Android Ad-hoc SMS Network

Final Report

By: Danny Rasin
Omri Dor

Supervisor: Aran Bergman
We would like to thank Aran Bergman for his active involvement, expert advice and oversight of our project.

We would also like to thank Victor Kulik for letting us use the various Android phones and tablets in the NSSL lab.

Finally, we would like to thank all our beta testers, who contributed their time and their Android phones to allow us to test our project more thoroughly: Orit Moskovich, Nir Even-Chen, Barak Farbman, Lior Noyovitch, Maya Fainshtein and Noam Kritenberg.
# Table of Contents

Abstract ................................................................................................................................. 9  
Glossary ................................................................................................................................. 10  
Overview............................................................................................................................... 11  

1. Introduction to the Mobile Ad-hoc Network Problem ...................................................... 12  
   1.1. Section Overview ..................................................................................................... 12  
   1.2. Problem Statement and Discussion ....................................................................... 12  
   1.3. Overview of Relevant Routing Concepts ................................................................ 12  
   1.4. Existing Ad-hoc Routing Solutions ........................................................................ 13  
      1.4.1. Data Flooding ................................................................................................. 13  
      1.4.2. Routing Protocols Classifications .................................................................. 13  
      1.4.3. Dynamic Source Routing (DSR) ................................................................... 13  
      1.4.4. Optimized Link State Routing (OLSR)........................................................... 14  
   1.5. Chosen Routing Solution .......................................................................................... 15  
   1.6. Requirements .......................................................................................................... 16  
      1.6.1. General Features ............................................................................................ 16  
      1.6.2. Functional Requirements ............................................................................... 16  
      1.6.3. Non-Functional Requirements ....................................................................... 17  

2. Proposed Solution ............................................................................................................. 19  
   2.1. The Datagram ........................................................................................................... 19  
   2.2. Architecture ............................................................................................................. 20  
   2.3. Data Flooding Algorithm ....................................................................................... 22  
      2.3.1. Basic Algorithm .............................................................................................. 22  
      2.3.2. Mobility Tolerance ......................................................................................... 23  

3. Android / Bluetooth Implementation Overview ............................................................ 24  
   3.1. The Android Environment ..................................................................................... 24  
      3.1.1. Activity ........................................................................................................... 24  
      3.1.2. Content Provider ............................................................................................. 24  
      3.1.3. Service ............................................................................................................ 25  
      3.1.3.1. Intent Service .............................................................................................. 25  
      3.1.4. Broadcast Receiver ......................................................................................... 25  
      3.1.5. Intent .............................................................................................................. 25  

Page 3 of 113
3.2. Object Overview ........................................................................................................... 25
3.3. Datagram(Class) ........................................................................................................... 27
3.4. Layer2Interface(Abstract Class) ............................................................................... 28
3.5. Layer3Interface (Abstract Class) ............................................................................. 28
3.6. Message Sequence Charts .......................................................................................... 29
   3.6.1. Application Usage Flow ....................................................................................... 29
   3.6.2. Discovery Flow ...................................................................................................... 30
   3.6.3. Compose Flow ...................................................................................................... 31
   3.6.4. Forward Flow ........................................................................................................ 32
   3.6.5. Receive Flow ........................................................................................................ 33
4. Layer 2 Implementation .................................................................................................... 34
   4.1. Bluetooth Implementation – Section Overview ......................................................... 34
   4.2. Bluetooth Implementation – General ........................................................................ 34
   4.3. DeviceManager (Class) .............................................................................................. 36
   4.4. BluetoothDiscoveryService (Class) ........................................................................ 37
   4.5. BlacklistService (Class) ............................................................................................ 38
   4.6. BluetoothWrapper (Class) ....................................................................................... 38
   4.7. Accept Flow ............................................................................................................... 39
   4.8. Discovery Flow .......................................................................................................... 40
   4.9. Connection Flow ........................................................................................................ 41
   4.10. Unicast Flow ............................................................................................................ 43
5. Layer 3 Implementation ................................................................................................... 44
   5.1. DbWrapper (Class) .................................................................................................... 45
   5.2. DataFlooder (Class) ................................................................................................... 45
      5.2.1. Broadcast Support ............................................................................................. 46
      5.2.2. Reliability ............................................................................................................ 46
      5.2.3. User Generated Datagram Handling ................................................................. 46
      5.2.4. Incoming Datagram Handling ............................................................................ 47
      5.2.5. Router Hibernation ............................................................................................ 48
   5.3. DataFlooderRoutingService (Class) ...................................................................... 49
      5.3.1. Practical Considerations ...................................................................................... 49
5.3.2. Optimizing Amount of Pending Datagrams .......................................................... 50
5.3.3. Timing and Concurrency .................................................................................... 50
5.3.4. Discovery Conservation .................................................................................... 51
5.3.5. Most Urgent Datagram vs. Focusing on a Single Neighbor .............................. 52
5.3.6. Unimplemented Optimizations ......................................................................... 52
5.3.7. Full Diagram of Routing Protocol .................................................................... 54

6. Application Layer .................................................................................................. 57
6.1. MessageArchive (Abstract Class) ......................................................................... 57
6.2. PhoneUtilities (Static Class) ............................................................................... 57
6.3. HelloService (Class) ......................................................................................... 57
6.4. AdhocApplication (Class) .................................................................................. 58
6.5. Compose(Activity Class) ..................................................................................... 58
6.6. Menu (Class) ...................................................................................................... 59
6.7. Preferences ......................................................................................................... 59
6.8. Inbox (Activity Class) ......................................................................................... 61
6.9. Outbox (Activity Class) ...................................................................................... 61
6.10. Debug (Activity Class) ....................................................................................... 62
6.11. Performance (Activity Class) ............................................................................ 63
6.12. Splash (Activity Class) ...................................................................................... 64

7. Development Process ............................................................................................. 65
7.1. Development Methodologies ............................................................................... 66
7.2. Design Patterns .................................................................................................. 67
7.2.1. Observer .......................................................................................................... 67
7.2.2. Concurrency Design Patterns .......................................................................... 67
7.2.3. Singleton .......................................................................................................... 68
7.3. Layer 2 Milestones ............................................................................................. 68
7.3.1. Problems Encountered With 2.3.3 Devices .................................................... 68
7.3.2. Data Bundling Experiment .............................................................................. 69
7.4. Layer 3 Milestones ............................................................................................. 70

8. Testing ...................................................................................................................... 70
8.1. Layer2Stub (Class) ............................................................................................. 71
Appendix A

Bibliography

9. Future Work ................................................................. 101
  9.1. Performance Optimizations ........................................ 101
  9.2. Complex Payload Type ............................................. 101
  9.3. Multicasting ........................................................... 102
  9.4. Connectivity to External Networks ................................ 102
  9.5. Authentication and Security ........................................ 103
  9.6. Alternative Layer 2 Implementations ............................ 103
  9.7. Optimization of Routing Protocol Parameters ................... 103

Bibliography ........................................................................ 103

Appendix A – User Manual .................................................. 104

Installation .......................................................................... 104

Main Screen (Compose) ...................................................... 104

Inbox ................................................................................. 104

Outbox ................................................................................ 106

Notifications ....................................................................... 107

Menu .................................................................................. 108
Table of Figures

Figure 1 - datagram design ................................................................. 19
Figure 2 - Alternative implementations in the layered architecture .................. 21
Figure 3 - The layered architecture of our solution .................................................... 26
Figure 4 - Application usage flow .............................................................. 30
Figure 5 - Discovery flow ........................................................................... 31
Figure 6 - Compose flow ............................................................................ 32
Figure 7 - Forward flow .............................................................................. 33
Figure 8 - Receive flow ............................................................................... 34
Figure 9 – Layer 2 design overview .............................................................. 35
Figure 10 - Bluetooth implementation, accept flow ........................................... 39
Figure 11 - Bluetooth implementation, discovery flow ........................................ 41
Figure 12 - Bluetooth implementation, connection flow ..................................... 42
Figure 13 - Bluetooth implementation, unicast flow ........................................... 44
Figure 14 - Flowchart of Data Flooding Routing Protocol ................................. 54
Figure 15 - Compose activity screen .............................................................. 59
Figure 16 - Preferences activity screen .......................................................... 61
Figure 17 - Inbox activity screen ................................................................. 61
Figure 18 - Outbox activity screen ............................................................... 62
Figure 19 - Debug activity screen ................................................................. 63
Figure 20 - Performance activity screen ......................................................... 64
Figure 21 - Splash activity screen .................................................................. 65
Figure 22 (a, b and c) - Toasts and Notification screens ..................................... 65
Figure 23 - Two Device One Way Latency Distribution .................................... 92
Figure 24 - Two Device One Way Latency Distribution (First Message In Batch) ................................................................. 93
Figure 25 - Two Device Round Trip Latency Distribution ................................ 94
Figure 26 - Two Device Round Trip Latency Distribution (First Message In Batch) ................................................................. 95
Figure 27 - Three Device Relay, Per Hop Latency Distribution .......................... 97
Figure 28 - Many Devices, Round Trip Latency Distribution ............................ 99
Figure 29 - Four Device Relay, Latency Distribution ....................................... 100

Go Visible / Invisible .................................................................................. 110
Turn Router On/Off ................................................................................... 110
Preferences ............................................................................................... 110
Main Preferences ....................................................................................... 111
Advanced Preferences ............................................................................... 112
Delete Options .......................................................................................... 113
List of Tables
Table 1 - Glossary ........................................................................................................................................... 11
Table 2 - Pros and Cons for routing protocols .......................................................................................... 15
Table 3 - Functional requirements ........................................................................................................... 17
Table 4 - Non functional requirements ...................................................................................................... 18
Table 5 - Received datagram decision table ............................................................................................... 48
Abstract
Mobile cellular communication may often be disrupted in large gatherings such as concerts, demonstrations or in disaster areas just as we have recently seen with hurricane Sandy. The plethora of available mobile devices, all equipped with medium range RF technologies such as Bluetooth and Wi-Fi, make such an environment very suitable for mobile ad hoc networking. With this approach, each mobile device doubles as a router, forwarding data for the sake of other devices, allowing for distributed communication that does not require infrastructure.

In this project we implement such a mobile ad hoc network on an Android/Bluetooth platform, offering a basic text messaging service. Our solution is based on ‘data flooding’ routing, meaning every packet of data eventually reaches every node in the network, with each node pulling only relevant packets from the stream.

We also successfully implement the ‘data mule’ concept: communication between disparate connected components is made possible by piggybacking on travelling devices (‘mules’).
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>A datagram of ‘Ack’ type, meant to acknowledge that a certain Message has been received.</td>
</tr>
<tr>
<td>Activity</td>
<td>An Android application building block that roughly corresponds to a UI ‘screen’.</td>
</tr>
<tr>
<td>Ad hoc network</td>
<td>A decentralized network in which nodes are also routers.</td>
</tr>
<tr>
<td>Cold Start</td>
<td>A device may take longer to forward / receive a datagram if it has not yet set up any Bluetooth connections. This is referred to as a cold start.</td>
</tr>
<tr>
<td>Contact list</td>
<td>A list of contacts in the network, which a certain device can determine their existence in the network due to received datagrams from those device. The contact list is displayed in the main screen of the application (see compose Activity in 6.5).</td>
</tr>
<tr>
<td>Contact picker</td>
<td>An android build in activity used to select contact saved in the phone’s database.</td>
</tr>
<tr>
<td>Control Datagram</td>
<td>A datagram of ‘Control’ type, used to send miscellaneous data which does not require acknowledgement.</td>
</tr>
<tr>
<td>Datagram</td>
<td>A basic bundle of data which consists of a payload and of metadata used for routing.</td>
</tr>
<tr>
<td>DB</td>
<td>Data base.</td>
</tr>
<tr>
<td>Eclipse IDE</td>
<td>The integrated development environment we used in developing the project.</td>
</tr>
<tr>
<td>Flooding</td>
<td>The act of broadcasting a datagram to all neighbors, with them doing the same recursively, so that the datagram reaches every node in the network</td>
</tr>
<tr>
<td>Global Address / Source / Destination</td>
<td>Addresses used for routing across the entire ad hoc network. Phone numbers in our implementation.</td>
</tr>
<tr>
<td>Global Timeout</td>
<td>A configurable time constant. If a sent message was not acked after that period, we assume the sending failed.</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface (Same as UI)</td>
</tr>
<tr>
<td>Hello Datagram</td>
<td>A control datagram that is broadcasted across the ad hoc network. By sending it the sender advertises his address and the fact that he is online</td>
</tr>
<tr>
<td>Intent</td>
<td>An Android object used for communication between the different building blocks or even different applications. May or may not contain additional arguments known as Extras.</td>
</tr>
<tr>
<td>Immediate Neighbors</td>
<td>Same as Layer 2 Neighbors</td>
</tr>
<tr>
<td>Layer 1</td>
<td>The different Android APIs responsible for RF communication, such as Bluetooth and Wi-Fi direct.</td>
</tr>
<tr>
<td>Layer 2</td>
<td>An encapsulation of Layer 1 which presents an identical predefined interface to upper layers</td>
</tr>
<tr>
<td>Layer 2 Neighbors</td>
<td>Devices within RF range of each other</td>
</tr>
<tr>
<td>Layer 3</td>
<td>Utilizes the Layer 2 interface to implement routing. Also allows the application layer to send and receive datagrams from across the network</td>
</tr>
<tr>
<td>Layer 3 Neighbors</td>
<td>Devices within the same ad hoc network, capable of communicating with one another</td>
</tr>
</tbody>
</table>
Overview

Section one is an introduction the reader to the Mobile Ad-hoc Network problem. It covers relevant existing solutions, weighs their pros and cons, and defines the requirements for our application. Section two reviews the architecture of our solution, in a way independent from any specific platform or RF technology. A layered architecture is outlined. Our implementation, specifically for Bluetooth and the Android Platform, is explained in the third section. There, we will translate our main architectural building blocks to Android/java code. In the fourth section, we offer a deeper look into the classes involved with the Bluetooth API. The following section is dedicated to the implementation of the routing protocol and the relevant classes. Section six surveys miscellaneous application layer classes that have more to do with the user experience (GUI). The seventh section is an in-depth look into our development methodologies: best practices, design patters and noteworthy design choices are discussed. The tests endured by our solution are specified in the eighth section. We will discuss both functional and non-functional performance testing. The final ninth section discusses future work – possible improvements and features that can be integrated into our application. In addition, we have one appendix which holds a user manual of the application.

While all java classes are described in the relevant sections, full code documentation based on in-code Jdoc notations is available in a separate document [6].
1. Introduction to the Mobile Ad-hoc Network Problem

1.1. Section Overview
In this section we will introduce the problem of the mobile Ad-hoc network, review the relevant theoretical background, and present the desirable requirements from a solution to the presented problem.

- Section 1.2 will present the motivation for this problem.
- Section 1.3 will introduce us to common terminology and routing concepts
- Section 1.4 will review existing solutions to similar problems
- Section 1.5 will present our chosen solution in comparison with the existing solutions reviewed earlier.
- Section 1.6 will describe the exact functional and performance requirements from the solution.

1.2. Problem Statement and Discussion
In disaster areas or large gatherings (e.g. crowded demonstrations), cellular communications can often become unavailable. This may be caused by a physical malfunction of the network or by too many subscribers trying to use their mobile phones roughly at the same time and at the same area. Under such high load, even a simple text message (SMS) cannot be sent, not even between two subscribers no more than a few hundred meters apart.

The density of these mobile nodes also offers an opportunity: the fact that there are so many mobile subscribers in close vicinity can be used to create an ad-hoc network capable of forwarding data from one to another until the destination is reached.

In this project, we have implemented a text messaging service over an ad-hoc network (over Bluetooth) using the Android platform.

1.3. Overview of Relevant Routing Concepts
A mobile ad-hoc network (MANET) is a network of mobile devices, communicating without a common infrastructure, using only the wireless transmitting and receiving abilities of the devices constituting the network. Each device in the network is a separate unit, free to move independently, forming new connections and abandoning old connections as time advances. The configuration of communication paths between hosts which are not directly connected is done automatically by a predefined routing algorithm.

