root privileges.
static String	runRootCommandGetOutput(Context context, String command) Execute a shell command with root permissions and get the text output that was produced by the command.
static Process	runRootCommandInBackground(Context context, String command) Execute a shell command with root privileges and place it in the background.
static void	unpackResources(Context context) Unpack the core Proxima binaries and configuration files from the raw Android resource locations to their proper place on the device filesystem.

Public Constructors

public **NativeTaskHelper** ()

Public Methods

public static String **chmod** (Context context, String path, String mode)

Change access permissions of a file or directory.

Parameters

context	the client application context
path	the path of the file or directory
mode	the new access permission string (such as "0644")

mode the new access permission string (such as "644")

Returns
the shell output that was produced

public static String **getBasePath** (Context context)

Get the base path where the Proxima binaries and configuration files are located.

Parameters
context the client application context

Returns
the base file path

public static InetAddress **getIpAddress** ()

Get the current IP address for this device.

Returns
the IP address

public static boolean **isProcessRunning** (String processName)

Check if a particular process is running on the system.

Parameters
processName the name of the process

Returns
true if the process is currently running, false otherwise

public static boolean **killProcess** (Context context, String processName)

Forcibly stop a running process.

Parameters
context the client application context
processName the name of the process

Returns
true if the process was killed, false otherwise

public static int **runCommand** (String command)

Execute a shell command.

Parameters
command the command to run

Returns
the shell exit code that the command produced

public static String **runCommandGetOutput** (String command)

Execute a shell command and get the text output that was produced by the command.

Parameters
command the command to run

Returns
the output of the command

public static int **runRootCommand** (Context context, String command)

Execute a shell command with root privileges.

Parameters
context the client application context
command the command to run

Returns
the shell exit code that the command produced.

public static String **runRootCommandGetOutput** (Context context, String command)

Execute a shell command with root permissions and get the text output that was produced by the command.

Parameters

context the client application context
command the connamd to run

Returns
the output of the command

public static Process **runRootCommandInBackground** (Context context, String command)

Execute a shell command with root privileges and place it in the background.

Parameters
context the client application context
command the command to run

Returns
a Process object representing the shell command

public static void **unpackResources** (Context context)

Unpack the core Proxima binaries and configuration files from the raw Android resource locations to their proper place on the device filesystem.

Parameters
context the client application context

public class
ProximityDevice

extends Object
implements Parcelable

java.lang.Object
↳org.proxima.ProximityDevice

Class Overview

A class representing a single device in the proximity-based network.

Summary

Fields		
public static final Creator<ProximityDevice>	CREATOR	Implement the Parcelable interface.

Public Constructors	
	ProximityDevice()
	ProximityDevice(String deviceName, String deviceAddress, Location deviceLocation)

Public Methods	
int	describeContents() Implement the Parcelable interface.
String	getDeviceAddress() Get the IP address of this device.
double	getDeviceLatitude() Get the currently known latitude of this device.
double	getDeviceLongitude() Get the currently known longitude of this device.
String	getDeviceName() Get the name of this device.
String	toString() Return a human-readable representation of this object.
void	writeToParcel(Parcel out, int flags) Implement the Parcelable interface.

Fields

public static final Creator<ProximityDevice> **CREATOR**

Implement the Parcelable interface.

Public Constructors

public **ProximityDevice** ()

public **ProximityDevice** (String deviceName, String deviceAddress, Location deviceLocation)

Public Methods

public int **describeContents** ()

Implement the Parcelable interface

Implement the Parcelable interface.

See Also

describeContents()

public String **getDeviceAddress** ()

Get the IP address of this device.

Returns

the IP address

public double **getDeviceLatitude** ()

Get the currently known latitude of this device.

Returns

the device GPS latitude.

public double **getDeviceLongitude** ()

Get the currently known longitude of this device.

Returns

the device GPS longitude.

public String **getDeviceName** ()

Get the name of this device.

Returns

the device name

public String **toString** ()

Return a human-readable representation of this object.

See Also

toString()

public void **writeToParcel** (Parcel out, int flags)

Implement the Parcelable interface.

See Also

writeToParcel(android.os.Parcel, int)

public static interface

ProximityManager.ActionListener

org.proxima.ProximityManager.ActionListener

Class Overview

Callback interface for use with various API method calls.

Summary

Public Methods	
abstract void	onFailure(int reason) Called when the requested action failed to complete successfully.
abstract void	onSuccess() Called when the requested action completed successfully.

Public Methods

public abstract void **onFailure** (int reason)

Called when the requested action failed to complete successfully.

Parameters

reason the failure code

public abstract void **onSuccess** ()

Called when the requested action completed successfully.

public class
ProximityManager
extends Object

java.lang.Object
↳org.proxima.ProximityManager

Class Overview

ProximityManager This is the class that users of this API will interact with to discover neighbours in proximity. This API takes inspiration from the Android Wi-Fi Peer-to-Peer API: http://developer .android.com/reference/android/net/wifi/p2p/package-summary.html

Summary

Nested Classes		
interface	ProximityManager.ActionListener	Callback interface for use with various API method calls.
interface	ProximityManager.NeighbourListListener	Callback interface for use with `requestNeighbours(Channel, NeighbourListListener)`.
interface	ProximityManager.ServiceResponseListener	Callback interface for use with `requestServices(Channel, ServiceResponseListener)` .

Constants		
int	DISCOVER_NEIGHBOURS	Action key for neighbour discovery
int	DISCOVER_NEIGHBOURS_FAILED	Indicates that neighbour discovery failed
int	DISCOVER_NEIGHBOURS_SUCCEEDED	Indicates that neighbour discovery succeeded
String	EXTRA_NEIGHBOUR_LIST	The lookup key for the new neighbour list when PROXIMITY_NEIGHBOURS_CHANGED_ACTION broadcast is sent.
String	EXTRA_PROXIMITY_DEVICE	The bundle lookup key for a single ProximityDevice object
String	EXTRA_PROXIMITY_STATE	The lookup key for an int that indicates whether proximity functionality is enabled or disabled.
String	EXTRA_SERVICE_INFO	The bundle lookup key for a single ServiceInfo object
String	EXTRA_SERVICE_LIST	The bundle lookup key for the new service list
String	PROXIMITY_NEIGHBOURS_CHANGED_ACTION	Broadcast intent action indicating that the available neighbour list has changed.
String	PROXIMITY_STATE_CHANGED_ACTION	Broadcast intent action to indicate whether Wi-Fi p2p is enabled or disabled.
int	PROXIMITY_STATE_DISABLED	Indicates that proximity functionality is disabled.
int	PROXIMITY_STATE_ENABLED	Indicates that proximity functionality is enabled.
int	REGISTER_SERVICE	Action key for service registration request
int	REGISTER_SERVICE_FAILED	Indicates that a service registration failed
int	REGISTER_SERVICE_SUCCEEDED	Indicates that a service registration succeeded
int	REQUEST_NEIGHBOURS	Action key for neighbour request
int	REQUEST_SERVICES	Action key for a service list request
int	RESPONSE_NEIGHBOURS	Response key for a neighbour request
int	RESPONSE_SERVICES	Response key for a service list request
String	SETTING_PROXIMITY_DEVICE_NAME	Namee for the device name system setting
int	SET_DEVICE_NAME	Action key for a device name change request
int	SET_DEVICE_NAME_FAILED	Indicates that a device name change failed
int	SET_DEVICE_NAME_SUCCEEDED	Indicates that a device name change succeeded
int	UNREGISTER_SERVICE	Action key for service unregistration request

int	UNREGISTER_SERVICE_FAILED	Indicates that a service unregistration failed
int	UNREGISTER_SERVICE_SUCCEEDED	Indicates that a service unregistration succeeded

Public Methods	
void	discoverNeighbours(Channel channel) Start the neighbour discovery process
void	discoverNeighbours(Channel channel, ProximityManager.ActionListener listener) Start the neighbour discovery process and receive a callback on success/ failure to the given action listener.
static ProximityManager	getInstance() Use this method to obtain a reference to the ProximityManager.
Channel	initialize(Context context) Initialize the proximity manager (connect to the proximity service)
Channel	initialize(Context context, Channel.ChannelListener channelListener) Initialize the proximity manager (connect to the proximity service) and receive a callback on success/failure to the given callback listener.
void	registerLocalService(Channel channel, ServiceInfo info, ProximityManager.ActionListener listener) Register a local service as being available on this device.
void	requestNeighbours(Channel channel, ProximityManager.NeighbourListListener listener) Request the current list of neighbours
void	requestServices(Channel channel, ProximityManager.ServiceResponseListener listener) Request the available services on the network.
void	setDeviceName(Channel channel, ProximityManager.ActionListener listener) Request a device name change.
void	unregisterLocalService(Channel channel, ServiceInfo info, ProximityManager.ActionListener listener) Unregister a local service.