The basic requirement from a Mobile ad-hoc network is that devices with no direct link may communicate, using other devices in the network to transfer their data. This requires that hosts in the network (all of them or some of them) will forward traffic unrelated to their own, and thus act as routers. In order to fulfill this requirement, the design of the network must take into account the possible high mobility factor. This means that data from different devices with a territorial contiguity (defined according to the transmission range of the different hosts) should
still be able to pass, even if the intermediary device constantly change their position and active links in the network.

A common characteristic of ad hoc network is the usage of “data flooding”. In this document, the term Data flooding will refer to the scenario where each node in the network forwards received data to all of its neighbors. The purpose of data flooding in different ad-hoc network may vary. Possible usages are flooding to synchronize the network, flooding to discover contacts (or publish contacts), flood control data that aids the routing algorithm, or it may be utilized to spread the user data itself. For MANET, see [10]

1.4. Existing Ad-hoc Routing Solutions

1.4.1. Data Flooding
The data flooding algorithm's basic concept is that each node floods all messages arriving to it to all of the nodes in its transmission range. The delivery is completed when a certain node identifies itself as the destination of an arriving message (see “Basic protocols” in [2]).

The most naive implementation of a basic data flooding solution may result with various performance issues:

- High bandwidth and power, due to redundant handling of the same messages
- Network congestion (so called ‘broadcast storms’)
- Reliability issues (will the message reach the destination at all?)
- Infinite message life time (when does flooding stop?)

Those problems may be resolved by various optimizations and heuristics. We will discuss these later in detail, as this is the eventual algorithm that we implement.

1.4.2. Routing Protocols Classifications
There are several different types of routing protocols in mobile Ad-Hoc networks. Pro-active protocols constantly maintain lists of destinations and routes to those destinations. In contrast, Reactive protocols only establish a route on demand, when some data needs to be transmitted by flooding the network with control packets. Flow oriented protocols are similar to reactive protocols in the sense of transmitting on demand, but they vary as they use a prediction based scheme for route selecting [7]. Hybrid routing protocols combine features of reactive and proactive protocols.

1.4.3. Dynamic Source Routing (DSR)
The Dynamic Source Routing protocol (DSR) is an example for a reactive MANET protocol. Reactive protocols are protocols where the network is active only when it has data to pass. The default state of the network is Idle, and when the need to pass a datagram rises, the nodes start forming a route to pass the datagram. DSR uses source routing, meaning the source is responsible for specifying the route of the packet. It contains two main stages: route discovery and route maintenance. When a datagram needs to be sent to a host to which the source has no
known path, it initiates the route discovery. Route maintenance is initiated when transmission fails due to a route error. Following that stage, a new route discovery is initiated.

A more detailed description of DSR may be found in [1]:

“The Dynamic Source Routing protocol (DSR) is a simple and efficient routing protocol designed specifically for use in multi-hop wireless ad hoc networks of mobile nodes. DSR allows the network to be completely self-organizing and self-configuring, without the need for any existing network infrastructure or administration. The protocol is composed of the two mechanisms of Route Discovery and Route Maintenance, which work together to allow nodes to discover and maintain source routes to arbitrary destinations in the ad hoc network. The use of source routing allows packet routing to be trivially loop-free, avoids the need for up-to-date routing information in the intermediate nodes through which packets are forwarded, and allows nodes forwarding or overhearing packets to cache the routing information in them for their own future use. All aspects of the protocol operate entirely on-demand, allowing the routing packet overhead of DSR to scale automatically to only that needed to react to changes in the routes currently in use.”

1.4.4. Optimized Link State Routing (OLSR)

OLSR is an example of a proactive routing protocol. A proactive protocol is a protocol where routes within the entire network are constantly maintained, using routing tables. The advantage of a proactive protocol over a reactive one is that the traffic overhead does not increase as the network grows and the amount of routes increases.

A link state protocol is a protocol where every host holds a connection map of the other hosts in the network. Each node is responsible for calculating the optimal path to any possible destination. This information (destinations and optimal paths) will construct each node’s routing table.

The OLSR protocol includes an optimization designed for mobile networks. Each node discovers all of its 2-hop neighbors. Then, the node selects a set of nodes called MPRs (multipoint relays) such that each of the node’s 2-hop neighbors is reachable via an MPR. The 2-hop neighbors are discovered and maintained using flooded Hello messages. After the selection of MPRs, only they are used to forward broadcasted datagrams. This reduces the high overhead of the topology control messages flooding in a link state protocol.

A more detailed description of OLSR may be found in [4]:

“The protocol is an optimization of the classical link state algorithm tailored to the requirements of a mobile wireless LAN. The key concept used in the protocol is that of multipoint relays (MPRs). MPRs are selected nodes which forward broadcast messages during the flooding process. This technique substantially reduces the message overhead as compared to a classical flooding mechanism, where every node retransmits each message when it receives the first copy of the message. In OLSR, link state information is generated only by nodes elected as MPRs. Thus, a second optimization is achieved by minimizing the number of control messages flooded
in the network. As a third optimization, an MPR node may choose to report only links between itself and its MPR selectors. Hence, as contrary to the classic link state algorithm, partial link state information is distributed in the network. This information is then used for route calculation. OLSR provides optimal routes (in terms of number of hops). The protocol is particularly suitable for large and dense networks as the technique of MPRs works well in this context."

1.5. **Chosen Routing Solution**

The following table summarizes the pros and cons of the algorithms discussed above:

<table>
<thead>
<tr>
<th></th>
<th>DSR</th>
<th>OLSR</th>
<th>Flooding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network Congestion</strong></td>
<td>Lowest if route is known</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Network Congestion (per message)</strong></td>
<td>otherwise - High</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>High Traffic Tolerance</strong></td>
<td>Highest if mobility is low</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td><strong>High Traffic Tolerance</strong></td>
<td>otherwise - Low</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transfer Time</strong></td>
<td>Short, but with High Variance</td>
<td>Medium</td>
<td>Shortest (if no congestion)</td>
</tr>
<tr>
<td><strong>Transfer Time (per message)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Control Packets</strong></td>
<td>Per new route</td>
<td>Constantly</td>
<td>None</td>
</tr>
<tr>
<td><strong>Simplicity</strong></td>
<td>X</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td><strong>Packets Size</strong></td>
<td>Increases with path length</td>
<td>~ Data</td>
<td>~ Data</td>
</tr>
<tr>
<td><strong>Mobility tolerance</strong></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td><strong>Mobility tolerance</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Idle Bandwidth</strong></td>
<td>None</td>
<td>High</td>
<td>None</td>
</tr>
</tbody>
</table>

*Table 2 - Pros and Cons for routing protocols*

Before comparing the different columns, we must emphasize one important assumption of our solution to the mobile ad hoc problem:

Our solution will focus on the transfer of simple text messages (similar to SMS). This means that the amount of traffic generated by a single send attempt of a certain node is relatively small. This assumption is of high importance when comparing the different solutions.

For instance, OLSR constantly floods control datagrams in order to maintain the synchronization of the topology data across the network. When the common case deals with high traffic datagrams, the effect of these control datagrams on the total traffic may seem negligible. On the other hand, when the common case is short low-traffic text messages, we may reach a situation where the control datagrams constitute most of the traffic in the network. In such a case, there is no advantage to flooding control datagrams, when a node can skip this overhead.
and simply flood its messages. Therefore, in our case – the OLSR algorithm has no advantage over the simple data flooding solution.

As for DSR, its performance varies significantly as a function of mobility vs. amount of traffic. The protocol has a high overhead whenever a new route needs to be established, but is quite efficient if it succeeds in passing a large amount of data using a certain route once it is established. As before, we assume that the nodes using the application will not generate a high amount of traffic, and that the mobility factor of the network (comparing to the traffic) is high. Therefore, the DSR algorithm has no significant advantage as well over the simple data flooding algorithm.

Thus, our conclusion was to use data flooding (with some optimizations) as our main routing algorithm.

1.6. Requirements

1.6.1. General Features
The following requirements should be met:

- Text messaging
- Unicasting and broadcasting shall be possible
- Delivered unicasted messages shall be ACKed, allowing for reliability
- Two devices with no line of sight shall be able to communicate via relays (assuming the relays form a continuous chain between the source and destination)
- Mobility and the existence of multiple routes shall not disrupt communication
- The user shall not be involved in the routing process

1.6.2. Functional Requirements
The following table expands the general requirements into smaller and more measurable requirements. The left columns describe a requirement and the right columns contain indices of a test from 0, testing the requirement.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The user shall be able to send a message to a specified destination</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>The user shall be able to choose a destination from the phones contact picker</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>The user shall be able to choose broadcast as the destination address</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>The user shall be able to choose multiple destination: from the contact picker and contact list</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Choosing the same destination from the contact picker and contact list only sends the message once</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>The user shall be able to make himself visible (meaning: he will appear in the “online” contact list of other devices in the network) without sending them a message</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>When broadcast is selected, the message is not unicasted to any device, even if they were selected before we checked “broadcast”</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>Two devices with no line of sight shall be able to transfer a message and ACK using a continuous chain of relays</td>
<td>8.4.2</td>
</tr>
</tbody>
</table>
9  Sent messages shall appear in the outbox with a rolling circle 6
10 ACKed messages shall appear in the outbox with a V icon 6
11 Timed out messages shall appear in the outbox with a X icon 6
12 A broadcast message in the outbox shall have a fitting icon 4
13 The user shall be able to reply to a message he sees in the Inbox by touching it and choosing reply 8
14 The user shall be able to resend a timed-out message he sees in the Outbox by touching it and choosing resend 9
15 If an ACK is received after a global timeout (see 2.3.1 or glossary), the X icon in the outbox shall change to V 9
16 Messages in the Inbox and Outbox shall be sorted by date 10
17 Rotating the screen and/or entering and exiting the Debug, Inbox and Outbox activity shall not cause loss of data in the compose activity 11
18 A message arriving for the second time shall not prompt any notification or toast 19
19 A broadcast message shall not appear in the Inbox of the device which created it, even if that device receives that message later from another device. 18
20 The application shall not need user interference to route datagrams which are not destined for the device running the application 15

Table 3 - Functional requirements

1.6.3. Non-Functional Requirements

The following table lists the criteria by which the performance of the solution will be tested. For each tested quantity, ‘Minimal’, ‘Target’ and ‘Outstanding’ goals are set. Our implementation must, at the very least, be on par with the minimal goals.

All latency related goals refer to the 90\textsuperscript{th} percentile. This means that a goal is considered ‘met’ if it has been exceeded in 90\% of cases (e.g. by 90\% of the datagrams sent in any test).

Early in development, it was found that a device will take longer to send/receive datagrams if it has not yet established any wireless connections. This is referred to as a ‘Cold Start’ scenario, abbreviated to ‘Cold’ in the table. The complementary is a ‘Warm’ start.

Each trait also refers to the performance tests that tested for it, by id (see Performance Testing in Section 8.5).

<table>
<thead>
<tr>
<th>ID</th>
<th>Trait</th>
<th>Details</th>
<th>Goals</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Single hop one way (Cold)</td>
<td>The latency, in seconds, until a unicasted/broadcasted datagram reaches a single available neighbor.</td>
<td>Minimal: &lt;60 [sec] Target: 30 [sec] Outstanding: 5 * [sec]</td>
<td>1,2,4,6,9</td>
</tr>
<tr>
<td>22</td>
<td>Single hop one way</td>
<td>See above, only tested on two devices with a preexisting</td>
<td>Minimal: 10 Target: 7</td>
<td>3,5,9,10</td>
</tr>
</tbody>
</table>

\* We eventually chose a Bluetooth based implementation. The absolute minimum latency for creating a new connection is 12 seconds, as dictated by the Bluetooth discovery mechanism.
<table>
<thead>
<tr>
<th>No.</th>
<th>Requirement Description</th>
<th>Description</th>
<th>Minimal:</th>
<th>Target:</th>
<th>Outstanding:</th>
<th>Devices Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Single hop round trip (Cold)</td>
<td>The latency, in seconds, until an ACK is returned after unicasting a datagram to a single available neighbor.</td>
<td>70</td>
<td>35</td>
<td>6</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>24</td>
<td>Single hop round trip (Warm)</td>
<td>See above, only tested on two devices with a preexisting wireless connection.</td>
<td>20</td>
<td>10</td>
<td>2</td>
<td>3,5</td>
</tr>
<tr>
<td>25</td>
<td>Relay (Cold)</td>
<td>The per-hop latency of a datagram being forwarded between devices. Measured between 'Cold' devices.</td>
<td>60</td>
<td>30</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>26</td>
<td>Relay (Warm)</td>
<td>See above, only between devices with a preexisting wireless connection.</td>
<td>10</td>
<td>7</td>
<td>1</td>
<td>8,11</td>
</tr>
<tr>
<td>27</td>
<td>Neighbor Packet Loss</td>
<td>The percentage of lost datagrams between two neighboring devices.</td>
<td>5%</td>
<td>0.1%</td>
<td>No loss in tests</td>
<td>1,2,3,4,5,6,9</td>
</tr>
<tr>
<td>28</td>
<td>End-to-end packet loss</td>
<td>The percentage of lost datagrams between two distant devices</td>
<td>10%</td>
<td>1%</td>
<td>No loss in tests</td>
<td>7,8,10,11</td>
</tr>
</tbody>
</table>

Table 4 - Non functional requirements
2. Proposed Solution

In this section we will describe the main building blocks and algorithms of our solution, without regard for any specific platform or RF technology.

- Section 2.1 presents the datagram – the basic bundle of data to be passed among different nodes. It describes the reasoning behind the fields we chose to include in the datagram.
- Section 1.3 introduces the layered architecture of our solution.
- Section 1.4 presents a quick review of the main routing algorithm, and the challenges it may encounter.

2.1. The Datagram

In designing our solution, the very first class we looked into was the datagram. This is to be the basic packet of information changing hands in our ad hoc network. The following structure, naturally similar to an IP packet, was decided upon:

![Datagram Design](image)

The datagram consists of the 6 data fields (mentioned below) and 3 size fields. The source and destination have no constant size (our application will typically use phone number, but we left this for a more general implementation). Therefore, a size field is required to parse both addresses. In addition, we also have a size field for the payload.

- The source and destination addresses are unique identifiers across the ad hoc network.
- The timestamp is the time when the datagram was created, according to the creating node.
- The type is an element of the set \{Message, Ack, Control\}
- TTL (Time to live) – hops left until the datagram will be discarded.
- Payload – the encapsulated data.

The global destination and source addresses are unique across the network. It is entirely possible to utilize the physical addresses (such as the Bluetooth device’s MAC) as global addresses as well, but we decided to leave the option for generalized identifiers. Indeed, in our implementation we opted for phone numbers as global addresses.

The timestamp is one key difference between this structure and IPv4 datagram. The triplet \( (\text{source}, \text{destination}, \text{timestamp}) \) should enable us to uniquely identify different datagrams and...
more importantly: copies of the same datagram. This is working under the assumption that the end user will never be able to create two datagrams with the same triplet through the UI.

Identification of copies is crucial in enabling flooding or broadcasting: Any node receiving an already known datagram will not flood it again, thus circumventing any possibility of broadcast storms [8] in the network. Let the reader be reminded that flooding is necessary in all the discussed ad hoc routing solutions – at the very least for control datagrams. For this reason, it makes sense to make such a design choice at this point, without regard for the routing algorithm that will eventually be chosen.

The time to live field is used exactly as it is in an IP packet, being reduced on each hop. Any node receiving a datagram with zero TTL will not forward it further, preventing any datagram from being routed forever.

It should be mentioned that infinite routing is extremely unlikely even if we do not use the TTL. Such a scenario is only possible in a network to which new devices keep joining indefinitely. This unlikely contingency could have also been thwarted by limiting the age of the datagram as calculated from the timestamp (a more literal ‘time to live’), assuming we somehow achieve time synchronization between the different nodes. We eventually decided to keep the somewhat redundant TTL field as it could be utilized for interesting data analysis (hop by hop route tracing, for instance).