Constants

public static final int **DISCOVER_NEIGHBOURS**

Action key for neighbour discovery

Constant Value: 2 (0x00000002)

public static final int **DISCOVER_NEIGHBOURS_FAILED**

Indicates that neighbour discovery failed

Constant Value: 3 (0x00000003)

public static final int **DISCOVER_NEIGHBOURS_SUCCEEDED**

Indicates that neighbour discovery succeeded

Constant Value: 4 (0x00000004)

public static final String **EXTRA_NEIGHBOUR_LIST**

The lookup key for the new neighbour list when PROXIMITY_NEIGHBOURS_CHANGED_ACTION broadcast is sent.

Constant Value: "neighbourList"

public static final String **EXTRA_PROXIMITY_DEVICE**

The bundle lookup key for a single ProximityDevice object

Constant Value: "proximityDevice"

public static final String **EXTRA_PROXIMITY_STATE**

The lookup key for an int that indicates whether proximity functionality is enabled or disabled.

Constant Value: "proximityState"

public static final String **EXTRA_SERVICE_INFO**

The bundle lookup key for a single ServiceInfo object

Constant Value: "serviceInfo"

public static final String **EXTRA_SERVICE_LIST**

The bundle lookup key for the new service list

Constant Value: "serviceList"

public static final String **PROXIMITY_NEIGHBOURS_CHANGED_ACTION**

Broadcast intent action indicating that the available neighbour list has changed.

Constant Value: "org.proxima.NEIGHBOURS_CHANGED"

public static final String **PROXIMITY_STATE_CHANGED_ACTION**

Broadcast intent action to indicate whether Wi-Fi p2p is enabled or disabled.

Constant Value: "org.proxima.STATE_CHANGED"

public static final int **PROXIMITY_STATE_DISABLED**

Indicates that proximity functionality is disabled.

Constant Value: 1 (0x00000001)

public static final int **PROXIMITY_STATE_ENABLED**

Indicates that proximity functionality is enabled.

Constant Value: 0 (0x00000000)

public static final int **REGISTER_SERVICE**

Action key for service registration request

Constant Value: 7 (0x00000007)

public static final int **REGISTER_SERVICE_FAILED**

Indicates that a service registration failed

Constant Value: 9 (0x00000009)

public static final int **REGISTER_SERVICE_SUCCEEDED**

Indicates that a service registration succeeded

Constant Value: 8 (0x00000008)

public static final int **REQUEST_NEIGHBOURS**

Action key for neighbour request

Constant Value: 5 (0x00000005)

public static final int **REQUEST_SERVICES**

Action key for a service list request

Constant Value: 13 (0x0000000d)

public static final int **RESPONSE_NEIGHBOURS**

Response key for a neighbour request

Constant Value: 6 (0x00000006)

public static final int **RESPONSE_SERVICES**

Response key for a service list request

Constant Value: 14 (0x0000000e)

public static final String **SETTING_PROXIMITY_DEVICE_NAME**

Namee for the device name system setting

Constant Value: "proximity_device_name"

public static final int **SET_DEVICE_NAME**

Action key for a device name change request

Constant Value: 15 (0x0000000f)

public static final int **SET_DEVICE_NAME_FAILED**

Indicates that a device name change failed

Constant Value: 17 (0x00000011)

public static final int **SET_DEVICE_NAME_SUCCEEDED**

Indicates that a device name change succeeded

Constant Value: 16 (0x00000010)

public static final int **UNREGISTER_SERVICE**

Action key for service unregistration request

Constant Value: 10 (0x0000000a)

public static final int **UNREGISTER_SERVICE_FAILED**

Indicates that a service unregistration failed

Constant Value: 12 (0x0000000c)

public static final int **UNREGISTER_SERVICE_SUCCEEDED**

Indicates that a service unregistration succeeded

Constant Value: 11 (0x0000000b)

Public Methods

public void **discoverNeighbours** (Channel channel)

Start the neighbour discovery process

Parameters

channel the client channel instance

public void **discoverNeighbours** (Channel channel, ProximityManager.ActionListener listener)

Start the neighbour discovery process and receive a callback on success/ failure to the given action listener.

Parameters

channel the client channel instance

listener the client callback listener to be notified on success/failure

public static ProximityManager **getInstance** ()

Use this method to obtain a reference to the ProximityManager.

Returns

the singleton instance of this class

public Channel **initialize** (Context context)

Initialize the proximity manager (connect to the proximity service)

Parameters

context the client context

Returns

a Channel object to be used with future API requests

public Channel **initialize** (Context context, Channel.ChannelListener channelListener)

Initialize the proximity manager (connect to the proximity service) and receive a callback on success/failure to the given callback listener.

Parameters

context	the client context
channelListener	the callback listener to be notified on channel connection/disconnection

Returns

a Channel object to be used with future API requests

public void **registerLocalService** (Channel channel, ServiceInfo info, ProximityManager.ActionListener listener)

Register a local service as being available on this device.

Parameters

channel	the client channel instance
info	the service info object containing information about the service to be registered
listener	the client callback listener to be notified when the service is registered successfully or when an error occurs

public void **requestNeighbours** (Channel channel, ProximityManager.NeighbourListListener listener)

Request the current list of neighbours

Parameters

channel	the client channel instance
listener	the client callback listener to be notified when the list of neighbours is available

public void **requestServices** (Channel channel, ProximityManager.ServiceResponseListener listener)

Request the available services on the network. Since devices can have multiple services registered, the supplied `ProximityManager.ServiceResponseListener` will be called multiple times; once per device along with the list of services for that particular device.

Parameters

channel	the client channel instance
listener	the service response listener to be called when the service info is available for a device.

public void **setDeviceName** (Channel channel, ProximityManager.ActionListener listener)

Request a device name change. This will cause a dialog to appear, prompting the device user to enter the device name. If the user submits the dialog, the `onSuccess()` method is called. If the user cancels the dialog, the `onFailure(int)` method is called.

Parameters

channel	the client channel instance
listener	the action listener to be notified when the user submits or cancels the name change dialog

public void **unregisterLocalService** (Channel channel, ServiceInfo info, ProximityManager.ActionListener listener)

Unregister a local service.

Parameters

channel	the client channel instance
info	the service info object that was previously registered with `registerLocalService(Channel, ServiceInfo, ActionListener)`
listener	the client callback listener to be notified when the service is unregistered successfully or when an error occurs

public static interface

ProximityManager.NeighbourListListener

org.proxima.ProximityManager.NeighbourListListener

Class Overview

Callback interface for use with `requestNeighbours(Channel, NeighbourListListener)`.

Summary

Public Methods	
abstract void	onNeighboursAvailable(List<ProximityDevice> neighbours) Called when the list of neighbours is available.

Public Methods

public abstract void **onNeighboursAvailable** (List<ProximityDevice> neighbours)

Called when the list of neighbours is available.

Parameters

neighbours the list of neighbours

public static interface

ProximityManager.ServiceResponseListener

org.proxima.ProximityManager.ServiceResponseListener

Class Overview

Callback interface for use with `requestServices(Channel, ServiceResponseListener)` .

Summary

Public Methods	
abstract void	onServicesAvailable(List<ServiceInfo> services, ProximityDevice device) Called when the list of services is available.

Public Methods

public abstract void **onServicesAvailable** (List<ServiceInfo> services, ProximityDevice device)

Called when the list of services is available.

Parameters

services	the list of services
device	the device associated with the service list

public class
ProximityServiceHelper
extends Object

java.lang.Object
↳org.proxima.ProximityServiceHelper

Class Overview

ProximityServiceHelper Simple helper class to perform basic tasks related to the ProximityService, such as enabling/disabling wifi and starting/stopping the routing protocol daemon.