Data implementation is described in 3.3

2.2. Architecture

A three layer partition of responsibilities was designed, offering fair modularity alongside simple implementation:

- **Layer 3**
  - Routing protocol implementation
  - Send/Receive to/from entire network
  - Relays en route datagrams through the node
  - Optional capabilities:
    - Unicast, Multicast, Broadcast, Discover (entire network)
  - Optional reliability
  - Uses layer 2 to communicate with immediate neighbors

- **Layer 2**
  - Send/Receive datagrams to/from immediate neighbors
  - Wraps layer 1 and presents unified interface for layer 3
  - Optional capabilities:
    - Unicast, Multicast, Broadcast, Discover (immediate neighbors)

- **Layer 1**
  - API for RF communication (supplied by the OS of the device).  

Page 20 of 113
Each layer can be implemented in any number of ways. The following diagram illustrates several different alternative stacks. The index in the left column identifies the layer number:

![Diagram of different alternative stacks](image)

**Figure 2 - Alternative implementations in the layered architecture**

At the time when this model was decided, we have not yet known what RF technology we would eventually use. However, with this model already in place, we were able to proceed with development despite major unknowns of this sort. This model is also expected to allow us to quickly update or upgrade our application: switching between Bluetooth and Wi-Fi direct, or between different routing protocols, while leaving the application level code unchanged. It is also possible to simultaneously work on implementing and testing different layers, while mimicking the adjacent layers with ‘stubs’.

Further delegation of responsibilities to additional layers was considered before consolidating into the three layer model:

- A reliability layer (OSI layer 4) was too small to justify a full layer (in practice – an abstract class and its implementation), so it is integrated into layer 3.
- A presentation layer (OSI layer 6) was considered which would bundle and parse data structures into and out of the payload of the datagram. We concluded that this job could easily be performed by the application itself, and that there is no need to encapsulate such a layer.

By not over-compartmentalizing the responsibilities, we avoid the overhead of wrapping little functionality with cumbersome abstract classes and event firing/handling.
2.3. Data Flooding Algorithm

2.3.1. Basic Algorithm
In a nutshell, relevant datagrams are flooded to all immediate neighbors, which flood the datagram again recursively, until each node hears all the datagrams. Datagrams arriving to the destination device are picked up from the stream, whereas the rest are flooded forward.

To ensure each datagram is handled for only a finite amount of time:

1. The datagram has a TTL field that is reduced on each hop.
2. Each node will only send the datagram to its neighbors a finite number of times (see in Mobility Tolerance, section 2.3.2).
3. Each node remembers the previously handled datagrams. If an old datagram makes another appearance, it should not be flooded again as it is not a new datagram.
4. Each node remembers per each datagram, which nodes must have already seen the datagram. Those are the immediate neighbors to which the datagram was successfully sent to or received from.

For the purpose of comparing new datagrams to old ones, equivalent datagrams are defined as having the same triplet:

1. Source ID
2. Destination ID
3. Time stamp (millisecond resolution)

This is based on the fair assumption that no two datagrams in the ad hoc network should share all of the above parameters.

One immediately obvious optimization is not to send datagrams to neighbors that already know them. We indeed implement such an optimization, which means every node keeps track of which datagrams are known by what neighbors.

In our algorithm we assume that the underlying data link layer (e.g. Bluetooth) is reliable. This means that if an indication is received from that layer that a certain datagram was successfully sent to a certain neighbor, then this status is final. Naturally, if a certain neighbor sends a particular datagram, this allows another way of deducing that that neighbor already received that datagram.

On top of this reliable data link layer, we provide a best effort routing layer by the flooding mechanism.

In addition, we implement a classic ACK / Timeout mechanism. When a user sends a message to a distant node, a countdown is started. If an ACK datagram is not received by the end of the countdown, we say that a Global Timeout event has occurred. The user is notified of this and may choose to resend the message. The term ‘Global’ was chosen to emphasize the end-to-end
nature of the mechanism. It also differentiates this mechanism from the Local Timeout mechanism which will be introduced next.

2.3.2. Mobility Tolerance

In our minds eye, the ad hoc network should offer fair reliability even in the face of frequent topology changes, meeting the challenges of the mobile ad hoc network. Furthermore, it should take advantage of some of the opportunities presented by mobility. For example, we want the datagrams to be able to traverse disparate connected components by piggybacking on travelling nodes. We propose the following mechanism to both tolerate and take advantage of the mobility problem.

Each node will perform several Flood Attempts. After each attempt we count down to a Local Timeout. After that timeout, another flood attempt is made. Each flood attempt consists of:

- Resending to already known neighbors for whom the previous send attempt failed.
- Sending to any new neighbors.

It is important to stress that all the nodes in the route follow this scheme - not only the original sender. This mechanism allows for:

1. Occasional failure to connect to a certain neighbor via layer 2.
2. Topological changes: Alice sends to Bob. Bob would have sent to Charlie but Charlie changes locations quickly and now is only in the Alice’s transmission range. Luckily Alice re-floods the datagram to any new neighbors, including Charlie.
3. Traversal of disparate connected components: Alice’s component is not connected to Bob’s component at any time. However, Charlie frequently moves between the two components. After Charlie changes components a local timeout occurs. Charlie re-floods datagrams from Alice’s component into Bob’s component and vice versa, thus allowing for communication between the otherwise isolated components.

Such a protocol requires that each node keeps a state for each datagram. This state is comprised of:

1. The remaining flood attempts.
2. The time of the next local timeout.
3. Which neighbors know this datagram (This was already a requirement in the Basic Algorithm description 2.3.1).

Based on this data, Pending Datagrams are selected for eventual routing. Pending datagrams:

1. Are not destined for the device running the application (this includes broadcast datagrams).
2. Still have time to live (TTL > 0).
3. Have remaining flood attempts.
Timed-out pending datagram are pending datagram for which the local timeout has expired. The timed-out pending datagrams are enqueued for flooding. After a flood attempt is made on a datagram, the number of its remaining flood attempts is decremented and the next local timeout is set.

A possible starvation scenario would be an ‘always busy’ router that handles the newest datagrams and never gets to the oldest. For instance, a user that keeps sending new messages will cause the router to ignore the old ones. To prevent such starvation, the most urgent datagram (which has waited longest since its timeout) is handled first.

For the full implementation of the protocol, see Section 5.3.7.

3. Android / Bluetooth Implementation Overview

In this section we will describe the design of our solution, based on the architecture presented in the previous section. The solution was written for Android devices, based on the Bluetooth protocol.

- Section 3.1 presents basic terms relevant for Android development.
- Section 3.2 presents an overview of the design and the different classes we used.
- Section 3.3 presents the implementation of the datagram class.
- Section 3.2 presents the interface for layer 2 classes.
- Section 3.2 presents the interface for layer 3 classes.
- Section 3.2 presents different flows (described by Message Sequence Charts) containing multi-layer communication.

3.1. The Android Environment

Before discussing the architecture and design of our solution, we must first introduce some basic components of the Android development API. The components mentioned below are often used in different parts of the project. The terms will be described shortly, focusing on their role at our project. For full detail explanation, see Android development guide [5].

3.1.1. Activity

An Activity is a GUI component of the application. Every activity consists of a layout, where different widgets are located (different buttons, text fields, etc.). Each operation of a user is translated to a certain action in the activity.

3.1.2. Content Provider

A content provider is a component providing an interface to a structured set of data. Content providers are the standard API for multi-process communication in the Android environment. In our application, a content provider will be used to access the phone’s contact list.
3.1.3. Service
A service is an application component used to perform some work in the background without prompting the user. A service may run when the application itself is in the background. By default, services run in the main thread of their hosting process—the service does not create its own thread and does not run in a separate process (unless specified otherwise). Nevertheless, in our application, the core services will run separate threads. For example, we use services for routing (Section 5.3).

3.1.3.1. Intent Service
An intent service is a special type of service, running on its own thread. Intent services are used for handling asynchronous requests. For example, we use an intent service for neighbor discovery (Section 4.4), as the Android API publishes asynchronous intents informing of device discovery.

3.1.4. Broadcast Receiver
A broadcast receiver is an application component designed to register and handle intents (abstract data structures informing on different events, see 3.1.5 below) published in the system. The publisher for the intent may be another component in our application, or an external component from another application. The broadcast receiver uses a filtering mechanism (using “intent filers”) in order for it to be invoked only when specific interesting intents are published. For example, our application uses a broadcast receiver for the neighbor discovery mechanism. The BluetoothDiscoveryService (Section 4.4) listens to the events of Bluetooth discovery start, Bluetooth discovery end and new Bluetooth device discovered.

3.1.5. Intent
Intents are the standard communication protocol between different application components. Intents may be notifications of certain events (usually called “actions”) read by broadcast receiver, a structured set of data passed to an activity when it is started/resumed/stopped or a command to perform a certain action when passed to a service. For example, when the user wants to reply to a message he read in his inbox, the inbox activity will send an intent to the compose activity with the recipients’ ID.

3.2. Object Overview
In this section we will present a quick overview of the different modules and their relations. The next section will provide a full detail description for all the mentioned modules.

The following chart describes the different modules in our application:
Figure 3 - The layered architecture of our solution

Below is a short description of all classes, starting from the application layer, descending to the layer 2 classes. It is followed with an in depth description of some of the classes, in an order more natural to the understanding of the solution.

- The AdhocApplication module is the main module of the application. It holds single instances of the layer 2 and layer 3 classes. It also provides an interface for querying the user ID (usually phone number) by different modules in the application.
- The HelloService is a service used to flood control messages over the network, allowing new nodes to familiarize themselves to the network.
- The application has five basic activities: Compose activity to choose destinations and compose the message for sending, InboxActivity to see incoming messages, OutboxActivity to see outgoing messages, a DebugActivity to see all currently en routed datagrams and see some general status fields, and a TestActivity that will aid performance testing.

In addition, a SplashActivity exists. This activity will always be alive in the background and will also be responsible for notifying the user even when the application is not in the foreground (by presenting toast messages, notifications and playing sounds).
The Preferences class handles the different configuration (mostly numeric constants) of the application. Its GUI is generated automatically from an Android specific preference XML resource.

Layer3Interface/Layer2Interface sets the interface implemented by a class implementing the relevant layer. The interface contains methods for sending datagrams (unicast, multicast and broadcast, where at least one of the three must be implemented) over a direct link (to immediate neighbor) / to nodes in the entire network, an event responsible for notifying of incoming datagrams, and a discover method for discovering immediate neighbors / network contacts. The interfaces are described in 3.4 and 3.5.

The DataFlooder class implements the layer 3 interface, and uses the Routing Service class to implement the flooding algorithm. These classes are described in 5.

The BluetoothWrapper class implements the layer 2 interface, using the Android Bluetooth API. It uses the BluetoothDiscoveryService to discover neighbors, uses the DeviceManager to manage known and connected devices and uses the BluetoothBlacklistService to filter devices that surely do not have the application installed (because they use another Bluetooth interface). All the class of layer 2 are described in 4.

The DbWrapper and PhoneUtilities are both utility classes to be used from the modules mentioned above in order to access the database and handle different phone number formats.

The Datagram class implements the datagram we introduced in 2.1. The class implementation is described next in 3.3.

### 3.3. Datagram(Class)

The Datagram object contains the fields mentioned in the datagram discussion in section 2.1. The timestamp is kept as a java Date object, which in itself is built around a long value indicating the elapsed milliseconds since the 1970 epoch. The global source / destination are strings, as well as the payload. The datagram type is an enumeration of {Message, Ack, Control}.

ACK datagrams have a distinct triplet, different then the ACKed message. In order to reference the ACKed message, its triplet has to be deduced. Since the ACK datagram and the ACKed message share the same (inverted) global addresses, it is sufficient for the ACK datagram to reference the timestamp of the ACKed message. This is accomplished by keeping the millisecond value of that timestamp, in string form, in the payload of the ACK datagram.

The Datagram class also has methods for comparison, ack/acked matching, serializing/deserializing to/from a byte array. The class also overrides the equals/hashCode abstract java methods, which allows us to use it in standard java containers.
3.4. Layer2Interface (Abstract Class)

This interface is used by the layer 3 class to discover, send to and receive from near-by devices in the device’s reception and transmission range. This interface can be implemented by any class wrapping a peer to peer communication protocol API, like Bluetooth or Wi-Fi direct. The basic features of the interface are:

- **unicast (Id, Datagram)**
  Transmits the datagram to the device whose physical ID (used for peer to peer communication, such as MAC address) is set in the Id field.

- **multicast (List<IDs>, Datagram)**
  Transmits the datagram to the devices specified in the list (This capability was not implemented by the BluetoothWrapper class).

- **broadcast (Datagram)**
  Transmits the datagram to all the nodes in transmission range (This capability was not implemented by the BluetoothWrapper class).

- **discover()**
  Returns a list of all nodes in the device’s reception range.

This class also raises a Layer2ReadEvent, to which other classes may register. This event notifies the listener of any incoming datagram from an immediate neighbor. It is important to note that such an event does not always mean a datagram had reached its destination, as the datagram may have been sent to the receiving device for the purpose of routing.

3.5. Layer3Interface (Abstract Class)

This interface is used by the application to discover, send to and receive from devices in the entire ad hoc network. This interface can be implemented by any number of routing protocols, which in turn use a generic Layer2Interface which may be implemented by any number of peer to peer RF communication technologies.

- **unicast (Datagram)**
  Sends the datagram to the address specified in the datagram itself. The target address may be anywhere in the ad hoc network.

- **broadcast (Datagram)**
  Sends the datagram to every single node in the network.

- **multicast (Datagram)**
  Sends the datagram to a group of several target nodes. (This has not been implemented in our Data Flooding implementation).

- **discover()**
  Returns the set of all known nodes in the entire ad hoc network.

This class also raises the Layer3Event, to which other classes may register. This event notifies the listener of either an incoming datagram or of a failure in unicasting one of the outgoing
datagrams. This failure notification, if implemented, is meant to be utilized as a reliability mechanism (see Reliability, Section 0).

The `Layer3Interface` class also has a public String representing the ‘broadcast’ address. Incoming broadcasted datagrams will have this string in place of the usual destination address. The string is made public by this class so that user classes may recognize it when it appears in incoming datagrams.

### 3.6. Message Sequence Charts

#### 3.6.1. Application Usage Flow

The following flow describes an overview of the application usage by different users. We describe a scenario where four devices (marked A-D) are in a row where each device can transmit to or receive from only its nearest neighbors from each side (left and right). User B (red) wants to send a message to user D (blue).

The flow describes how the message is distributed: B will send the message to A and C. Device A has only one neighbor (B) from which it received the datagram. Therefore, A does not have any device to send the message to. C will forward the message to D. Upon receiving the message, D will generate an Ack that will spread back to B. Devices B, C and D have received the ack for the sent message – therefore they will stop trying to forward the original message, unlike device A, which will periodically try to discover new neighbors and send the message. In addition, both C and D will continue in trying to spread the ACK meant for B, as they have no indication that the ACK reached its final destination. B will not forward the ack, since B is the ack’s destination.
The following flow describes the process of discovering nearby devices. The flow starts when the routing service decides to issue a discovery request. The decision is based on a heuristic which implies that service is idle, as described in Section 5.3.3. The request is passed to the BluetoothWrapper, which is responsible of implementing the layer 2 interface. The BluetoothWrapper performs the discovery request by invoking the BluetoothDiscoveryService, which registers for intents from the Android OS. Using the information retrieved from those intents, the service builds the neighbor list which is eventually returned by the BluetoothWrapper. It is worth mentioning that the Android API discovery mechanism is implemented by callbacks, whereas the layer 2 interface provides a synchronous discovery method. This demonstrates the layered architecture at best – The layer 3 routing service is completely unaware of the actual discovery mechanism (provided by the operating system), as it should only be familiar with the behavior of the layer 2 interface.

For a more detailed discovery flow in layer 2, see Section 4.8.
3.6.3. Compose Flow

The compose flow is responsible for handling user requests for sending a message. The flow starts with the user choosing destinations, and writing a message in the compose activity. Afterwards, a call to the layer 3 DataFlooder is called (may be unicast or broadcast) with a request so transmit the message. The flow ends with the datagram being inserted into the datagram table. This ends the flow, as our routing algorithm handles all datagrams in the same manner, by flooding them to all possible neighbors. Therefore, by inserting the datagram to the table, we make sure it will be handled later, exactly like any other datagram which may have been received from a different device. The routing itself will be done by the forward flow (Section 3.6.4).
### 3.6.4. Forward Flow

The forward flow is responsible for routing different datagrams. It starts with the layer 3 routing service querying the database for pending timed-out (see 2.3.2) datagrams to send. For each neighbor and for each datagram (not necessarily, see Section 5.3.4 for a more exact description), the service will invoke a call to the layer 2 `BluetoothWrapper`'s `unicast` method. The `BluetoothWrapper` will serialize the datagram and pass it to the chosen neighbor using the Android API. If the delivery is successful (assuming the reliability of the Android API), the `BluetoothWrapper` will notify the calling layer 3 service, thus updating the Transaction table (Section 5.1), which in turns helps the device determine which of its neighbors already hold a copy of the datagram.
3.6.5. Receive Flow

The receive flow completes the main flows picture. It starts with an indication from the Android OS of new received data. The layer 2 BluetoothWrapper (Section 4.6) handles this notification. It is responsible for serializing the raw data into our familiar datagram (Section 2.1) and raising an event for layer 3. Upon receiving the message, the layer 3 DataFlooder needs to decide its way of action, based on the following criteria:

- If the datagram is new – it is inserted into the datagram table.
- If the datagram is an ACK, and the original message is already in the table, set “ACKed” for the original message (so it will not be forwarded any longer).
- If the message is a new message which is destined for the receiving device, the user shall be notified (by sound, toast and Android notification). In addition, the different activities shall be updated.
- If this is a new message destined for the receiving device (and not a broadcast message), a new ACK datagram shall be generated and sent back to the source of the message.
4. Layer 2 Implementation

4.1. Bluetooth Implementation – Section Overview
In this section we will review the four modules constituting the second layer of the design.

- Section 4.2 presents a general overview of these modules
- Section 4.3 presents the DeviceManager class
- Section 4.4 presents the BluetoothDiscoveryService class
- Section 4.5 presents the BlacklistService class
- Section 4.6 presents the BluetoothWrapper class
- Section 4.7 presents the accept flow (accepting new connections)
- Section 4.8 presents the discovery flow (discovering new neighbors)
- Section 4.9 presents the connection flow (using and maintaining existing connection)
- Section 4.10 presents the unicast flow (sending a datagram)

4.2. Bluetooth Implementation – General
The main role of the classes in this section is to implement (partially) the layer2interface (Section 3.4). The layer 2 interface itself is inherited by the BluetoothWrapper class (Section...
4.6), where each of the other classes is used for a specific purpose, and managed by the BluetoothWrapper. The four classes presented below manage the application’s calls to the Bluetooth adapter of the Android phone. They manage connections with other Bluetooth devices, handle data transmitting and receiving and provide the upper layers of the design with an API, based on their need. The basic features of these classes are:

- Discovering immediate neighbors.
- Unicast (sending a datagram to a specific immediate neighbor).
- Receive – Reading a datagram sent from a certain neighbor, and publishing an event to the upper layers.

The classes described in this section are:

- **DeviceManager** (Section 4.3) – responsible for holding information about all discovered devices (“neighbors”) and devices which the current device is connected to.
- **BluetoothDiscoveryService** (Section 4.4) – an Android intent service (a service running in a separate thread) responsible for managing the discovery process.
- **BlackListService** (Section 4.5) – an Android intent service responsible for intercepting pairing requests (built by the OS), querying the connection method (PIN code or Passkey) and filtering devices that cannot have the application installed.
- **BluetoothWrapper** (Section 4.6) – This is the main class which contains most of the code of our layer 2 implementation. This class uses the services of the other three.
4.3. DeviceManager (Class)

The DeviceManager maintains four data structures:

- Last discovery list - a list of items of type BluetoothDevice, that holds the devices discovered since the last discovery started. The list is cleared when discovery starts, and updated whenever a device is discovered (see Section 4.8 for discovery flow).

- Connection Thread map – A map with keys of type String (MAC addresses) and values of type ConnectionThreadPair, holding information about the devices to which a connection exist at the present time. The value is a pair of connection threads, as we allow two connections for a certain device (Incoming and Outgoing), although one of the connections is bound to close in a stable state. Whenever a new connection is made, relevant connection thread will be added to the map. When the connection closes, it will be set as null in the connection pair. If both connections are null, the entry is removed from the map.

- All known map – A map (hash table) with keys of type String, representing the MAC address of the discovered Bluetooth devices, and values of type BluetoothDevice. This map will hold all the Bluetooth devices that were discovered since the application started. When a new device is discovered, it will be added to the map. If later a device with a known MAC address is discovered, the map will be updated (the new BluetoothDevice instance will replace the old one). In addition, we must take into consideration that device A may be acquainted with neighbor device B not only through direct discovery. That is, in case device B requests to connect to device A before A ever discovered B. In that case, device B shall be added to the all known map as well.

- Blacklist – A list of device addresses, which were identified as devices that do not have the AD-HOC application. This list is used to filter the result of the Bluetooth discovery (we have no interest in devices that do not have the application) or to return an immediate failure of a unicast request, in case the device was black-listed only after the discovery returned (In the common case – the device will be black listed only after a connection attempt, meaning – after the first discovery).

The DeviceManager holds various methods for querying those data structures and managing them. For example:

- **ClearLastDiscovery()**
  Clears the last discovery list.

- **AddNewConnection(BluetoothDevice device, ConnectedThread ct, boolean isClient)**
  Adds the connection thread to the connection map, and updates the all known map.
  The device parameter is used to update the “all know” map.
  The ct parameter is used to update the connection thread map.
  The isClient parameter is a flag indicating the position of the new connection in the connection pair (Incoming or outgoing).
• LastDiscoveryPair()  
  Creates a list with items of type IdAndNamePair, based on the last discovery list. The ID of each item will be the MAC address of the discovered Bluetooth device, and the name will be the device name, given by the user of that device (this information is part of the BluetoothDevice class, it is retrieved when the device is discovered or when it initiates a connection).

For a full description of the different method of the DeviceManager (mostly get/set or add/remove, which may be a little more complex as they maintain several data structures) see the Java doc [6]

One interesting method implemented in the DeviceManager is getThread of the ConnectionThreadPair. The method is used to break the symmetry between to devices and allow them to communicate using a single connection. In case only one of the connections in the pair exists, the method will return that connection. Otherwise, it will return the connection thread for which the MAC address of the client side is lexicographically smaller. As we would not want to double the number of connections we use, it is important to break symmetry and make sure both devices use the same connections. This will help us when we would like to safely close the second connection.

For access convenience, the DeviceManager is implemented as a singleton. Since we make sure the application has a single instance of a layer 2 class, this does not cause any correctness issues. It is accessed from both the BluetoothWrapper class and the BluetoothDiscoveryService class (to update the discovery list).

In addition to the data structures mentioned above, the DeviceManager holds a lock to synchronize the DiscoveryService (Section 4.4). The DeviceManager publishes wait and notify methods for that lock. The wait method is called from the BluetoothWrapper (Section 4.6) class when it starts the DiscoveryService, and the notify method is called from within the service, before it dies. The reason that the DeviceManager is responsible for that lock, is to notify external classes (specifically the BluetoothWrapper) when the last discovery list is updated and may be accessed.

4.4. BluetoothDiscoveryService (Class)
The DiscoveryService is a simple Android Intent Service (Section 3.1.3.1). When the service is created, it uses the API of the Bluetooth adapter to start the discovery of immediate neighbors. The heart of that service is a broadcast receiver, which registers for three events sent by the Android OS, and handles each of them, as described below:

• ACTION_DISCOVERY_STARTED  
  Indicates the start of the discovery. The service calls the DeviceManager, and clears the last discovery list.
• **ACTION_FOUND**  
Indicates that a Bluetooth device was discovered. The service filters discovered devices by their device class (and by the blacklist, see 4.5) as we do not want to include devices that cannot run our application. If the discovered device has a valid device class and is not blacklisted, the DeviceManager’s discovery list is updated with the new discovered device.

• **ACTION_DISCOVERY_FINISHED**  
Indicates the end of the Bluetooth discovery. Upon receiving this event, the BluetoothDiscoveryService calls the DeviceManager’s notify method, to release the discovery lock.

### 4.5. BlacklistService (Class)

The blacklist (managed by the DeviceManager) is a list of devices which are excluded from any other data structure in the DeviceManager (see 4.3), and to which connection attempts will not be made. The importance of the blacklist was to fulfill the requirement that the routing will not require any active interaction with the user, as specified in 1.6.2. During development, we discovered that when certain (older) devices were nearby, our application constantly prompted a dialog asking the user for a PIN code. The problem was caused by devices that had an older version of Bluetooth, which uses a different connection protocol. As the device we were using was compatible with the older Bluetooth version as well, it prompted the dialog asking for the user for a key to complete the connection.

The BlackListService is implemented by a simple Android Intent Service. The heart of this service is a broadcast receiver, which registers for a single event, and handles it as described below:

• **ACTION_PAIRING_REQUEST** ("android.bluetooth.device.action.PAIRING_REQUEST")  
Indicates the device is submitting a pairing request to another device. This action is published by the Android OS, and is not part of the standard Bluetooth API. Nevertheless, since the intent filtering mechanism is string-based, by knowing the relevant string (which we can find out, as the Bluetooth Android API is open-sourced), we can intercept the relevant intent and respond to it.

Upon receiving the intent, the BlackListService reads the EXTRA_PAIRING_VARIANT field, indicating the Bluetooth connection mode. If the field is of type PAIRING_VARIANT_PIN, it means the connection requires a PIN code, thus – the remote device cannot have the application installed (The Bluetooth connection protocol is too old). In such case – the device address will be added to the blacklist of the DeviceManager.

### 4.6. BluetoothWrapper (Class)

The BluetoothWrapper class is responsible for the direct implementation of the layer 2 interface (Section3.4). It creates connections (both incoming and outgoing), detects connection loss and handles data transmitting and receiving. To this end, this class manages several threads
of five different types. Each thread is responsible for a certain part in one of the four main flows of Bluetooth layer 2. The implementation of the different threads constituting this class is explained below (4.7, 4.8, 4.9, 4.10)

4.7. Accept Flow

Accept flow – The main limitation of the Bluetooth API is that devices cannot create new connections (not incoming nor outgoing) while discovering neighbors. To ensure reliability, we must make sure that the device is accepting new connections a significant part of the time. Therefore, we would like the device to try and accept new connections whenever it is not discovering. In addition, we assume that layer 3 is responsible for not calling the `discovery` method too often. This will imply that the device will be in an accepting state most of the time.

The basic accept flow is as described in the following chart:

![Blu常规实现，接受流程](image)

**Figure 10 - Bluetooth implementation, accept flow**

In our solution, the discovery/accept state switch mechanism is implemented by two threads (accept thread and watcher thread) and a `discovery` method, responsible for canceling the accepting thread before starting discovery. The implementation of the threads is described below:

- Accept thread – The accept thread is responsible for accepting new connections. It contains a server socket, which is initialized in the creation of the thread. The accept thread waits (using a blocking call) for new connection request.
If a new connection is established – the thread calls the handleNewConnection method, to notify the DeviceManager of the new connection.

The blocking call may be terminated. For instance, if the discovery method is called, we ought to cancel the accepting thread, since the Bluetooth protocol does not allow simultaneous discover and accept.

After the accept call (the blocking call mentioned above) returns (whether successful or not), the StartListener method is called and the previous accept thread exits. The StartListener method is responsible for creating a new accept thread (by using the watcher thread, see next bullet). If the accepting thread was canceled due to discovery, as mentioned above – a locking mechanism prevents the new accept thread from running until the discovery is finished.

- **Watcher thread** – The watcher thread’s role is to verify a successful initialization of the accept thread. It is used to mask Bluetooth related errors, that may fail the creation of the listening socket (simple case – Bluetooth is turned off). If the initialization of the accept thread fails, the Bluetooth adapter of the device is restarted. Since the application runs with a BLUETOOTH_ADMIN permission, which must be confirmed by the user prior to the installation, we are allowed to perform this reset. This behavior is repeated until the accepting thread is initialized successfully, or until the BluetoothWrapper class receives a terminate command.

### 4.8. Discovery Flow

The discovery is initiated by a method of the BluetoothWrapper class (Section 4.6), but most of its logic is done by the BluetoothDiscoveryService (Section 4.4) and the DeviceManager (Section 4.3). The BluetoothWrapper’s part is only to cancel the accepting thread before starting discovery, lock the next accept thread from starting until the discovery is finished and starting the BluetoothDiscoveryService. The DeviceManager’s role is to hold and update the last discovery list, as described in Section 4.3. The BluetoothDiscoveryService is responsible for most of the logic, by responding to events from the operating system, as we already described in Section 4.4.

The discovery flow is initiated when the discover method of the BluetoothWrapper is called, typically by layer 3. The flow is described in the chart below:
4.9. Connection Flow

We assume that the connection flow starts when a new connection is established. This may happen as a result of a successful accept call (incoming connection, see Section 4.7), or as a result of a unicast request (Section 4.10) to a host to which the current node is not currently connected. The connection flow starts after the connection is established, so the connection requests are not part of the flow (see unicast in Section 4.10). The main purpose of the connection flow is to maintain connectivity (using ping requests managed by the Ping Thread described below) and to wait for incoming datagrams. The connection flow is described in the chart below:
The connection flow is implemented by two threads: the connection thread and the ping thread.

- **Connection thread** – This thread is responsible for managing an existing connection. The thread is created and started by the `handleNewConnection` method of the `BluetoothWrapper`, when a new connection is established (for both incoming and outgoing connections). The thread holds two stream objects, input and output stream, retrieved from the socket that holds the connection. While the connection is alive, the thread will typically wait on a blocking read call, until new data is received, or until the thread is cancelled.

  The receiving process consists of the following stages:
  1. Read four bytes, holding the information of the size of the datagram (may be 0).
  2. Clear the “wake up counter” (used to mark that the connection is alive. Explained under the ping thread).
  3. If the received size is zero (ping message), quit (and wait for next incoming data).
  4. Otherwise –read a buffer of the wanted size.
  5. Build a datagram object from the given buffer.
  6. Raise an event indicating of a new received datagram.

In addition, the connection thread object holds a `write` method, to be called externally if the devices need to send a datagram of its own.

- **Ping thread** – A ping thread is an inner thread of the connection thread. Each connection thread holds a single instance of a ping thread. The Ping thread is responsible for sending ping messages (i.e. messages with zero length payload), and verifying that incoming messages are constantly received (no matter if those are ping messages or other messages).
The thread wakes up once in a few seconds (defined by a constant parameter) and sends a ping message. In addition, at each wake up, the thread increases the wake up counter. The wake up counter is set to zero after each received message. If the wake up counter reaches the value of three (meaning – three consecutive sent ping requests without receiving a single message between them), an internal terminate method is called for the appropriate connection thread.

We note that the wake up interval cannot be changed by the preference menu. This is done because in order to assert that the connection is still alive (using ping request), we must know how often to expect ping messages. If any user could change this value, other devices will not be able to know that the rate of the remote device is different.