Summary

Public Constructors	
	ProximityServiceHelper(Context context) Constructor

Public Methods	
void	configureWirelessInterface(String ip) Configure the wireless interface to ad-hoc mode.
void	disableWifi() Disable the default wifi interface for this device
void	enableWifi() Enable the default wifi interface for this device
boolean	isInterfaceConfigured() Query the configuration state of the wireless interface.
boolean	isRoutingProtocolStarted() Query the running status of the routing protocol
Collection<Neighbor>	requestNeighbours() Request the current list of neighbours from the routing protocol interface
boolean	startRoutingProtocol() Start the routing protocol daemon (currently olsrd)
boolean	stopRoutingProtocol() Stop the routing protocol

Public Constructors

public **ProximityServiceHelper** (Context context)

Constructor

Parameters
context the parent context reference

Public Methods

public void **configureWirelessInterface** (String ip)

Configure the wireless interface to ad-hoc mode. TODO: don't set fixed IP address TODO: use edify script for speed

Parameters
ip the ip address to use for this device

public void **disableWifi** ()

Disable the default wifi interface for this device

public void **enableWifi** ()

Enable the default wifi interface for this device

public boolean **isInterfaceConfigured** ()

Query the configuration state of the wireless interface.

Returns
true if the interface is configured, false otherwise

public boolean **isRoutingProtocolStarted** ()

Query the running status of the routing protocol

Returns
true if the routing protocol is currently running, false otherwise

public Collection<Neighbor> **requestNeighbours** ()

Request the current list of neighbours from the routing protocol interface

Returns
the current neighbour list

public boolean **startRoutingProtocol** ()

Start the routing protocol daemon (currently olsrd)

Returns
true if the routing protocol was successfully started, false otherwise

public boolean **stopRoutingProtocol** ()

Stop the routing protocol

Returns
true if the routing protocol was successfully stopped, false otherwise

public class
ProximityService
extends Service

java.lang.Object
↳android.content.Context
↳android.content.ContextWrapper
↳android.app.Service
↳org.proxima.ProximityService

Class Overview

The ProximityService is the main background service that responds to requests from client applications for proximity-based functionality such as neighbour discovery and retrieval (via Channel object).

Summary

Public Constructors	
	ProximityService()

Public Methods	
IBinder	onBind(Intent intent) Return the communication channel to this service.
void	onCreate() Called when the service is first created
void	onDestroy() Called when the service is finishing or being destroyed by the system
int	onStartCommand(Intent intent, int flags, int startId) Called by the system every time a client explicitly starts the service.

Protected Methods	
boolean	discoverNeighbours() Begin the neighbour discovery process.
boolean	registerService(ServiceInfo info) Register a new service for discovery by neighbouring devices.
Bundle	requestNeighbours() Obtain a list of neighbours from the routing protocol daemon interface and return it back to the client.
void	requestServices(Message source) Retrieve the available services on the network.
void	setDeviceName(Message source) Start the activity that displays the device name edit dialog.
boolean	unregisterService(ServiceInfo info) Unregister a previously registered service.

Public Constructors

public **ProximityService** ()

Public Methods

public IBinder **onBind** (Intent intent)

Return the communication channel to this service.

See Also
onBind(android.content.Intent)

public void **onCreate** ()

Called when the service is first created

See Also

onCreate()

public void **onDestroy** ()

Called when the service is finishing or being destroyed by the system

See Also

onDestroy()

public int **onStartCommand** (Intent intent, int flags, int startId)

Called by the system every time a client explicitly starts the service.

See Also

onStartCommand(android.content.Intent, int, int)

Protected Methods

protected boolean **discoverNeighbours** ()

Begin the neighbour discovery process. This involves unpacking the necessary binary files and config files, configuring the wireless interface (setting ad-hoc mode) and starting the routing protocol daemon. If the neighbour discovery process started correctly (i.e. we performed the aforementioned steps successfully) then we respond to the client with a DISCOVER_NEIGHBOURS_SUCCEEDED message, and additionally broadcast a PROXIMITY_NEIGHBOURS_CHANGED_ACTION intent. Otherwise, we respond to the client with a DISCOVER_NEIGHBOURS_FAILED message.

protected boolean **registerService** (ServiceInfo info)

Register a new service for discovery by neighbouring devices.

Parameters

info the object containing info about the service to be registered

Returns

true if the service was successfully registered, false otherwise

protected Bundle **requestNeighbours** ()

Obtain a list of neighbours from the routing protocol daemon interface and return it back to the client.

Returns

a Bundle containing the list of neighbours.

protected void **requestServices** (Message source)

Retrieve the available services on the network.

Parameters

source the original client message used as the reply-to address

protected void **setDeviceName** (Message source)

Start the activity that displays the device name edit dialog.

Parameters

source the original client message used as the reply-to address

protected boolean **unregisterService** (ServiceInfo info)

Unregister a previously registered service.

Parameters

info the object containing info about the service to be unregistered

Returns

true if the service was successfully unregistered, false otherwise

public class
RoutingHelper
extends Object

java.lang.Object
↳org.proxima.RoutingHelper

Class Overview

This class helps with operations related to the OLSR routing protocol.

Summary

Public Constructors	
	RoutingHelper(Context context)

Public Methods	
boolean	isDaemonRunning() Check if the routing daemon is currently running.
Collection<Neighbor>	requestNeighbours() Request the current list of neighbours from the routing daemon.
boolean	startDaemon() Start the routing daemon.
boolean	stopDaemon() Stop the routing daemon.

Public Constructors

public **RoutingHelper** (Context context)

Public Methods

public boolean **isDaemonRunning** ()

Check if the routing daemon is currently running.

Returns
true if the daemon is running, false otherwise

public Collection<Neighbor> **requestNeighbours** ()

Request the current list of neighbours from the routing daemon.

Returns
the list of neighbours

public boolean **startDaemon** ()

Start the routing daemon.

Returns
true if the daemon was started successfully, false otherwise

public boolean **stopDaemon** ()

Stop the routing daemon.

Returns
true if the daemon was stopped successfully, false otherwise

public class
ServiceInfo

extends Object
implements Parcelable

java.lang.Object
↳org.proxima.ServiceInfo

Class Overview

A class to represent a single service on the network.

Summary

Fields		
public static final Creator<ServiceInfo>	CREATOR	Implement the Parcelable interface.

Public Constructors	
	ServiceInfo(String uuid, String serviceName) Constructor.

Public Methods	
int	describeContents() Implement the Parcelable interface.
String	getServiceName() Get the service name.
String	getUuid() Get the service universally unique identifier.
String	toString() Return a human-readable representation of this service.
void	writeToParcel(Parcel out, int flags) Implement the Parcelable interface.

Fields

public static final Creator<ServiceInfo> **CREATOR**

Implement the Parcelable interface.

Public Constructors

public **ServiceInfo** (String uuid, String serviceName)

Constructor.

Parameters

uuid	the universally unique identifier for this service.
serviceName	the name of this service.

Public Methods

public int **describeContents** ()

Implement the Parcelable interface.

See Also

describeContents()

public String **getServiceName** ()

Get the service name.

Returns
the service name

public String **getUuid** ()

Get the service universally unique identifier.

Returns
the service UUID

public String **toString** ()

Return a human-readable representation of this service.

See Also
`toString()`

public void **writeToParcel** (Parcel out, int flags)

Implement the Parcelable interface.

See Also
`writeToParcel(android.os.Parcel, int)`

Appendix D

IEEE MS 2014 conference paper

This appendix contains the paper that was submitted for peer-review to the 3rd IEEE International Conference on Mobile Services (MS 2014).

A Proximity-Based Framework for Mobile Services

Justin Lewis Salmon Rong Yang
Department of Computer Science and Creative Technologies
University of the West of England
Bristol, United Kingdom
Email: justin2.salmon@live.uwe.ac.uk
rong.yang@uwe.ac.uk

***Abstract*—Peer-to-peer proximity-based wireless networking can provide improved spatial and temporal semantics and independence over alternative wireless topologies that rely on static network infrastructure, and can potentially enable new classes of mobile applications. However, the difficulties of setting up such ad-hoc connections has thus far been a development barrier. There is currently a need for an abstraction tool to allow application developers to exploit the potential advantages of such networks with minimal knowledge of the underlying connectivity. We present Proxima, a framework for the Android platform, which employs ad-hoc device-to-device connections and proactive mesh routing for a decentralised topology with solely proximity-based rich content dissemination. The framework is designed to be developer- and user-friendly with minimal configuration effort, lightweight, reusable and hardware independent. After compilation, the size of its binary distribution is only 6MB. We have further developed a real-life application, named TuneSpy, based around sharing music with local peers. The development of TuneSpy has produced two positive outcomes. Firstly, it strongly demonstrates the ease of writing proximity-based applications with Proxima. Secondly, it has served as a testing platform for all framework functionality.**

***Keywords*-Networking; proximity-based; mobile; ad-hoc; peer-to-peer.**

I. Introduction

Location-based applications have had several successes in the mobile arena. These applications have usually relied on traditional infrastructure networks such as Wi-Fi or cellular towers for timely publication of the location information to a centralised server, where the location information is determined by the users current GPS location. Some location-based applications, such as Foursquare [1], have traditionally oriented around the mobile device user arriving within proximity of a particular fixed location, while others have also oriented around the location of other nearby devices.

A. The proximity-based paradigm

This paper aims to explore a different paradigm, whereby reliance is solely upon mobile users being within proximity of one another, in a peer-to-peer orientation. Publication of information is also solely to proximal neighbours, and thus does not rely on additional networking infrastructure. We refer to this paradigm as the *proximity-based* paradigm throughout the remainder of the paper.

The concept of purely local, ephemeral, decentralised information dissemination has several apparent advantages. The ability to sense and connect with those around us allows purely real-time and real-space communication and collaboration. It gives us additional spatial and temporal semantics, i.e. a sense of *"here-and-now"*, on which to build applications upon [2]. The lack of infrastructure dependence also enables communication in a far wider range of locations.

There are unique potentials for mobile applications that might necessitate or otherwise exploit these *"here-and-now"* semantics, such as social, multimedia, crisis management, and gaming. There are some scenarios where dissemination onto the wider Internet is not desired, not useful, or simply not necessary due to the inherent importance of the temporal aspect.

B. A scenario

To illustrate the idea of proximity-based computing, let us imagine a scenario where this paradigm might apply. Consider, for example, a hypothetical train journey. Two people, Alice and Bob, are both passengers on this journey. Bob happens to be using his smartphone to listen to a track by his favourite music artist through a pair of headphones. Alice, who also has a smartphone, notices Bob, and begins to wonder what he might be listening to. She wishes that there were some way for her to tune-in to what Bob is currently listening to (assuming that Bob has pre-authorised this type of sharing behaviour), as if Bob were able to broadcast his current track to the people around him. Assuming that this train does not have a Wi-Fi access point available, how is Alice able to listen to Bob? How might their smartphones be able to connect directly to each other, simply based on the fact that they are in proximity to each other? The advanced capabilities of their devices should surely allow this behaviour.

The *"here-and-now"* semantics of this encounter are clear. Alice must be spatially close to Bob, and the information she is able to retrieve is temporally dependent, i.e. Bob will only be listening to a particular track for a certain period of time; after that time, the information is irrelevant and

unobtainable. Such is the ephemeral nature of the proximity-based paradigm.

C. The problem

As it stands today, there is a lack of a reusable platform on which to build applications that can make use of the aforementioned functionality. This is in part due to the complexity and difficulty of setting up such peer-to-peer connections, and the various underlying technologies that could potentially be used. Additionally, unlike traditional infrastructure connections, issues such as IP address assignment and service discovery do not have standard precedents in peer-to-peer networks.

This paper aims to tackle these issues, using modern tools to create a platform that is capable of operation regardless of infrastructure availability, and reaches as close to zero-configuration as possible. We present a reusable platform to allow future application developers to exploit this temporal and spatial information. Due to the myriad of available mobile devices and their varying levels of compatibility and interoperability, we will initially build on the Android platform, whilst still aiming to support as wide a range of devices as possible.

The rest of the paper is organised as follows. The next section discusses the background to the paper and related work. Section 3 describes our work, the from the top level view to implementation details. Section 4 discusses testing and evaluation. Finally we conclude with Section 5.

II. Background and Related Work

This section analyses the technologies that one might use to implement peer-to-peer mobile connectivity and the rationale behind the selections that have been made for our implementation. Some existing solutions are outlined and compared with our implementation.

A. Background

Two major technologies that allow wireless connection between mobile devices are Bluetooth and Wi-Fi. The former is standardised by the Bluetooth Special Interest Group, and the latter standardised by the IEEE. Both are available on most modern mobile phones, tablets, laptops, and other electronic devices.

Bluetooth uses short-wavelength radio waves; the data transmission speed is comparatively slow. It is best suited to low-bandwidth peer-to-peer applications. Wi-Fi can achieve much higher transmission speed and wider range than Bluetooth. Therefore Wi-Fi technology has dominated for most high-bandwidth internet-based applications.

However, we know that under standard Wi-Fi (otherwise known as Basic Service Set (BSS) for single access points or Extended Service Set (ESS) for multiple access points), the connection usually relies upon being within range of a fixed wireless network access point. This is something against our proximity-based paradigm. For a modern mobile platform, we need a technology like Bluetooth which allows direct device-to-device connection but with the speed and operable range benefits of a technology like Wi-Fi.

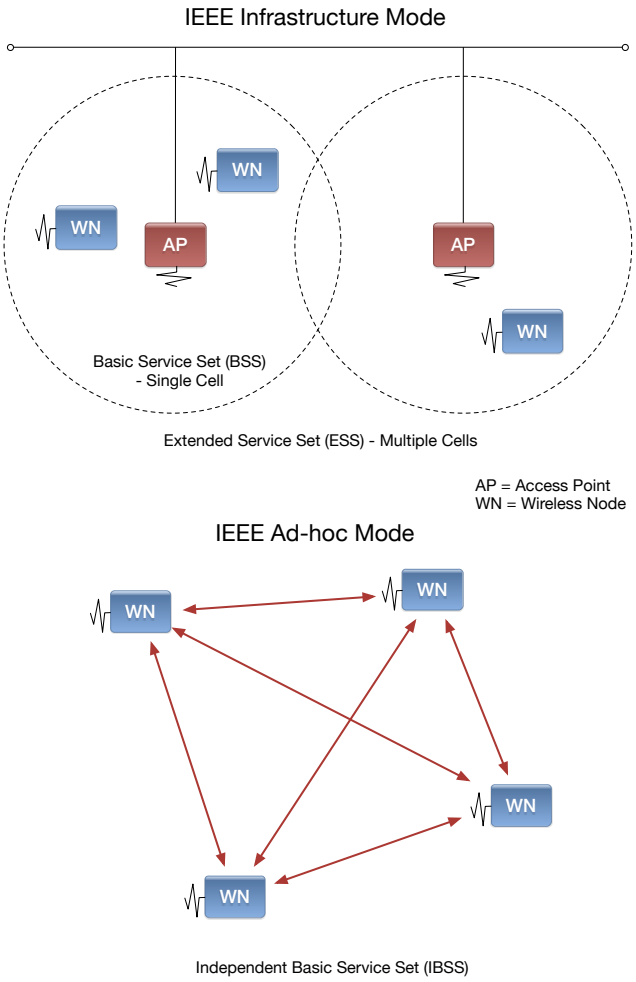


Figure 1. IEEE 802.11 operational modes.

The proposed solution is to use Wi-Fi ad-hoc mode, otherwise known as Independent Basic Service Set (IBSS), which is an alternative operation mode allowing connection without an access point. With Wi-Fi ad-hoc mode, we can achieve device-to-device communication like Bluetooth yet at a high efficient Wi-Fi speed. See Figure 1. for an illustration of infrastructure Wi-Fi vs ad-hoc Wi-Fi. This mode of operation is supported by most network interface drivers. It is, however, not supported by default on Android devices. Some configuration is needed to enable it, including gaining root access to the device.

This is exactly what we need for achieving a real-time mobile platform. However, Wi-Fi ad-hoc mode only provides the basic connectivity infrastructure. In order for it to be

useful for complex applications, more software is needed to build on to achieve functions such as automatic network configuration, routing and service discovery. These problems have burdened mobile application developers wishing to write applications leveraging the potential benefits of Wi-Fi ad-hoc mode. This is largely factored by the tediousness of configuring such networks, and by the diversity of competing strategies and protocols.