4.10. Unicast Flow

The unicast flow is handled mostly by the unicast method and the makeDeviceConnection (called indirectly by unicast if a connection to the destination does not exist). It also uses the timeout thread described below. When unicast is called (typically by layer 3), it searches for an existing connection to the destination device, using the getThread method of the DeviceManager (4.3). If such a connection does not exist, it tries to establish one. Before trying to establish the connection, a timeout thread is started to externally terminate the request after a given timeout. Finally, if the connection is established (or existed before) the device serializes and transmits the datagram. Otherwise, it returns a failure indicator.

The "Connection Exists?" check in Figure 13 - Bluetooth implementation, unicast flow is actually more complicated, as it is used to prevent a stable state where two devices have two connections between them (to save resources). While trying to perform this optimization, we must take into account another significant requirement, which is to prevent packet loss as much as possible. Therefore, we cannot immediately close one connection when two connections exist between the same devices (typically this will happen when devices connect to each other simultaneously). This is solved by defining a strict characteristic to when and by whom a connection may be terminated. A connection may be terminated only by the initiator of the connection and only if its MAC address is alphabetically higher than the server’s. If both conditions apply, the initiator will close the second connection upon discovering that two connections exist to the same device.

The unicast flow chart is described in Figure 13 - Bluetooth implementation, unicast flow below. The flow starts when the unicast method is called by layer 3.
**Figure 13 - Bluetooth implementation, unicast flow**

- Connection Timeout thread implementation – A connection timeout thread is created every time a connection attempt is made. Since the Bluetooth `connect` method does not necessarily return, we must provide a solution to ensure responsiveness to the caller. The connection timeout thread holds the socket to which the connection attempt is made as a parameter. After the thread is created, it waits for a timeout (parameter). During that time, the thread may be notified that the connection call returned. In that case, the timeout thread quits and does nothing. If the timeout thread was not notified of the return of the `connect` call, it forces the close of the socket, and causes the `connect` method to throw an exception (which is caught and handled).

By using this mechanism, we ensure that the class using the `BluetoothWrapper`'s API will never encounter a non-responsive call.

**5. Layer 3 Implementation**

In this section we will describe the different modules responsible for implementing the layer 3 interface, and the main flows of that layer.

- Section 5.1 describes the `DbWrapper` class (which envelopes the internals of the DB)
- Section 0 describes the `DataFlooder` class (which implements the layer 3 interface).
- Section 0 describes the `DataFloodingRoutingService` class (which is responsible for implementing the routing itself).
5.1. DbWrapper (Class)
The DbWrapper is a singleton object responsible for defining the schema (table structure) and allowed transactions of a SQLite database that services our application. All other classes use this class to retrieve or update data in a coherent and thread safe manner. Data retrieval takes into consideration the global address of 'this' device. For instance, retrieving the inbox means retrieving messages meant for a certain address. This address is kept as a member. The DbWrapper and the DB schema it defines are tailored specifically for the DataFlooder layer 3 implementation (see discussion of relevant requirements in Section 2.3).

The singleton approach was chosen to ease synchronized access to the database. Any synchronized method on a singleton object ensures that no other thread will enter any other synchronized method simultaneously.

The following tables are held:

- Datagram table. Holds all the fields of a datagram along with metadata: ID, remaining flood attempts and the next local timeout (see Mobility Tolerance in Section 2.3.2). The columns global source/dest and timestamp are commonly globally unique (see Datagram structure in Section 2.1).
- Transaction table. Holds a record of any instance in which a datagram changed hands with a neighbor (i.e. successfully received or transmitted). This enables us to know which neighbor has what datagrams, and also creates data which may be useful for later processing. Each row consists of:
  - DatagramID
  - Neighbor (string)
  - Direction (incoming, outgoing)
  - Time of transaction

5.2. DataFlooder (Class)
This class is an implementation of the layer3interface. It implements the unicast and broadcast methods which can reach any node in the ad hoc network. However, it does not implement the multicast method.

The DataFlooder uses the listener/event mechanism built into the interface to notify the user of incoming datagrams or of global timeouts (see global timeouts under Reliability, Section 5.2.2). This class may utilize any implementation of the layer 2 class to achieve its functionality; however it has been optimized specifically for the Bluetooth implementation of layer 2 (see relevant optimizations in Section 0).

This class works in tandem with the DataFloodingRouterService (Section 5.3), which continuously forwards relevant datagrams to neighbors via layer 2.
5.2.1. Broadcast Support
As the reader may recall, the Data Flooding algorithm dictates that every node hears every datagram in the network, but only relevant datagrams are shown to the user. Based on this mechanism, we easily implement broadcasting by using a predefined string that is planted in the datagram’s destination field. Respectively, a DataFlooder sifting through the passing datagram stream will pick up the broadcasted datagrams in addition to the datagrams meant for it specifically. For the predefined broadcast string we use the same string discussed in the generic layer 3 (Section 3.5).

The originator of the broadcasted datagram is fairly likely to receive his own broadcast at some point. In such a case, the incoming datagram should not raise a layer 3 event and (as usual) will not be flooded again.

5.2.2. Reliability
Just as specified in the basic algorithm (Section 2.3.1), we provide reliability on top of our best effort routing protocol by employing an ACK / timeout system. When a user unicasts a datagram to a distant node, a countdown is set off. If an ACK datagram is not received in time, we say that a global timeout event has occurred. The DataFlooder propagates this event to the user using the failure event (see layer 3 event in Section 3.5).

As discussed in the Datagram class (see Section 3.3), ACK datagrams hold in their payload the timestamp of the original datagram that they are ACKing. When a node receives an ACK, it can quickly deduce the triplet of the original ACKed datagram, concluding that it has indeed been ACKed. This will suppress the would-be global timeout event.

Respectively, the DataFlooder at the recipient side should automatically create and send back ACK datagrams when appropriate.

It must be stressed that after a global timeout event occurs, no action takes place other than notifying the user. The user may decide to send a new copy of the datagram, with a new timestamp, using unicast. Resending will not take place automatically.

5.2.3. User Generated Datagram Handling
When the user creates a datagram, it may be unicasted or broadcasted by layer 3. In our DataFlooder implementation, broadcasting merely replaces the destination field with the predefined broadcast address.

An initial state is given to the datagram and saved as metadata alongside the datagram in the database (see datagram state in the Mobility Tolerance discussion in Section 2.3.2, also see datagram table in Section 5.1). This initial state contains a default initial number of remaining flood attempts and an initial local timeout set to ‘now’, causing the router to immediately handle the datagram. In addition, we note that the datagram is known by the creating device in the DB’s transaction table.
The RouterService (see Section 0) is invoked, meaning it is either created (if there is no running instance) or woken up (if it is in a sleep state). If the router is already busy, invoking has no effect. This step is skipped if the user chooses to send the datagram to himself (in which case no routing should be done).

Datagrams that are of a ‘Message’ type and are unicasted (rather than broadcasted) are considered reliable and therefore utilize the global timeout (see Reliability Section 5.2.2). A countdown to the global timeout is set off. When the deadline is reached, we check whether an appropriate ACK datagram has already been received. If it has not, a global timeout event notifies the application layer of this.

Broadcasted datagrams, in contrast, do not initiate the countdown. This is the case because when broadcasting, one does not wish to get ACKs from every single node in the ad hoc network.

5.2.4. Incoming Datagram Handling
When receiving a datagram from a neighbor, the following logic is applied to decide what should be done with the datagram.

1. Never seen before datagrams (As decided by the triplet, see Section 2.1) are saved to the DB’s datagram table (Section 5.1). The initial state is decided: if the datagram is meant for this device, zero flood attempts are given, classifying the datagram as non-pending (Section 2.3.2). If the datagram is meant for another device, the next local timeout is set to now (causing the router to handle this datagram) and the number of flood attempts is set to the default number.

2. On receiving a previously known datagram (again by the triplet, see Section 2.1), the possible conflict is resolved by keeping the higher TTL of the two versions. The other fields are taken from the new incoming datagram but are presumed unchanged. The metadata (flood attempts, next timeout) is also left unchanged.

3. A record of any datagram sent to us, previously known or not, is kept in the DB’s transaction table (see Section 5.1). This table enables us to know which neighbor has what (to the best of this device’s knowledge).

4. In any case where the datagram is not at its final destination (i.e. it is meant for another device or has been broadcasted) the router is invoked (invoking described in Section 5.2.3). This is done even if the device already knows the datagram, for the following reason: It is possible that the router stopped handling a previously known copy of the datagram due to the TTL being exhausted. In such a case, receiving a new instance of the known datagram with a higher TTL means the router should resume handling it.

5. If the datagram is not previously known and is relevant to the application layer of this device (unicasted for this device’s ID or broadcasted), the application layer is notified of this event. This includes notifying the application layer of ACK and Control datagrams.

6. Finally, an incoming reliable datagram (‘data’ type that was unicasted rather than broadcasted) triggers the creation and unicasting of an appropriate ACK datagram. We
fully expect to receive several copies of the original datagram. Therefore it follows that an ACK datagram should be created only once, upon the first sighting of the original ‘data’ datagram. The reader should note that if the original sender does not receive the destination’s ACK he may choose to resend the datagram with a new timestamp. In such a case it will be treated as an entirely new datagram that will prompt an entirely new ACK. See Section 5.2.2 (Reliability) for more details.

A summary of this logic is described in the following table:

<table>
<thead>
<tr>
<th>Previously known?</th>
<th>Destination</th>
<th>Type</th>
<th>Datagram Table</th>
<th>Transaction Table</th>
<th>Invoke Router</th>
<th>Notify Listeners</th>
<th>Generate ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>This Device</td>
<td>Ack</td>
<td>Update</td>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>This Device</td>
<td>Control</td>
<td>Update</td>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>This Device</td>
<td>Data</td>
<td>Update</td>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Another Device</td>
<td>Ack</td>
<td>Update</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Another Device</td>
<td>Control</td>
<td>Update</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Another Device</td>
<td>Data</td>
<td>Update</td>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Broadcast</td>
<td>Ack</td>
<td>---</td>
<td>Undefined</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Broadcast</td>
<td>Control</td>
<td>Update</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Broadcast</td>
<td>Data</td>
<td>Update</td>
<td>Insert</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>This Device</td>
<td>Ack</td>
<td>Insert</td>
<td>Insert</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>This Device</td>
<td>Control</td>
<td>Insert</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>This Device</td>
<td>Data</td>
<td>Insert</td>
<td>Insert</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Another Device</td>
<td>Ack</td>
<td>Insert</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Another Device</td>
<td>Control</td>
<td>Insert</td>
<td>Insert</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Another Device</td>
<td>Data</td>
<td>Insert</td>
<td>Insert</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Broadcast</td>
<td>Ack</td>
<td>---</td>
<td>Undefined</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Broadcast</td>
<td>Control</td>
<td>Insert</td>
<td>Insert</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Broadcast</td>
<td>Data</td>
<td>Insert</td>
<td>Insert</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 - Received datagram decision table

5.2.5. Router Hibernation
The DataFlooder offers a method that switches the router ‘on’ or ‘off’. Switching the router off means that the router service (Section 0) is shut down, and that it will no longer be invoked even if there are timed-out pending datagrams.

While in hibernation, no datagrams will be sent to any neighbors. However, incoming datagrams will still be received and queued for forwarding. In addition, the different counters for the global

* In practice, the datagram will be updated/inserted to the datagram and transaction tables. The router will be invoked and will flood the datagram forward. The listeners will be invoked and they will ignore this event. In any case, such a datagram is never generated by our code.
timeout event will still run, and global timeout events will still be raised. When switching the router back on, the router service is immediately invoked and will handle all the pending datagrams as per usual.

The purpose of this switch is to enable the user to save battery in times when the user knows no new neighbors are about to show up. It is especially suited for mobile nodes which travel between different connected components (Section 2.3.2). For instance, an ad hoc network may be deployed in a disaster area. Ambulances driving back and forth between the disaster area and a hospital would switch the router off while they are in transit. This will conserve battery, and the router service will not reduce the remaining flood attempts of the datagrams. When arriving at the hospital, the router will be switched on, dumping all the transported datagrams into the connected component.

5.3. DataFloodingRoutingService (Class)

This object is the heart of the Data Flooding protocol implementation, and should be considered a direct implementation of the generally discussed protocol given in Section 2.3.

This object is a Service – an Android construct that may run code in the background, independent of any UI, with a life cycle that may be arbitrarily long (longer then the application’s Activities – which in layman’s terms are the different ‘screens’ of UI).

The implemented protocol’s flowchart is available at the end of this section (see Figure 14 - Flowchart of Data Flooding Routing Protocol)

5.3.1. Practical Considerations

In implementing the data flooding router in our Android / Bluetooth environment, and in order to minimize response times and battery usage, the following considerations were taken into account:

1. The Bluetooth discovery method, which reveals the device’s neighbors, is very costly. It uses up a lot of battery and takes 12 seconds during which no new Bluetooth connections can be made (Data can still be sent over the preexisting connections). Therefore, as few discovery calls as possible should be made. In addition, some minimal time between discoveries must be allowed for incoming new Bluetooth connections. See Section 0.

2. An ‘Urgent datagram first’ policy prevents starvation of old datagrams while ensuring progress, as previously mentioned in 2.3. However, once a connection is made with a neighbor it is more efficient to send it all the relevant datagrams in bulk. A combined policy is employed. See Section 5.3.5.

3. When performing the Bluetooth discovery, only a limited amount of information is known about each neighbor. Filtering out neighbors which are not smart phones is easy. However we cannot tell whether a device has our application installed. When trying to connect to a device which is not running our application, several seconds may pass
before the Bluetooth API reports a failure (when the server socket does not respond). See Section 0.

4. The router runs on a separate thread, which introduces concurrency issues: data such as the list of current neighbors or the current outgoing datagram queue needs to be provided and queried by this and other threads. When the router is finished, we must avoid a race scenario between this shut down sequence and possible newly incoming datagrams. All of these issues have to be dealt with proper locking mechanisms. See Section 5.3.3.

5. The ACK and global timeout reliability mechanism enables some optimization (no need to keep flooding a datagram if the device has already heard its ACK pass by). See Section 5.3.2.

6. ‘Hello’ control datagrams are sent periodically over the basic protocol (see HelloService 6.3). This enables some optimization (no need to flood a Hello datagram that has already been superseded by a new one). See Section 5.3.2.

7. The need to unit test the router dictates the incorporation of built-in testing facilities – such as allowing the thread running the test case to wait until the router is done. See Section 5.3.3.

5.3.2. Optimizing Amount of Pending Datagrams
In practice, the criteria for selecting the pending datagrams to be handled by the router are now stricter (as compared to Section 2.3.2, new filters in bold). Pending datagrams:

1. Are not meant for this device (this includes broadcast datagrams).
2. Still have time to live (TTL > 0).
3. Have remaining send attempts.
4. Are not ACKed ‘Message’ type datagrams (see Reliability Section 5.2.2).
5. Are not obsolete ‘Hello’ datagrams (see HelloService Section 6.3).

5.3.3. Timing and Concurrency
The following timing and concurrency mechanisms have been implemented:

1. Parallelism - the router runs on a separate thread that centrally handles all the unicasting and initiates discoveries. This centralized approach enables us to time the discoveries in a way that better utilizes the idle time.
2. The discovery calls are made asynchronously, such that the router can still send datagrams to neighbors with whom a connection already exists. This means that a list of the current neighbors must be accessed synchronously for it to be thread safe: The asynchronous discovery call will update this list while the main router thread is processing it. For this reason the list of current neighbors is synchronized with a mutex.
3. When there are no further pending datagrams, we want the router thread to terminate. The decision to start or stop the thread is done in a synchronized fashion. Once an old router thread has decided to terminate, any new pending datagrams are sure to start a
new thread. On the other hand, no new thread will be started as long as the old one is still at work. This is synchronized by a mutex.

4. If there are still pending datagrams, but they should not be sent yet as they did not reach their local timeout, the router should sleep until the next local timeout occurs. For this purpose, a list of future timeouts is kept.

5. While the router thread sleeps, waiting for the next timeout, new timedout pending datagrams should interrupt the sleep (see invoking the router Section 5.2.3). This has the purpose of enhancing the response times experienced by the user. A wait/notify monitor is employed.