Our goal was to build a general framework on top of ad-hoc Wi-Fi, which allows people to enjoy the benefits of ad-hoc Wi-Fi without worrying about any low level issues. Our aim is to make the framework user-friendly, hardware-independent, powerful, lightweight and reusable.

B. Related Work

Wi-Fi Direct [3] is a piece of software which does not use ad-hoc Wi-Fi. It provides a good interface allowing people to develop mobile applications. The main idea of Wi-Fi Direct is to create a software access point to active connectivity without a real access point. It is a well established software and available only on newer mobile devices. So Wi-Fi Direct is hardware-dependent, that is against one of our initial aims. Moreover, Wi-Fi Direct is not capable of multi-hop routing, and requires a manual setup procedure (Wi-Fi Protected Setup). These are the major differences from our work.

The Serval Project [4] aims to bring infrastructure-free mobile communication to people in need, such as during crisis and disaster situation when vulnerable infrastructure like phone cell tower and mains electricity are cut off. This is also one of our aims. We want to our framework to be more generic and reusable, while the Serval Project is only dedicated to this special purpose.

Commotion Wireless [5] is an open-source communication tool that uses mobile phones, computers, and other wireless devices to create decentralised mesh networks. We share one common aim that is decentralisation. However we are focusing more on a framework to allow people to develop applications rather than just communications.

Open Garden [6] is another free piece of software which creates an overlay mesh network using Bluetooth and Wi-Fi connections across a range of mobile devices, from smartphones to tablets to laptops and desktops. The aim of Open Garden is to provide connection to internet without an Wi-Fi access point, while our aim is more general than just accessing the Internet.

Project SPAN [7] is an Android API for ad-hoc networks. While actively developed, it does not provide any mechanisms for automatic IP configuration, service discovery or name resolution.

The findings from our field study show that the existing work on proximity-based wireless networks is either application-specific, i.e. based around a specific use case; or it is not quite functionally adequate to meet our requirements for a useful framework for proximity-based computing.

III. Proxima–A general framework for proximity-based communication

In this section we describe the Proxima framework from an application developers viewpoint, in terms of how to integrate proximity-based functionality into an Android application. We then describe some of the low-level detail and design decisions that were made during development.

A. Developer Viewpoint

The Proxima framework provides a fully asynchronous, thread-safe API and can be used by multiple client applications on a single device. It acts like a ”neighbours and resources finder” server. It is not necessary for API users to worry about how to discover neighbors and what services are available; the framework will take care of these low-level details.. Communication with the framework is done using asynchronous method calls with user-supplied callback functions. Users are sent broadcast intents when changes occur, such as a new device coming into proximity.

Registering a client application and retrieving a list of neighbors is demonstrated in Figure 1. The client-facing API takes inspiration from the Android Wi-Fi P2P API. This is a simplified version; the calls to initialize() and discoverNeighbors() both take nullable callback arguments to notify the user of success or failure. These calls are usually placed into the onCreate() method of an Android activity.

The framework also sends periodic broadcasts to all registered client applications when there is a neighbor change, so clients should implement a broadcast receiver to listen for these specific broadcasts. Neighbors can be retrieved at any point once the discovery process has begun.

Listing 1. Example usage of the Proxima API.

```
ProximityManager mProximityManager
    = ProximityManager.getInstance();

// This must be the first call into the API
Channel mChannel = mProximityManager
    .initialize(this);

// Begin the neighbor discovery procedure
mProximityManager.discoverNeighbors(mChannel);

// Retrieve the list of neighbors
mProximityManager.retrieveNeighbors(mChannel,
    new NeighborListListener()
{
    @Override
    public void onNeighborsAvailable
        (NeighborList neighbors)
    {
        // ...
    }
});
```

Each neighbour in the retrieved list represents a device detected by the routing daemon. Prior to passing this list to the user via the callback, the framework queries the a service on each neighbour and retrieves metadata such as the user-specified device name, the device GPS coordinates and the available services (Proxima applications) provided by the device. The metadata service is described below.

In summary, as a mobile application developer, to use the Proxima framework, all you need to do is to register the application with the framework as a client at the beginning. Then, the framework will send the devices found in the neighbourhood to you, and any changes in the network will be updated via self-defined callback function. It is very simple and easy to use.

B. Implementation Details

As mentioned in the background section, ad-hoc Wi-Fi is used for the underlying connectivity. To enable ad-hoc Wi-Fi mode on Android, it is necessary to have root access to the device. This is because the network interface must be switched from infrastructure to ad-hoc mode. Other parameters, such as IP address and channel must also be configured in this manner. Native binaries required for this were compiled for the ARM processor and packaged with the framework. Core binaries include ifconfig, iwconfig, olsrd and dnsmasq. Utility/testing binaries include busybox and tcpdump.

1) Routing: For the routing mechanism, we chose to use the Optimized Link State Routing (OLSR) protocol [8]. It is widely tested, scalable, reliable, and is open source. It is a proactive routing protocol, as opposed to reactive or hybrid routing protocols [9]. It allows multi-hop routing, and is fully decentralised and self-healing. It also allows routing of traffic onto the Internet via gateway nodes if desirable; this is not, however, one of our key requirements. The olsrd implementation of OLSR has been used [10] , following recompilation of the code for Android.

The olsrd implementation features a plugin system. The jsoninfo plugin [10] is used to interact with the olsrd daemon. It listens on the localhost interface on port 9090 and takes a URL of the form: `http://127.0.0.1:9090/command` where `command` is the command to be given to the plugin. Many commands are available. To get a list of nearby devices, the links command is used, i.e. `http://127.0.0.1:9090/links`. Multiple commands can be supplied at once, e.g. `http://127.0.0.1:9090/links/routes`. The Jackson JSON library is used to parse the output from the olsrd daemon. A configuration file packaged with the framework specifies the plugins to be used and wireless interfaces to be used.

2) Name resolution: DNS-like name resolution was initially handled using a combination of the olsrd nameservice plugin and the dnsmasq program. The olsrd plugin modified the /etc/hosts and /etc/resolv.conf files and sent a SIGHUP signal to the dnsmasq process when there was a change in local devices. Each node was then effectively a DNS server; it would resolve its own hostname when asked, and kept a cache of hostnames for other nodes. Other such solutions exist, such as described in [11], [12], but the functionality was already in the OLSR implementation and is open-source and extendable. However, we found this solution to be too brittle. For example, if a node changed IP address, then it would take a while for each node to update its DNS cache, causing inconsistencies. Additionally, this solution only allows names composed of the small subset of characters allowed by the DNS protocol, which does not include Unicode characters or even whitespace characters. Following these discoveries, name resolution has been absorbed into the metadata service, which is explained below.

3) Service discovery: We have decided to incorporate UPnP-like service discovery into the metadata service. Service discovery could potentially be implemented using an extension to the olsrd nameservice plugin, and having additional framework API methods for registering/searching/unregistering services. However, this puts additional reliance upon the particular routing protocol, making any future changes more difficult. Other methods for service discovery in ad-hoc Wi-Fi networks are described in [13], [14], [15].

4) Geolocation: Previous approaches to GPS coordinate sharing in ad-hoc networks have tried to modify the OLSR protocol to pass GPS coordinates, such as in [16]. However, we feel that this approach is disruptive to the protocol and hence becomes less cross-compatible. Geolocation information is distributed on request via the metadata service.

5) IP autoconfiguration: DHCP-like IP autoconfiguration is currently done in a very rudimentary way. The subnet ID is simply randomly generated (i.e. 192.168.1.x where x is randomly generated from 1-254). Will need to have a collision resolution strategy. Several such strategies currently exist, such as [17]. These solutions will be explored in future iterations.

In the next subsection, we propose a solution for the name resolution, geolocation and service discovery problems. Our approach is conceptually simple; we embed a HTTP server on each device which acts as a metadata server.

C. The metadata service

The metadata service functions as a HTTP server and is responsible for serving information about its parent node, such as GPS coordinates, and human-readable node identifiers. The metadata service also acts as a service discovery layer. Client applications can register their services, and can even specify custom per-application metadata to be served.