6. A dedicated static function allows test code to wait until the router thread is done. This is accomplished with a monitor that is notified from within the synchronized block mentioned in bullet 3 above.

7. Randomization of some wait times has been considered carefully. This type of mechanism is meant to prevent a timing deadlock that prevents progress (e.g. Alice has data for Bob but always tries to contact him just as he is running discovery). After careful examination of the protocol’s flowchart, this has been deemed unnecessary yet noteworthy (provided that the protocol’s time constants are well thought out).

5.3.4. Discovery Conservation

Our guideline in designing the discovery policy is ‘As few as possible’, while avoiding starvation scenarios. The policy is:

1. ‘Discover on idle’ - Discover only if the routing service has nothing left to do with the currently known neighbors (neighbors returned in the last discovery call), or if there are no known neighbors at the moment. This idle state is reached when all the pending datagrams were successfully sent to all the known neighbors (‘all’ – possibly all zero of them).

2. Starvation avoidance of new neighbors. If we only follow bullet 1, Starvation can occur if the device is constantly busy with new datagrams. It will never discover, and therefore may ignore any new neighbors. To avoid this, we force a discover call if it has been more than some maximum allowed time since the last discovery (this time is defined in the Preferences, see 6.7).

3. Incoming connections starvation avoidance. The following scenario is of interest: Alice has data for Bob. Bob has pending datagrams that are already known by all his neighbors, so he spends all of his ‘idle’ time constantly rediscovering. Alice cannot open a connection with Bob and there is no progress. To avoid this, we ensure a minimal time constant has passed before rediscovering (This time constant is also defined in the Preferences, see 6.7).

As an additional optimization, the Router Service implements the Layer2EventListener (see 3.4) and is notified of any incoming datagrams from neighbors. It uses this information to find out about new neighbors – without calling the discovery method.
The importance of this mechanism is that the receiving device is thus able to make a quick initial response to this new information (the datagram itself as well as the information about a new neighbor) before doing any discovery call.

As an example, if Alice sends a message to Bob, Bob will be able to reply immediately with an ACK. In contrast, without this mechanism Bob would have to do a lengthy discovery process before sending the ACK.

5.3.5. Most Urgent Datagram vs. Focusing on a Single Neighbor

Our algorithm begins by selecting the most urgent datagram for processing, out of the set of timedout pending datagrams. With this datagram in mind, a list of neighbors which do not know this datagram is compiled. The router service now goes over this neighbor list and ‘focus’ on each neighbor.

When a neighbor is focused upon, every single relevant timedout pending datagram is sent its way, one after another. This optimization is based on the assumption of spatial and temporal locality: if a single datagram was unicast successfully to a certain neighbor, then unicasting the next one again immediately, before the neighbor or the transmitting device had a chance to change locations, will probably be successful as well.

The assumption is made both ways: If the transmission of the datagram to a certain neighbor has just failed, there is no use in trying again immediately. Therefore we drop the ‘focused’ neighbor and proceed to the next one.

After trying to focus on all the neighbors that were relevant to the most urgent datagram, the router sets the next local timeout and goes on to the new most urgent datagram. By this time it is perfectly likely that this next datagram was already sent to many if not all of the available neighbors on the previous round of ‘focusing’. This simply means that the list of relevant neighbors is shorter or zero. Indeed, we expect that as we progress with the timed-out pending datagram queue, each datagram will require less and less processing.

In conclusion, this mechanism prevents starvation of old datagrams by dismissing the most urgent datagram first. At the same time, this mechanism optimizes the flooding time by giving each and every neighbor a concentrated effort.

5.3.6. Unimplemented Optimizations

Several additional optimizations were discussed but eventually dismissed:

1. When focusing on a particular neighbor (see immediately above, Section 5.3.50) it is possible to send all the relevant datagrams in a single bundle, possibly reducing the overhead of calling layer 2’s `unicast` method again and again. An experiment was run to see whether these consecutive calls to `unicast` amount to any significant delay. The results show that such an optimization offers little improvement (see Section 7.3.2).
2. As mentioned in the practical consideration Section 5.3.1, neighboring Bluetooth devices which are not running this application may cause some delay. This could have been mitigated by keeping a ‘black list’ of neighbors not running the application. Any such responsibility was delegated to and encapsulated within layer 2 (Section 4.5).

3. A second way of mitigating the effect of neighbors which are not running the application is to try to handle all the different neighbors in parallel. On each cycle of the protocol, a thread will be started for each neighbor and will unicast all the relevant datagrams to it. This is possible in principle as the Bluetooth API generally allows multithreading. This optimization was eventually dismissed due to the project’s time constraints.
5.3.7. Full Diagram of Routing Protocol

![Flowchart of Data Flooding Routing Protocol](image)

**Figure 14 - Flowchart of Data Flooding Routing Protocol**
First two decision nodes:
  - The definition of pending datagrams is given in Sections 5.3.2 and 2.3.
  - These nodes separate three different states:
    - No pending datagrams: There nothing left to do – exit.
    - There are pending datagrams but they have not timed out. This means there is nothing to do right now, but there will be soon – Go to ‘Wait / Discover if Idle’ block.
    - There are timed out datagrams that need to be sent – progress downward on diagram.

Wait / discover on idle:
  - The router thread sleeps until the next local timeout, or:
    - If we are in the ‘idle’ state, defined heuristically as having nothing to send to the currently known neighbors -> discover.
    - This block is meant to minimize the impact of discovering by timing it to a more idle timeframe, when the need to make a new Bluetooth connection is less likely to arise. This is in accordance with Section 5.3.3.

Discover if out of date:
  - Before the router starts to actually send the messages, it checks whether the current neighbor list is up to date. This is done by comparing the current time with the time of the last discovery call, in light of an allowed time constant defined in the Preferences (Section 6.7). If the list is out of date, it starts the discovery process asynchronously and proceeds.
  - This block prevents the starvation of old neighbors, in accordance with Section 2.3.1.

With most urgent datagram:
  - The most urgent datagram, whose timeout expired first, is selected. This is in accordance with the mechanism mentioned in 5.3.5, and is meant to prevent the starvation of old datagrams.
  - With respect to this datagram, a list of neighbors which are not known to hold a copy of it is made. This data is available in the database through the DbWrapper (Section 5.1). The router proceeds to go over this list of relevant neighbors, focusing on them (the focusing process is defined in Section 5.3.5).

With focused neighbor:
  - Once a neighbor has been selected for focused handling, the router creates a list of datagrams that the neighbor is not known to hold. This required data is in the transaction table of the DB, accessible through the DbWrapper (Section 5.1).
  - The router loops through this list, unicasting each datagram to the focused neighbor via the generic layer 2 interface.
  - If any of the unicasts fails, we assume that further attempts at unicasting to this neighbor will be more likely to fail as well. For this reason the loop is exited, so
that the next neighbor can be approached. This is also described in Section 5.3.3.

- Once the router is finished with the most urgent datagram, the metadata for that datagram is updated (decrementing the remaining resend attempts, setting the next local timeout), and the algorithm returns to the main stopping condition.
6. Application Layer

6.1. MessageArchive (Abstract Class)

The message archive is an interface used by the application activities to retrieve data in a way independent from the layer 3 and DB implementation. In this way, different implementations of layer 3, which also warrant a different DB implementation, will not require any changes to the code in the application layer. This interface is implemented by the DbWrapper (Section 5.1).

The following methods are required for this interface:

- `getInbox()` returns messages meant for this device (including broadcasts for instance)
- `getAckedOutbox()` returns messages equivalent to a ‘sent items’ folder, meaning outgoing messages that are known to have been sent successfully.
- `getUnackedOutbox()` returns messages that were sent by this device but which may not have reached their destination (i.e. messages for which we did not receive an Ack).

6.2. PhoneUtilities (Static Class)

This static class handles phone number strings with the following methods:

- `isPhoneNumber(string)` indicates whether the string is a phone number by comparing to a regular expression.
- `toCanonicalizedForm(string)` translates the number into a globally single-valued canonicalized form (e.g. 972586884486). This allows truthful comparison between phone numbers.
- `numToName(string)` looks up the phone number in the Android contacts content provider. If found, the display name is returned. If not, the original argument is returned unchanged. This is used by the UI.

6.3. HelloService (Class)

The Hello service runs a thread that broadcasts ‘Hello’ control datagrams periodically. The interval between broadcasts is derived from the global preferences. The service is started with an intent which may or may not contain (as an Intent extra) the amount of cycles that the thread should run. If no such extra is given, a default amount is derived from the global preferences. In this way, the thread will always run for a bound number of cycles, thus circumventing any non-terminating loop that would drain the battery. Starting the service again
with a different intent will override the remaining cycles with the new value. Stopping the service will, naturally, stop the thread.

Starting and stopping the thread is done in a synchronized fashion. This guarantees that different threads starting or stopping the service will not cause two threads to start, for instance.

The static `waitForThread()` method waits on a static monitor until the service is either done (no more cycles) or stopped externally. This enables efficient unit testing.

6.4. AdhocApplication (Class)

The `AdhocApplication` class extends the `Application` class. A reference to the application object is easily obtained from any Android context within the application (all activities and services, for instance). It is primarily used to hold global references to important objects, enabling said objects to communicate with each other. In addition, this class handles various methods that deal with `preferences` (see `Preferences` 6.7) due to their global nature.

The following object references are kept:

- `Layer2Interface` implementation
- `Layer3Interface` implementation
- `MessageArchive` implementation
- `HelloService` instance
- List of the current immediate neighbors (supplied by layer 2).

6.5. Compose (Activity Class)

The compose activity is the main activity of the application. The activity consists of the following components:

- Buttons to open Inbox, Outbox, Debug and Performance activities.
- Code for menu button, to open the menu and `preferences`.
- A list of network contacts: This list contains all contact IDs that our application knows about their existence. That is, any device that sent a datagram that was received or forwarded by our application, during the recent time (configurable). This list is intended to show the user the contacts that are currently “Online”, and allow the user to pick the destination from that list. In addition, the list will always include one entry for a broadcast address (see the contact list in Figure 15 - Compose activity screen).
- A button to open the contact picker. This should be used in case the user wants to choose a contact from his phonebook and not from the “online” user list.
- An edit box to allow the user to enter a number manually.
- An edit box to allow the user to enter the message he would like to send.
6.6. Menu (Class)
The menu is inflated from a simple XML file and is accessible from the compose activity. It has three options:

- **Make me visible**: starts the **HelloService** (Section 6.3).
- **Switch router on/off**: changes the router on/off state (Section 5.2.5).
- **Preferences**: opens the **Preferences** activity (Section 6.7).

While the menu is being inflated, the state of the **HelloService** and the **DataFlooder** is checked in order to display the correct text (make me visible/invisible, switch router on/off).

6.7. Preferences
Android preferences are persistent key-value pairs that may be stored and accessed by the Android API. Preferences are globally available across the application. The preferences are laid out in an XML file that both defines them and forms the basis for the inflation of an activity that manipulates them.

In our XML file, the following preferences were defined (see Section 5.2 (**DataFlooder**), Section 6.3 (**HelloService**) and Section 3.3 (**Datagram**) to better understand the meaning of specific preferences).

- **phone_number**
  The phone number entered by the user on the first run. Used as a global address in the ad hoc network.
- **global_timeout_seconds**
  An unACKed sent message is assumed lost if this much time has elapsed since the unicast transmission.

- **visibility_duration_minutes**
  Switching on the Hello service will last this long.

- **hello_interval_seconds**
  The Hello service broadcasts Hello control datagrams periodically with this interval.

- **online_lookback_time_minutes**
  A global address is labeled online, as in available in the ad hoc network, if it has been active in this timeframe.

- **reflood_time_minutes**
  The amount of time during which a pending datagram will be periodically reprocessed and sent to new neighbors.

- **local_timeout_seconds**
  The time waited between flood attempts. Must be shorter than the next item.

- **min_time_between_discoveries_seconds**
  The router service, if idle, will initialize a Bluetooth discovery call unless this amount of time has not yet elapsed since the last call. Must be longer than the previous item, and preferably longer than the Bluetooth discovery time (12 seconds).

- **max_time_between_discoveries_seconds**
  While the router is busy forwarding datagrams, it will not waste time on discoveries – unless this much time has passed since the last discovery.

- **TTL**
  the default amount of maximum hops given to locally generated datagrams.

The user may enter, through the menu (see menu6.6), a ‘preferences activity’ inflated from the preferences XML. The full XML includes additional XML nodes that organize the above preferences in a meaningful way. Some of these nodes act as buttons that invoke some functionality. This activity enables:

- Editing of all the preferences
- Editing the phone_number preference will be done coherently in the DataFlooder and the DbWrapper as well.
- Restoring the default values (except for the phone number)
- Emptying the inbox/outbox
- Clearing the DB entirely
6.8. Inbox (Activity Class)

The inbox activity displays a list of the messages meant for this device, including broadcasted messages (which are labeled with a “(Broadcast)” suffix, see Figure 17 - Inbox activity screen). Touching a message opens a dialog that enables the user to quickly reply to the message.

6.9. Outbox (Activity Class)

The outbox activity displays the messages sent from this device. These messages may be in the following, clearly labeled states (see Figure 18 - Outbox activity screen):
December 2012

- In transit: The message was sent but no ACK has been received yet. Indicated by a ‘spinner’
- ACKed: The message was ACKed by the intended recipient. Indicated by a green tick.
- Timeout: ACK was not received in time and the message is assumed lost.
- Broadcast: Broadcasted messages are not reliable and have no ACKed/unACKed state. Broadcasted items are labeled with a blue broadcast icon.

Clicking on a timed out message opens a dialog that enables the user to resend a copy of the lost message.

**Figure 18 - Outbox activity screen**

### 6.10. Debug (Activity Class)

The debug activity is meant to allow the programmers to observe the behavior of the data flooding router more carefully. This activity assumes (by explicit casting!) that the application is using the BluetoothWrapper, the DataFlooder and the DbWrapper with its specific schema. It is not meant to be used by the end user and should be removed if the application is to be released.

The debug activity displays pending datagrams as they are being handled by the router. The following data is provided on each datagram:

- From/To/Timestamp
- Type (Message, Ack, Control)
- Time to live
- Remaining flood attempts
- Next local timeout
- Number of neighbors that sent the datagram to the current device or received the datagram from the current device.
- Touching item displaying the datagram opens a dialog showing the payload (content of the message or Ack information, see Section 3.3)

The activity also shows some additional data on the state of the device:

- Number of current immediate neighbors
- Number of open Bluetooth connections
- Is the Bluetooth accepting new connections? (false when discovering)

![Debug activity screen](image)

**Figure 19 - Debug activity screen**

### 6.11. Performance (Activity Class)

This is a modified compose activity designed to enable more efficient performance testing. It unicasts 10 datagrams immediately to one or more recipients with the push of a button, and displays a clear record of the data on the screen.

It also enables the quick clearing of the DB, clearing the current immediate neighbors list and closing open Bluetooth connections (see Figure 20 - Performance activity screen). This eases the recreation of specific scenarios such as ‘cold start’ (see in 1.6.3)
6.12. Splash (Activity Class)

The splash activity handles the startup procedure. It initializes the DbWrapper, BluetoothWrapper and DataFlooder before going on to the main activity (compose). This is done immediately and the user will not register that a transient activity took place.

However, on the first run of the application, this activity will persist and open a dialog prompting the user to enter his phone number (see Figure 21 - Splash activity screen). In this way we assert that we have this crucial parameter before attempting to initialize layer 3, which requires this phone number as a network ID.

In addition, this activity was chosen, as the point from which notifications and toast alerts (see Glossary) will be raised, due to its global nature. The notifications and toasts are raised on layer 3 events: global timeout of a sent message or a new incoming message. Figure 22a presents a toasts for a global timeout (see 5.2.2). Figure 22b describes a toasts for an incoming message (see 5.2.4). Finally, Figure 22c describes notifications for both incoming messages and global timeouts. When the user clicks on a timeout notification, the outbox activity is opened. Clicking an incoming message notification takes the user to the inbox activity.
Figure 21 - Splash activity screen

Figure 22 (a, b and c) - Toasts and Notification screens

7. Development Process
In this section we discuss the general development methodologies we used in the aspects of source control, testing, logging and more. We will discuss different design patterns we used and notable events during development which caused significant changes in our code.
Section 7.1 reviews the development methodologies.
Section 7.2 reviews different design patterns.
Section 7.3 and 7.4 describe notable events concerning layer 2 and layer 3 respectively.

7.1. Development Methodologies
All development was done on eclipse (Indigo 3.7.2), using the 2.3.3 (API 10) Android SDK, with the code being managed in an SVN repository hosted at the Technion NSSL lab.

All classes, methods, and non-trivial variables have Jdoc commentary. On the eclipse IDE (see Glossary), this allows for on the spot method ‘help’, which eased the cooperation between our two man team members. In addition, the Jdoc commentary can be used to automatically generate the full code documentation (available in [6]).

The Android environment has a logging feature, easily used with the Android Log class (see [[5]]). Whenever the programmer wants to print to the log, Log.d() is called with two arguments: a ‘tag’ string and the message itself. The tag is used to differentiate between different contexts in the application. While and after the app/test has run, the programmer has access to this log, which can be filtered by the mentioned tags. We utilized this feature extensively, logging on almost every branch, with the tags differentiating the different classes and different threads. This was obviously of immense help during development and debugging, as other methods of debugging (breakpoints and steps) are not always suitable for a multithread application.

Our main source control tool was SVN ([9]). We tried to divide the work into different units as much as possible, allowing each one of us work independent of the other. When a coding feature or a bug fix has been completed, we committed the relevant files to SVN, sometimes merging when necessary.

All but the most trivial classes were unit tested. In most cases, the test code was written alongside the tested class. These tests correspond to the Unit Testing section in 8.2. Unit testing was done with the JUnit test environment and its Android derived classes. These tests were run before any change was merged into our source control repository (SVN, see [9]).

Layer 2 and layer 3, while composed of several classes, were also tested as units. Since their input/output includes messages to the other layer, each layer has a ‘stub’ implementation, meant for testing only. For instance, unit testing of layer 3 is done by having it send datagrams to a layer 2 stub, which has no RF abilities of any kind. The test code can later query this stub to see that it received the proper input.

The application uses the Android ‘preferences’ mechanism extensively. However, since the default time constants have a decisive effect on the performance of our application, and since entirely different values must be used when unit testing, we decided to keep all the default values in a special XML file. This file enabled us to tweak the different constants easily, as well as to share or replace them. This also allowed the unit tests to be run quickly, with short time constants, without having to manually change any constants.
We avoided any premature optimizations whenever possible. The code was written in the most readable and easily maintainable fashion at first, and performance issues were handled as they were encountered. This included a complete rewriting of the routing protocol implementation in the most extreme case (See layer 3 milestones in 7.4).

Final tweaks were made to aid performance by running method traces (with the tools provided in the SDK), and watching for the few methods that were responsible for most of the load.

### 7.2. Design Patterns

We used several design patterns in our project listed in the following.

#### 7.2.1. Observer

Both layer 2 and layer 3 interfaces implement the observer design pattern. In the layer 2 interface, the observer design pattern is used to notify of an incoming datagram (to any possible destination). The listener of this notification is layer 3. In the layer 3 interface, the observer design pattern is used to notify of incoming datagram which are destined to the receiving device (may be broadcast as well). The listeners of this notification are the different activities (Inbox, Outbox and Splash, for toasts and notification).

#### 7.2.2. Concurrency Design Patterns

In several places we used the Lock and Monitor object design patterns to enforce safe access to different data structures from different threads. For example:

- The DbWrapper class is a full monitor, meaning – all of its methods are synchronized and cannot be called concurrently.
- The BluetoothWrapper class synchronized some of its methods (for example: discover and makeDeviceConnection) to enforce the requirement of the Bluetooth protocol that new connections cannot be established while discovery is active.
- In the Compose activity, the contacts list adapter may be accessed from different threads: One thread is responsible for determining which contacts are checked in the list when sending a datagram, and another thread is responsible for updating the contact list in case a new contact is discovered or a contact becomes “too old”. This is resolved by a lock.
- The asynchronous discovery locks and updates the list of current neighbors, which is also synchronously accessed by the router thread.
- The router thread and Hello thread both have a static method that enables the test code to wait for them to finish. When they finish, they notify the waiting thread which is waiting on a monitor.
- Many more examples exist in the code.
7.2.3. Singleton
There are two classes implemented as singletons: The DbWrapper that encapsulates all accesses to the data base. The concurrency issues in accessing the DB are solved by implementing the DBWrapper as a monitor, but this solution would have been in vain if there were 2 different instances of that class! (both monitors, but not synchronized with one another). Therefore, the singleton design pattern enforces the existence of a single instance of the DbWrapper.

The DeviceManager is implemented as singleton. This is used to prevent us from opening multiple connections to the same device.

7.3. Layer 2 Milestones

7.3.1. Problems Encountered With 2.3.3 Devices
The development and testing of the BluetoothWrapper started using two devices with Android 4.0.3 installed (personal devices). The initial version of the BluetoothWrapper was written, using the Android 10 API (compatible with Android 2.3.3 or later version) and it passed the tests. When performing tests with multiple devices, we also included two devices with Android 2.3.3. While testing the application on these devices, we encountered some new problems unseen in the 4.03 devices. Some of the problems resulted in code modification, but there were also two major problems that made the application practically unusable on these devices, although the API level should have been compatible.

- The first major problem concerns the Bluetooth visibility issue. Our routing mechanism relies on the fact that devices in the network are discoverable via Bluetooth frequently enough. A Bluetooth device must be discoverable in order to accept connections from other devices. The Android Bluetooth API supports two options to ask the user for making a device discoverable. The first option is to specify a time limit for the discovery (i.e. a maximum amount of time during which the device will be discoverable), which can be any value from 1 to 300 seconds. After this period of time, the device will return to be undiscoverable. This means that in order to maintain the device in a discoverable state, the application must prompt a request to the user, as often as every 5 minutes, which does not fit the nature of the application (and the requirement in 1.6.2). The second option is to ask the user to make the device discoverable without specifying a time limit. In this case, the limit will be taken from the default value set in the Android phone. The 4.0.3 devices have an option to set the default to “never timeout”, allowing a device to be discoverable until further notice, whereas the 2.3.3 version’s limit (for the default value) was 5 minutes. Therefore, using this option with the 2.3.3 devices yields the same problems we discussed with the time limit.

- The second major problem was the unexpected behavior of the insecure connection in the Android 2.3.3 devices. A Bluetooth insecure connection is a connection established without prompting the user with a pairing request (see requirement 20 in 1.6.2). The
Android API 10 (for 2.3.3 and above) supports this feature, according to the Android development guide [5]. Nevertheless, when testing with the 2.3.3 devices, a pairing request was constantly prompted when the devices were not paired. This did not occur with more advanced devices we tested (different 4.0.X and one 2.3.6). Searching the web for the problem, we found evidence of other developers encountering the same problem with 2.3.3 devices. Just like the prior problem, this is significant because our usage assumption was that the application may run in the background without interrupting the user.

Due to these problems, we decided that the application will be used only on more advanced Android operating systems.

The other minor problems we encountered and resulted in a change in our code:

- The `BluetoothSocket connect` method did not always return (neither with a successful connection nor with a thrown exception in case of an error). This was solved by adding the timeout thread, described in 4.10.
- When the connection was disrupted (by closing one of the devices, for example), the device on the other side did not respond (at least not for a while). Thus, the device still thought it holds a valid connection to a turned off device. This was solved by adding the ping thread, described in 4.9.
- The command `listenUsingInsecureRfcommWithServiceRecord` was failing without any visible reason. We preformed this test on the BluetoothChat example – a sample code provided by the Android development guide explaining how to work with Bluetooth (see sample in [5]). As the problem occurred in the sample code as well, we assume there is a problem with the device and not with our code. This was solved by adding the watcher thread described in 4.7. The thread checks whether the listen method was successful and retries if not.
- After adding the ping command, we encountered write requests from different threads. This revealed a bug we did not encounter in the 4.0.3 test. The output stream had the datagram data and ping data joined together. In the receiving side, all of the input buffer’s content was read and deserialized as one datagram, which caused an error. This was solved by sending the full size of the datagram (in a constant field: 4 bytes) before sending the datagram.

**7.3.2. Data Bundling Experiment**

We have noticed that a common use case in our routing protocol is when a certain node transmits several datagrams to another node one after the other. We considered an optimization for this scenario: sending a single object aggregating the datagrams, instead of sending the datagrams one by one. The consideration against this optimization was that it may actually increase overhead instead of decrease it. In the common case, the overhead for opening the connection affects at most once – for the first datagram. Therefore, in the stable state, the datagrams are passed one after the other using an open connection. Creating one
object to wrap them will result in more overhead for each such object, and will not improve the application’s performance. We decided to do a performance test to see whether the multiple send calls are causing a slowdown, and after seeing that consecutive messages have hardly any delay between them (as compared with our performance goals), we decided to abandon this optimization.

7.4. Layer 3 Milestones

Layer 3 was successfully implemented and functionally tested with the layer 2 stub (see 8.1). When it was integrated with the Bluetooth implementation of layer 2 the functionality was intact, but it was immediately clear that the performance was not satisfactory (All requirements described as warm in 1.6.3 did not pass).

Upon inspection, it became apparent that the root cause was the layer 2 discover method being called too often. This not only stopped the router for 12 seconds, but also made the device unavailable for new incoming connections from other devices.

For this reason, the entire routing code (DataFloodingRouterService) was rewritten with the main intention of reducing the number of discovery calls made.

This showed a noticeable markup in performance, however it was still limited. Even with two neighboring devices which have an open Bluetooth connection (a state that can be checked in the debug activity), unicasting would often take over 10 seconds, with the ACK often taking another 10 seconds to return.

This was obviously caused by a lack of parallelism. While the router was waiting for layer 2’s discovery to return, no unicasts were attempted. This is despite the fact that the Bluetooth API enables such concurrency. We therefore chose to take advantage of this concurrency by modifying both layers. The router now calls discover asynchronously, while still trying to flood the datagrams to the previously known neighbors. Our final improved design is described in 5.3.7.

8. Testing

This section describes the different tests we performed while testing the application and its components.

- Section 8.1 describes the “Layer 2 stub” – a special class written for the purpose of testing, simulating the behavior of layer 2.
- Section 8.2 describes the unit tests of the different components of the application.
- Section 8.3 describes the integration testing between layer 3 and the application layer.
- Section 8.4 describes the integration functional testing of the entire application.
- Section 8.5 describes performance testing performed with three devices.
Section 8.6 describes a test performed on six devices, allowing our friends to play and enjoy the application.

8.1. Layer2Stub (Class)
Many of the unit tests were run on a single device – often emulated – with no way to communicate via Bluetooth. The Layer2Stub was created to allow testing of the entire layered stack except for the RF part.

This stub allowed not only functional testing of layer 3 and above, but also enabled development and testing of layer 2 and 3 in a separate and parallel manner.

This stub includes the following features:

- Full implementation of Layer2Interface
- A list of layer 2 neighbors that can be updated or retrieved (by discover).
- Datagrams unicasted to the stub are kept in a list along with the target neighbor. This list can also be queried to assert certain datagrams were sent.
- The stub can be switched to a ‘broken’ state. Datagrams sent in this state will reach a separate list that can also be queried to assert failure.
- It is possible to simulate an incoming datagram from a specified neighbor.
- It is possible for the test code to wait on the stub until a specified datagram is unicasted to the stub. This allows the test code to wait until the router thread is done with certain tasks before proceeding with the scenario.

8.2. Unit Testing
Nearly all unit tests were done using the JUnit java unit testing framework and its Android specific derivatives. This means a dedicated test project, constructed of test classes that extend the TestCase class. More precisely, we extend the Android specific AndroidTestCase and ServiceTestCase.

For each of the following test cases, a concise table describes the tested scenarios. In most cases the scenario name is equivalent to the method(s) being tested.

8.2.1. DatagramTest

<table>
<thead>
<tr>
<th>testEquals</th>
<th>Create three datagrams:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Normal</td>
</tr>
<tr>
<td></td>
<td>• Exact copy</td>
</tr>
<tr>
<td></td>
<td>• Same triplet, different on other fields</td>
</tr>
<tr>
<td>Assert that normal.equals() both due to the triplet. However normal.exactMatch() only with the exact copy.</td>
<td></td>
</tr>
<tr>
<td>testCollections</td>
<td>• Prepare two different datagrams with the same triplet.</td>
</tr>
<tr>
<td></td>
<td>• Insert one into a java collection (List). Assert it can be found with the other datagram.</td>
</tr>
</tbody>
</table>
- Insert both into a java set. Only one item should be kept in the set due to matching triplet

**testSerialization**
- Create datagram.
- Serialize it into a byte array.
- Deserialize byte array into a new datagrams.
- Datagrams should be exact matches.

**testAckMechanism**
- Create a datagram and a corresponding ACK datagram using the getAckString() method
- Try to getAckedTimestamp() from the original datagram, which is of a wrong type (data instead of ACK). Should throw exception.
- getAckedTimestamp() from ACK datagram. It should match the timestamp of the original datagram.

**testIsAckedBy**
- Prepare a datagram (‘Original’), a correct ACK datagram, and several bad ACK datagrams with corrupted fields.
- Original.isAckedBy(possibleAck) should return true only for the correct ACK datagram.

### 8.2.2. AdhocDBTest

<table>
<thead>
<tr>
<th>SetUp (before each test)</th>
<th>Get DB instance and clear all tables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>testConstructor</strong></td>
<td>Set up the database parameters (Layer 3 ID and Android Context) and get the newly created singleton copy.</td>
</tr>
<tr>
<td></td>
<td>The returned singleton should not be null.</td>
</tr>
</tbody>
</table>

**testUpdateOrInsertDatagram**
- Call updateOrInsertDatagramAndTransaction with:
  - Null neighbor string
  - Empty neighbor string
  - Null source string
  - Should throw exception
- Insert datagram for first time, returns a database ID.
- Call function again with same datagram but a different neighbor. The returned database ID should match the previous function call, as this is the same datagram.
- Insert another datagram (another triplet). A different database ID should return.