See Figure 2 for a high-level overview of the architecture of the Proxima framework, shown as a UML deployment

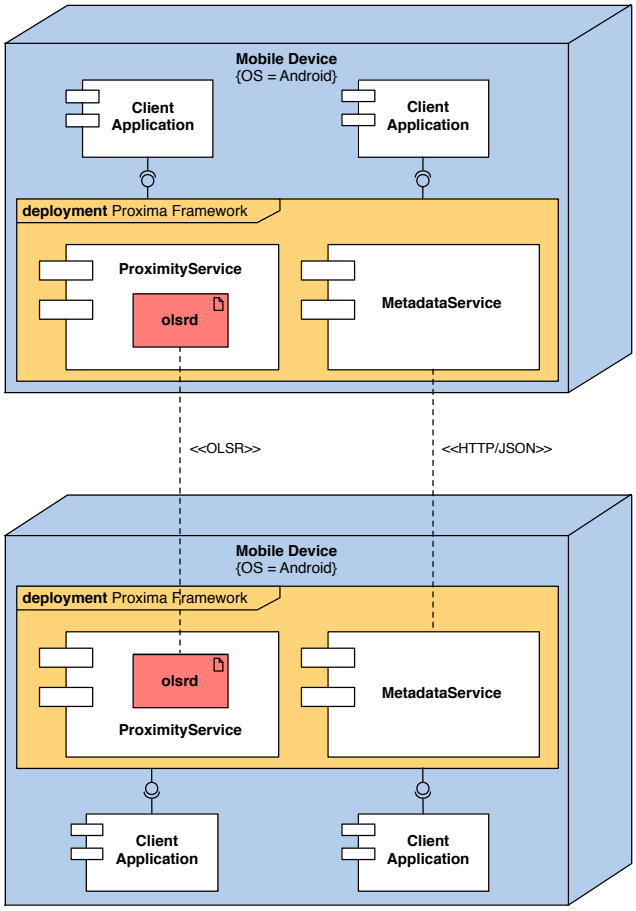


Figure 2. Proxima architectural overview.

diagram. Figure 2 shows the Proxima framework being used by multiple clients on multiple devices. It shows the communication between two metadata services, and the underlying OLSR daemon communication mechanism.

The Proxima API was written using the Android Java SDK. It is compatible with Android 1.6 (codename Donut, API level 4). It is deployed as an Android Library Project. It requires a rooted device. Development was done using the Eclipse IDE (Kepler) with the Android ADT plugin on a machine running Ubuntu 10.04 (Lucid Lynx) x86_64 with kernel version 2.6.32. The Java Process.exec() class is used to interact with the native system, to modify the network interface into ad-hoc mode, set the IP address and netmask, and to start/stop the olsrd and dnsmasq daemons. It is also used to make calls to the olsrd jsoninfo plugin.

IV. Results

We have successfully implemented a real-world application which uses all the functionality of the framework. The application demonstrates the ease of use of the Proxima

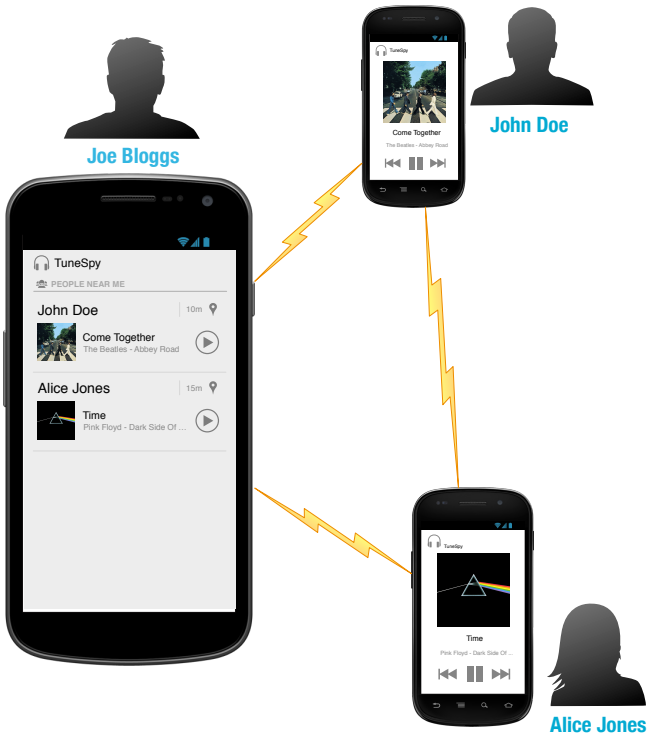


Figure 3. TuneSpy operational overview.

framework and serves as a platform for functional testing. Development was done on the following devices:

- Samsung Galaxy S4, running Android version 4.3 (Jelly Bean)
- Samsung Galaxy Tab 10.1 running Android version 3.2 (Honeycomb)
- HTC Wildfire running Android version 2.2.3 (Froyo)

A. *TuneSpy*

Our sample application, named TuneSpy, allows users in proximity to one another to stream their current music track to each other. The application uses the Proxima framework to broadcast metadata about itself to surrounding neighbours. Upon request from a neighbour, the application will directly stream the actual track data. An illustration of this is given in Figure 3. The application was inspired by the initial scenario given above, and is designed to operate on a purely local, ephemeral, peer-to-peer basis.

V. Conclusion and Future Work

A. *Conclusion*

We have created a useful abstraction tool for mobile peer-to-peer proximity-based networking, following our initial vision of the proximity-based paradigm. We believe that our work is a significant contribution to the collaborative mechanism and architecture. This belief is held for the following reasons.

Firstly, a framework such as this, with its unique semantics of purely-local information dissemination, could potentially allow application developers to incorporate this proximity-based functionality into their existing applications, or develop entirely new classes of unique mobile applications. Based on our current research, we have not found any other system with matching generality or functionality of the Proxima framework.

Secondly, Proxima provides a nearly zero-configuration interface. The only aspect that needs to be configured is the device/user name (similar to that of the Bluetooth device name) which can be achieved programmatically with a simple API call. This ease of configuration makes the framework both user- and developer-friendly.

Thirdly, our separation of underlying connectivity mechanism, routing functionality and higher-level services gives us flexibility to potentially change any one of these elements in the future. Since we are currently using the OLSR protocol as a routing backend, applications developed using the framework benefit from all the scalability and resilience of the protocol. The framework takes care of the transport mechanics, device specifics and configuration issues, leaving the application developer to focus solely on implementing their application.

Finally, the framework is very lightweight at only 6MB (including all necessary native binaries and compiled code), representing a small overhead for addition into applications.

B. Future work

We hope in the future to extend the framework, improving the IP autoconfiguration strategy and implementing improved security mechanisms, amongst other things.

One significant limitation is the lack of built-in support for ad-hoc mode on Android and hence the necessity for extensive modifications to gain root access to the device. It is not likely for this support to be added in the future. Thus we hope to explore alternative ad-hoc connectivity mechanisms that do not require extensive device modification but still endow us with the proximity-based semantics that we desire.

We will continue to test and support the framework on a wider variety of Android devices and larger networks. We also hope to implement a version of the framework for Apple iOS, and another version for desktop operating systems.

REFERENCES

[1] Foursquare, "Foursquare web page [online]," Available: https://foursquare.com/ [Accessed 17 February 2014].

[2] B. Bostanipour, B. Garbinato, and A. Holzer, "Spotcast – a communication abstraction for proximity-based mobile applications," in *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, Aug 2012, pp. 121–129.

[3] WiFi Alliance, "Wi-fi peer-to-peer (p2p) technical 7 specification [online]," Available: http://www.wi-fi.org/discover-wi-fi/wi-fi-direct [Accessed 17 February 2014].

[4] P. Gardner-Stephen and S. Palaniswamy, "Serval mesh software-wifi multi model management," in *Proceedings of the 1st International Conference on Wireless Technologies for Humanitarian Relief*, ser. ACWR '11. New York, NY, USA: ACM, 2011, pp. 71–77. [Online]. Available: http://doi.acm.org/10.1145/2185216.2185245

[5] A. Reynolds, J. King, S. Meinrath, and T. Gideon, "The commotion wireless project," in *Proceedings of the 6th ACM Workshop on Challenged Networks*. ACM, 2011, pp. 1–2.

[6] Open Garden, "Open garden web page [online]," Available: http://opengarden.com/ [Accessed 17 February 2014].

[7] J. Thomas, J. Robble, and N. Modly, "Off grid communications with android meshing the mobile world," in *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, 2012, pp. 401–405.

[8] T. Clausen, P. Jacquet, C. Adjih, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L. Viennot, "Optimized Link State Routing Protocol (OLSR)," 2003, network Working Group Network Working Group. [Online]. Available: http://hal.inria.fr/inria-00471712

[9] M. Campista, P. Esposito, I. Moraes, L. Costa, O. Duarte, D. Passos, C. de Albuquerque, D. Saade, and M. Rubinstein, "Routing metrics and protocols for wireless mesh networks," *Network, IEEE*, vol. 22, no. 1, pp. 6–12, Jan 2008.

[10] A. Tonnesen, T. Lopatic, H. Gredler, B. Petrovitsch, A. Kaplan, and S. Turke, "Olsrd: An adhoc wireless mesh routing daemon [online]," Available: http://olsrd.org [Accessed 14 February 2014], 2008.

[11] S. G. Hong, S. Srinivasan, and H. Schulzrinne, "Accelerating service discovery in ad-hoc zero configuration networking," in *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, Nov 2007, pp. 961–965.

[12] X. Hong, J. Liu, R. Smith, and Y.-Z. Lee, "Distributed naming system for mobile ad-hoc networks," *contract*, vol. 14, no. 01-C, p. 0016, 2005.

[13] X. Shao, L. H. Ngoh, T. K. Lee, T. Chai, L. Zhou, and J. Teo, "Multipath cross-layer service discovery (mcsd) for mobile ad hoc networks," in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, 2009, pp. 408–413.

[14] N. Le Sommer and Y. Mahéo, "OLFServ: an Opportunistic and Location-Aware Forwarding Protocol for Service Delivery in Disconnected MANETs," in *Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm 2011)*, X. P. Services, Ed., Lisbon, Portugal, Nov. 2011, pp. 115–122. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00663455

[15] U. Aguilera and D. López-de Ipiña, "A parameter-based service discovery protocol for mobile ad-hoc networks," in *Ad-hoc, Mobile, and Wireless Networks*. Springer, 2012, pp. 274–287.

[16] W. Anbao and Z. Bin, "Realize 1-hop node localization based on olsr protocol in ad hoc networks," in *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*, Dec 2012, pp. 1475–1478.

[17] C. Bernardos, M. Calderón, and H. Moustafa, "Survey of ip address autoconfiguration mechanisms for manets," *IETF Internet Draft*, October 2007.

Glossary

3G Third Generation is a communications technology allowing high-speed data access and voice communication over cellular networks.

802.11b The 802.11b standard is a wireless networking specification governed by the IEEE and otherwise known as Wi-Fi.

Access point An access point is a special wireless device on a network that receives and transmits Wi-Fi signals to support Internet accessibility for other devices.

Ad-hoc network A wireless network comprising direct device-to-device connections in an impromptu fashion without requiring traditional access points.

Android An operating system based on the Linux kernel specially designed for embedded touchscreen mobile devices..

AODV Ad-Hoc On-Demand Distance Vector Routing is a reactive routing protocol designed for ad-hoc wireless networks such as MANETs.

API An Application Programming Interface is a set of methods made available by a software system specifying how other software components will interact with it.

APK file An Application Package file is the file format used on Android systems to install applications.

Asymmetric neighbour A link between two OLSR interfaces that has been verified in one direction only.

Asynchronous method call A design pattern used in event-based systems to instigate long-running operations without blocking the calling thread.

Bluetooth A wireless technology for exchanging data between devices over a short distance.

Bonjour A zero-configuration service discovery protocol designed by Apple Inc.

Bound service An Android background service that allows back-and-forth client communication.

BSS The Basic Service Set is the basic operational configuration of an infrastructure Wi-Fi network.

Cellular network A wireless network comprising multiple fixed cells distributed over a large geographical area.

Control message A message sent over an OLSR-based network used to notify and update neighbours of network topology changes.

CRC card A technique for brainstorming design in an object-oriented system.

Daemon Computer programs that run in the background without user interaction are known as daemons.

Dalvik VM The Android equivalent of the Java Virtual Machine.

DHCP The Dynamic Host Control Protocol is a networking protocol used to assign IP addresses to devices on a network.

DNS The Domain Name System is a naming system for networked devices based on a hierarchical distributed global architecture.

EDGE Enhanced Data Rates for GSM Evolution is a second-generation mobile cellular communication technology.

ESS The Extended Service Set is a wireless network comprising two or more Basic Service Sets (BSSs).

Flooding A technique of transmitting a message to all nodes of a wireless ad-hoc network.

Framework A set of reusable tools and APIs providing some functionality in order to facilitate development of a solution to a problem.

GPRS The General Packet Radio Service is a packet-switched cellular networking technology.

GPS The Global Positioning System is a satellite-based navigation system providing location triangulation from anywhere on Earth.

IBSS The Independent Basic Service Set is a network of devices connected via 802.11b ad-hoc Wi-Fi.

IEEE The Institute of Electrical and Electronics Engineers is a non-profit professional organisation dedicated to furthering technological excellence and innovation.

IETF The Internet Engineering Task Force creates and maintains standards for Internet-based technologies.

IP address A numerical identifier assigned to all networked devices communicating via the Internet Protocol standard.

IP address autoconfiguration A mechanism for assigning IP addresses to devices on a network automatically.

IWMN Infrastructure Wireless Mesh Networks are mesh networks comprising fixed infrastructure nodes.

JRE The Java Runtime Environment is the combination of the Java Virtual Machine and the Java SDK.

JVM The Java Virtual Machine is a virtual device that comprises the code execution component of the JRE.

Kernel The kernel is a fundamental component of an operating system that is responsible for memory and process management and other core tasks..

Library project A repository of source code that is reusable between Android applications.

Link A pair of interfaces running the OLSR protocol and capable of communication.

LRU cache A caching algorithm which maintains a fixed-size queue of items and periodically evicts the least used item.

LTE Long Term Evolution is a modern, high-speed cellular communication standard.

MANET Mobile Ad-Hoc Networks are infrastructure-free peer-to-peer wireless networks comprising mobile devices capable of routing traffic.

Mesh network A network in which all nodes are considered equal and cooperate in the routing of traffic throughout the network.

Mock data Simulated objects used for testing purposes to mimic real objects.

MPR Multi-Point Relaying is a special optimisation within OLSR networks designed to reduce the number of transmissions required to disseminate a message across an entire network.

Multi-hop routing The ability for a node in a mesh network to communicate with another node via intermediary forwarding nodes.

Name resolution The process of converting a numerical identifier into a human-readable alphanumeric identifier.

Native library A code library written for a specific platform in a language that compiles into natively executable code.

NDK The Native Development Kit is a set of tools for Android that allows writing code in native languages such as C++.

NFC Near-Field Communication is a short-range, low power proximity-based communication standard for mobile devices.

Node A single device or entity in a computer network.

OLSR Optimized Link State Routing is a proactive table-driven routing protocol for wireless mesh networking.

One-hop neighbour A node in an OLSR network reachable from another node via a single hop.

OOAD Object-Oriented Analysis and Design is an approach to designing a software system under the object-oriented paradigm.

PBF Proximity-Based Functionality refers to the operation of components within the Proxima framework that allows devices to communicate within proximity of one another.

Peer-to-peer network A decentralised network architecture in contrast to the client-server model.

Proxima A general framework for proximity-based computing over ad-hoc mesh networks.

ROM A custom version of the Android operating system build by community enthusiasts.

Rooting The process of obtaining superuser privileges on an Android device.

Routing protocol A software system operating on interconnected nodes to cooperatively distribute information via route determination.

Sandboxing A security safeguard mechanism used to isolate untrusted applications from the host system.

SDK Software Development Kits are development tools that allow applications to be written for a particular software platform.

Service discovery The process of detecting services offered by devices on a network via a common language.

Started service An Android background service that runs indefinitely without client interaction.

Symmetric neighbour A link between two OLSR interfaces that has been verified in both directions.

Thread safety Software that is thread-safe can be executed concurrently by multiple threads without adverse effects.

TuneSpy An Android application that allows users in proximity to share music without an Internet connection.

Two-hop neighbour A node in an OLSR network reachable from another node via a single intermediate node.

UDP The User Datagram Protocol is a simple messaging protocol for Internet devices.

UPnP Universal Plug and Play is a service discovery protocol for personal computing devices in residential networks.

WAN Wide Area Networks are networks that span large areas, typically of metropolitan or regional size.

Wi-Fi A well-known trademark for wireless networking.

Wireless driver A piece of software that controls the wireless networking hardware on a computing device.

Zero-confiuration networking Techniques based on several collaborating technologies to allow usable network creation with no user intervention.

Bibliography

Aguilera, U. and López-de Ipiña, D. (2012) A Parameter-Based Service Discovery Protocol for Mobile Ad-Hoc Networks. In *Ad-hoc, Mobile, and Wireless Networks*, vol. 7363 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-642-31637-1, pp. 274–287, doi:10.1007/978-3-642-31638-8_21.

Beck, K. and Cunningham, W. (1989) A Laboratory for Teaching Object Oriented Thinking. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, New York, NY, USA: ACM, ISBN 0-89791-333-7, pp. 1–6, doi:10.1145/74877.74879, URL `http://doi.acm.org/10.1145/74877.74879`.

Bernardos, C., Calderón, M., and Moustafa, H. (2008) Survey of IP address autoconfiguration mechanisms for MANETs [Online]. Available: `https://tools.ietf.org/html/draft-bernardos-manet-autoconf-survey-04` [Accessed 15 March 2014].

Bluetooth SIG (2014) Bluetooth Special Interest Group [Online]. Available: `http://www.bluetooth.com/Pages/about-bluetooth-sig.aspx` [Accessed 15 March 2014].

Bostanipour, B., Garbinato, B., and Holzer, A. (2012) Spotcast – A Communication Abstraction for Proximity-Based Mobile Applications. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pp. 121–129, doi:10.1109/NCA.2012.34.

Burnette, E. (2009) *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2nd ed., ISBN 1934356492, 9781934356494.

Campista, M., Esposito, P., Moraes, I., Costa, L., Duarte, O., Passos, D., de Albuquerque, C., Saade, D., and Rubinstein, M., Routing Metrics and Protocols for

Wireless Mesh Networks. *Network, IEEE*, 22 (2008) (1), pp. 6–12, ISSN 0890-8044, doi:10.1109/MNET.2008.4435897.

Clausen, T., Jacquet, P., Adjih, C., Laouiti, A., Minet, P., Muhlethaler, P., Qayyum, A., and Viennot, L. (2003) Optimized Link State Routing Protocol (OLSR) [Online]. Available: `http://hal.inria.fr/inria-00471712` [Accessed 15 March 2014].

Codehaus (2014) Jackson: A high-performance JSON processor [Online]. Available: `http://jackson.codehaus.org/` [Accessed 15 March 2014].

Dawson, C. W. (2000) *The essence of computing projects: a student's guide*. Prentice Hall, ISBN 013021972X.

DSDM (2014) MoSCoW Prioritisation [Online]. Available: `http://dsdm.org/content/10-moscow-prioritisation` [Accessed 21 March 2014].

Foursquare (2014) Foursquare web page [Online]. Available: `https://foursquare.com/` [Accessed 17 February 2014].

Gardner-Stephen, P. and Palaniswamy, S. (2011) Serval Mesh software-WiFi Multi Model Management. In *Proceedings of the 1st International Conference on Wireless Technologies for Humanitarian Relief*, ACWR '11, New York, NY, USA: ACM, ISBN 978-1-4503-1011-6, pp. 71–77, doi:10.1145/2185216.2185245.

Google, Inc. (2014) Testing Fundamentals [Online]. Available: `https://developer.android.com/tools/testing/testing-android.html` [Accessed 15 March 2014].

Haas, Z. (1997) A new routing protocol for the reconfigurable wireless networks. In *Universal Personal Communications Record, 1997. Conference Record., 1997 IEEE 6th International Conference on*, vol. 2, ISSN 1091-8442, pp. 562–566 vol.2, doi:10.1109/ICUPC.1997.627227.

Hawke, P. (2012) NanoHttpd: A tiny, easily embeddable HTTP server in Java. Available: `https://github.com/NanoHttpd/nanohttpd` [Accessed 25 March 2014].

Hong, S. G., Srinivasan, S., and Schulzrinne, H. (2007) Accelerating Service Discovery in Ad-Hoc Zero Configuration Networking. In *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pp. 961–965, doi:10.1109/GLOCOM.2007.185.

Hong, X., Liu, J., Smith, R., and Lee, Y.-Z. (2005) Distributed naming system for mobile ad-hoc networks. In *contract*, vol. 14, p. 0016.

IETF (2014) IETF Working Group for Mobile Ad-hoc Networks [Online]. Available: `http://datatracker.ietf.org/wg/manet/charter/` [Accessed 15 March 2014].

JUnit (2004) JUnit [Online]. Available: `https://github.com/junit-team/junit` [Accessed 15 March 2014].

Khorov, E., Kiryanov, A., Lyakhov, A., and Ostrovsky, D. (2012) Analytical study of neighborhood discovery and link management in OLSR. In *Wireless Days (WD), 2012 IFIP*, IEEE, pp. 1–6, doi:10.1134/S1064226912120030.

Kim, D., Chun, H., and Lee, H. (2014) Determining the factors that influence college students' adoption of smartphones. In *Journal of the Association for Information Science and Technology*, pp. 2330–1643, doi:10.1002/asi.22987.

Le Sommer, N. and Mahéo, Y. (2011) OLFServ: an Opportunistic and Location-Aware Forwarding Protocol for Service Delivery in Disconnected MANETs. In X. P. Services (Editor), *Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm 2011)*, Lisbon, Portugal, pp. 115–122.

Open Garden (2014) Open Garden web page [Online]. Available: `http://opengarden.com/` [Accessed 17 February 2014].

Perkins, C., Belding-Royer, E., Das, S., *et al.*, RFC 3561: Ad-Hoc On-Demand Distance Vector Routing (AODV). *Internet RFCs*, (2003), pp. 1–38.

Qualcomm (2014) About Alljoyn [Online]. Available: `https://www.alljoyn.org/about` [Accessed 25 March 2014].

Reynolds, A., King, J., Meinrath, S., and Gideon, T. (2011) The commotion wireless project. In *Proceedings of the 6th ACM Workshop on Challenged Networks*, ACM, pp. 1–2, doi:10.1145/2030652.2030654.

Rumbaugh, J. (2003) Object-oriented Analysis and Design (OOAD). In *Encyclopedia of Computer Science*, Chichester, UK: John Wiley and Sons Ltd., ISBN 0-470-86412-5, pp. 1275–1279, URL `http://dl.acm.org/citation.cfm?id=1074100.1074650`.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004) *The Unified Modeling Language Reference Manual*. Pearson Higher Education.

Saville, W. (2008) Issue 82: Support Wi-Fi ad hoc networking [Online]. Available: `https://code.google.com/p/android/issues/detail?id=82` [Accessed 22 February 2014].

Shao, X., Ngoh, L. H., Lee, T. K., Chai, T., Zhou, L., and Teo, J. (2009) Multi-path cross-layer service discovery (MCSD) for mobile ad hoc networks. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pp. 408–413, doi: 10.1109/APSCC.2009.5394093.

Thomas, J., Robble, J., and Modly, N. (2012) Off Grid communications with Android Meshing the mobile world. In *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, pp. 401–405, doi:10.1109/THS.2012.6459882.

Tonnesen, A., Lopatic, T., Gredler, H., Petrovitsch, B., Kaplan, A., and Turke, S. (2008) Olsrd: An adhoc wireless mesh routing daemon [Online]. Available: http://olsrd.org [Accessed 14 February 2014].

Wang, A. and Zhu, B. (2012) Realize 1-hop node localization based on OLSR protocol in Ad Hoc networks. In *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*, pp. 1475–1478, doi:10.1109/ICCSNT.2012.6526199.

WiFi Alliance (2014) Wi-Fi Peer-to-Peer (P2P) Technical 7 Specification [Online]. Available: `http://www.wi-fi.org/discover-wi-fi/wi-fi-direct` [Accessed 17 February 2014].

Wilhelm, A. (2013) Android Market Share 2013 [Online]. Available: `http://techcrunch.com/2013/11/12/windows-phone-android-gain-market-share-while-apple-slips-despite-growth-in-iphone-shipments/` [Accessed 15 March 2014].