**testFindAndUpdateMechanism**
- findExistingCopy() in the empty DB with an arbitrary reference datagram. Should return null.
- Find again with null datagram. Should throw exception
- Insert datagram to DB
- Find in DB using exact match as reference. Should succeed.
- Find using reference with same triplet. Should succeed.
- Update the existing datagram, using a similar
<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
</tr>
</thead>
</table>
| **testSetNextTimeout** | - Try on empty DB or with null parameters. Should throw exception.  
- Insert datagram with initial `nextTimeout` set to be in one second, and with one send attempt.  
- Get the pending datagram queue. It should be empty.  
- `setNextTimeout()` to 10 ms ago. Pending datagram queue should contain the datagram.  
- `setNextTimeout` and decrease send attempts.  
- Pending datagram queue should again be empty. |
| **testInboxOutbox** | - Create some datagrams without inserting to DB.  
- Get inbox, `ackedOutbox` and `unackedOutbox`. All should be empty.  
- Insert one outgoing datagram to DB. Should appear in the returned `unackedOutbox`.  
- Insert ACK on original outgoing. Should no longer appear in `unackedOutbox`, but will appear in the `ackedOutbox`. |
| **testIsMessageAcked** | - Call `isMessageAcked()` with null, badly typed, or correct type on empty DB. Exceptions should be thrown.  
- Insert datagram  
- Calling `isMessageAcked()` should return false.  
- ACK the message  
- Should now return true. |
| **testGetPendingDatagrams** | - `getPendingDatagrams()` should return empty list.  
- Insert datagram that is meant for this device. Calling again should still return empty list.  
- Set `nextTimeout` to the past. It still should not appear as pending because it is meant for this device.  
- Insert a datagram that should be forwarded, with timeout in the future. Should appear as pending but not timedout.  
- Update and zero the remaining flood attempts. Set timeout to the past. Should not appear as pending at all.  
- Update with some flood attempts, but with `timeout==zero`. Should not appear as pending at all (this is a special ‘signal’ to mark a datagram as irrelevant).  
- Change timeout to the past. Should now appear as... |
| testGetFamiliarNeighbors | • Try on empty db. Returned list is empty.  
• Insert datagram with a neighbor. Returned list should contain it.  
• Update with second neighbor. Both should be returned.  
• Insert a different datagram with a different neighbor. Original neighbor list should be unchanged. List for the new datagram should have one neighbor. |
|-------------------------|---------------------------------------------------------------------------------------------------------------|
| testUpdateOrInsertTransaction | • Call method several times with bad arguments: null neighbor, empty neighbor string, null datagram. Should throw exception.  
• Call on a datagram that has not yet been inserted. Should throw exception.  
• Insert datagram with one neighbor.  
• Add transaction with same datagram, different neighbor.  
• getFamiliarNeighbors() should return list with both neighbors. |
| testGetGloballyKnownDevices | • getGloballyKnownDevices() on empty DB. Should return empty list.  
• Insert one datagram. It’s source, but not dest, should be returned in the method. |
| testDeleteOldStuff | • Insert two datagrams, one with a timestamp of over one year ago, the other with a current timestamp. Both should be timed out under normal conditions.  
• getPendingDatagrams() will only return one datagram. The old one was deleted by the automatic cleanup routine. |
| testGetMetaData | • getNextTimeout and getRemainingAttempts. Should throw exception when called with null datagram, or with a datagram that was not inserted to DB.  
• Insert datagram to DB, with a certain nextTimeout and remainingAttempts.  
• Successfully return these values with the methods. |

### 8.2.3. DataFloodingTest

**setup (before each test)**

- Change all preference to the predefined test values
- Get DB instance and clear it, set myID to “me”
- Initialize Layer2Stub (default with 3 simulated neighbors: “1”, “2”, “3”)
- Initialize `DataFlooder` with the stub and myID “me”
- Initialize `Layer3EventIndicator` and register it as listener to the `DataFlooders` event
- Wait until the router thread is done (should happen almost immediately as it has nothing to do)

<table>
<thead>
<tr>
<th>teardown (After each test)</th>
<th>After each test, restore the preferences to the default values</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>testConstructor</th>
<th>Do nothing. Verify that the setup and tear down works. This tests all the relevant constructors.</th>
</tr>
</thead>
</table>

| testUnicastBadArgs | Call the unicast method with bad datagram as argument: null, spoofed source, negative number of flood attempts. Should throw exceptions.  
Wait for router (should not be on anyway)  
Assert nothing was sent. |
|-------------------|---------------------------------------------------------------------|

| testUnicast | Unicast a datagram to an arbitrary node which is not one of the simulated neighbors  
Wait for router to finish  
Assert via `Layer2Stub` that the datagram was sent to all 3 simulated neighbors  
Assert the sent datagrams had reduced TTL  
Unicast, set destination as one of the neighbors  
Should be sent via the `Layer2Stub` to all neighbors (flooding)  
Call unicast repeatedly with the same datagram. It should only be sent once (to each neighbor)  
Unicast with TTL=0. Should not reach any neighbors.  
Unicast with many send attempts.  
Wait for datagram to be sent to the first neighbor.  
Add a new neighbor to the `Layer2Stub`.  
Wait for router to finish  
Assert datagram was sent to all four neighbors. |
|----------------|---------------------------------------------------------------------|

| testIncomingNoEvent | Simulate incoming datagram with a global destination that is not this device nor a neighbor  
Datagram should be flooded to every neighbor except the originator.  
Simulate incoming datagram with one of the neighbors as the global destination  
Should be flooded to every neighbor except the originator  
Simulate incoming datagram with one of the neighbors as the global destination  
Wait for the datagram to reach the first neighbor  
Add a neighbor and wait for router to finish.  
Datagram should have reached all four neighbors  
Simulate incoming with TTL=0. Should not be forwarded |
|-------------------|-------------------------------------------------------------------------------------------------|

<table>
<thead>
<tr>
<th>testIncomingWithEvent</th>
<th>Simulate incoming ACK datagram meant for this device.</th>
</tr>
</thead>
</table>
- Incoming event should be raised, containing the ACK datagram.
- Simulate incoming message datagram, meant for the tested device, from a non-neighboring node.
- Incoming event should be raised with that message.
- Wait for router to finish
- An ACK datagram should have been generated and sent to all neighbors
- Simulate incoming message datagram, meant for the tested device, from a neighboring node.
- Incoming event should be raised with that message.
- Wait for router to finish
- An ACK datagram should have been generated and sent to all neighbors

<table>
<thead>
<tr>
<th>testTTLOverride</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulate incoming datagram with TTL=zero</td>
</tr>
<tr>
<td>Wait for router</td>
</tr>
<tr>
<td>Assert datagram was not flooded.</td>
</tr>
<tr>
<td>Simulate another incoming copy of the datagram, with TTL=3.</td>
</tr>
<tr>
<td>Wait for router</td>
</tr>
<tr>
<td>Assert the router flooded the datagram to the neighbors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>testAckMeantForMe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicast datagram with 20 flood attempts. Wait for it to reach the first neighbor. Add another simulated neighbor to the Layer 2 Stub.</td>
</tr>
<tr>
<td>Simulate incoming irrelevant ACK that does not match the previously unicast datagram. The ACK should be meant for the tested device.</td>
</tr>
<tr>
<td>It should raise an event but should not affect the original datagram, which should still be flooded. The original datagram will reach the newly added neighbor eventually.</td>
</tr>
<tr>
<td>The irrelevant ACK is meant for this device and therefore not forwarded further.</td>
</tr>
<tr>
<td>Add a fifth neighbor, and quickly simulate a true ACK.</td>
</tr>
<tr>
<td>The original datagram will not be forwarded to the fifth neighbor due to the ACK making it irrelevant.</td>
</tr>
<tr>
<td>The ACK itself is meant for this device and therefore not forwarded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>testAckMeantForOther</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulate incoming datagram that should be forwarded</td>
</tr>
<tr>
<td>Wait for it to reach the first neighbor, then add another neighbor</td>
</tr>
<tr>
<td>Simulate incoming irrelevant ACK (not meant for the device under test and does not match the previous datagram). It should not stop the original from reaching the new neighbor. The ACK itself should be flooded.</td>
</tr>
<tr>
<td>Add a fifth neighbor and simulate a true ACK.</td>
</tr>
</tbody>
</table>
- Original should not reach the fifth neighbor due to the ACK making it irrelevant.
- The true ACK itself should be flooded forward.

**testDiscover**
- Perform L3.discover() on empty DB. Should return an empty list
- Broadcast a Hello datagram. Wait for router to finish.
- Still discover() returns empty list
- Unicast a datagram. The destination may not exist, so it is not taken into account as an active neighbor: discover() still returns empty list
- Simulate incoming from global source “5”. “5” should also appear in list returned from discover()
- Simulate incoming Hello from “6” (destination is broadcast). “6” should also return from discover()

**testManyUnicast**
- Unicast four consecutive datagrams to four different neighbors.
- Wait until **first** datagram reaches all neighbors.
- Assert that by this time, **all** datagrams reached the second to last neighbor (known from Layer2Stub log). This verifies that the ‘neighbor focusing’ works

**testManyIncoming**
- Simulate several incoming datagrams, and simulate ACK for the last one.
- The ACKed message should not reach all the neighbors.
- The rest of the datagrams should reach all the neighbors except for the neighbor from which they have come.

**testAckInMiddle**
- Simulate 10000 neighbors on the Layer2Stub
- Unicast a message from the device under test
- Wait until it reaches the first neighbor
- Simulate incoming ACK
- Wait for router to finish
- Assert the original message reached some but not all neighbors
- Comment: It is possible but unlikely that the router will finish taking care of the 10000 neighbors before a context switch occurs to the main thread that simulates the ACK. In such a case the test will fail (This was a problem initially, but was solved by increasing the amount of neighbors to 10000).

**testAckInMiddle2**
- Unicast and simulate incoming messages
- Simulate an ACK datagram to ACK one of the messages. While not synchronized, this most likely happens before the router had a chance to flood the ACKed message.
- Wait for router to finish
- Assert the ACKed message never got flooded at all since it has become irrelevant.

**testBroadcast**
- Reminder: the broadcast() method simply overwrites the
destination field of the datagram with a predefined broadcast address.

- Unicast to the literal broadcast address. Should be flooded
- Broadcast() a datagram with an arbitrary global destination. Global destination should be overridden with the proper broadcast address. Should be flooded.
- Simulate incoming broadcast. Should raise an event and be flooded forward.

| testOnOffSwitch | • Switch router off, then unicast.
|                | • Wait for router to finish. Assert unicastdatagram never reached the Layer2Stub.
|                | • Switch back on, wait for router, assert datagram got flooded.
|                | • Unicast again. Wait for datagram to reach first neighbor.
|                | • Stop the router and add a neighbor.
|                | • The datagram will not be sent to the new neighbor until the router is turned back on. Wait and assert this.
|                | • Turn on router and assert that the datagram reaches the new neighbor.

| testBrokenLayer2 | • Switch Layer2Stub to ‘broken’ state.
|                 | • Try to unicast. Wait until datagram reaches the Layer2Stub’s ‘failed datagram’ list.
|                 | • Fix Layer2Stub and wait for router to finish
|                 | • After the local timeout, datagram is flooded as normal.

| testGlobalTimeoutEvent | • Unicast datagram
|                        | • Wait until router is finished
|                        | • Wait until a global timeout event reaches the Layer3EventIndicator.
|                        | • Assert the event is of correct type and carries the correct (timedout) datagram.

| repeatTest | Run all of the test cases 10 times. This can possibly catch some multithreading issues, however there is no guarantee.

### 8.2.4. PhoneUtilitiesTest

| testIsPhoneNumber | • Run isPhoneNumber() on several good numbers with different formatting. Should return true.
|                  | • Run on several bad numbers (with letters, etc). Should return false.

| testCanonization | • Canonize a list of numbers that refer to the same number.
|                 | • Assert that they are all equal to the canonized version

| testTranslation | • ‘Translate’ several un-canonized numbers to the display name saved in the device’s address book (which should contain “OmriDor” as 97254688486).
|                | • All variations of that number should return “OmriDor”
8.2.5. HelloServiceTest

| setup (before each test) | • Change all preference to the predefined test values  
  • Get DB instance and clear it, set myID to “me”  
  • Initialize Layer2Stub (default with 3 simulated neighbors: “1”, “2”, “3”)  
  • Initialize DataFlooder with the stub and myID “me”  
  • Wait until the router thread is done (should happen almost immediately as it has nothing to do) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>teardown (After each test)</td>
<td>• After each test, restore the preferences to the default values</td>
</tr>
</tbody>
</table>
| test | • Start service with extra parameter “CyclesToGo” = 0. Wait for router to finish. Assert in Layer2Stub that nothing was sent.  
  • Start service with 1 cycle to go. Wait for router. Assert Hello was sent to all neighbors.  
  • Start service with 3 cycles to go. Assert all three datagrams reached all three neighbors. |
| testConflicts | • Start the service many times consecutively with default settings.  
  • Start it with 50 cycles to go.  
  • Start it with 1 cycle to go.  
  • The last invocation should supersede the others. Several datagrams might be sent, but nowhere near 50. We Assert less than 10 were sent. |

8.2.6. Bluetooth Test Activities

These tests were performed using a special activity written for the unit test of the Bluetooth classes. The activity allows discovering neighbors and sending a datagram to a specific MAC address. In addition, it displays some status information (accepting/discovering, number of connections).

<table>
<thead>
<tr>
<th>Setup test</th>
<th>• Make sure the initial state of the Bluetooth is “accepting”</th>
</tr>
</thead>
</table>
| Discovery test | • Start discovery  
  • Assert that a toast message is displayed saying discovery had started.  
  • Assert that the status is changed to “not accepting”.  
  • Assert that a toast message is displayed saying discovery had ended.  
  • Assert that the status is changed back to “accepting”.  
  • Assert that the MAC addresses of near-by devices appear |
<table>
<thead>
<tr>
<th>Activity Type</th>
<th>Steps</th>
</tr>
</thead>
</table>
| Basic Send    | • Discover neighbors on one device  
• Send a message to a discovered device  
• Assert that the message appears in the second device |
| Several sends | • Send a short message (“hello”)  
• Send a longer message (“Hello. This is Android”)  
• Send a long message (“Hello. This is a test for Android ad hoc network development. This message is the longest out of three. I hope all of them arrive”)  
• Assert that all messages arrived |
| Basic conversation | • Call discover on 2 devices  
• Send a message from one device to the other  
• Send a message from the second device to the first  
• Assert that both message appear |
| Advanced Conversation | • Call discover on 2 devices  
• Send a message from one device to the other  
• Send several messages from the second device to the first.  
• Try different combination of: X message from the first to the second than Y messages from the second to the first and repeat several time with different X,Y |
| Simultaneous connection | • Call discover on 2 devices  
• Try sending a message from both devices (to the other) simultaneously.  
• Assert that both messages arrive |
| Send/Receive during discovery for existing outgoing connection | • Run the application on 2 devices.  
• Send a message from A to B (to create connection)  
• Start discovery on **device A**  
• During discovery: send a message from A to B and from B to A  
• Assert that all 3 messages arrive before the discovery ends |
| Send/Receive during discovery for existing incoming connection | • Run the application on 2 devices.  
• Send a message from A to B (to create connection)  
• Start discovery on **device B**  
• During discovery: send a message from A to B and from B to A  
• Assert that all 3 messages arrive before the discovery ends |
| Send/Receive after discovery for existing outgoing connection | • Run the application on 2 devices.  
• Send a message from A to B (to create connection)  
• Start discovery on **device A**, and wait for it to end  
• Assert that number of connections on both devices has not changed  
• send a message from A to B and from B to A  
• Assert that all 3 messages arrive |
| Send/Receive after discovery for existing incoming connection | • Run the application on 2 devices.  
• Send a message from A to B (to create connection)  
• Start discovery on **device B**, and wait for it to end  
• Assert that number of connections on both devices has not changed  
• Send a message from A to B and from B to A  
• Assert that all 3 messages arrive |
|---|---|
| Client reconnection after **client** crash | • Run the application on 2 devices.  
• Send a message from A to B  
• Kill and restart the application on device A  
• Assert that the number of connections in B decreases  
• Send a message from A to B  
• Assert that the number of connections in B increases back and that the message arrives |
| Client reconnection after **server** crash | • Run the application on 2 devices.  
• Send a message from A to B  
• Kill and restart the application on device B  
• Assert that the number of connections in A decreases  
• Send a message from A to B  
• Assert that the number of connections in A increases back and that the message arrives |
| Client-server switch after **client** crash | • Run the application on 2 devices.  
• Send a message from A to B  
• Kill and restart the application on device A  
• Assert that the number of connections in B decreases  
• Send a message from B to A  
• Assert that the number of connections in B increases back and that the message arrives to A |
| Client-server switch after **server** crash | • Run the application on 2 devices.  
• Send a message from A to B  
• Kill and restart the application on device B  
• Assert that the number of connections in A decreases  
• Send a message from B to A  
• Assert that the number of connections in A increases back and that the message arrives to A |
| Send after send failed | • Run the application on 2 devices.  
• Send a message from A to B (should arrive)  
• Kill application on B  
• Send a message from A to B (should see connection error)  
• Start application on device B  
• Send a message from A to B (should arrive) |
| 2 Outgoing connections | • Start the application on three devices  
• Send a message from device A to device B and C  
• Send a reply message from B and C to A  
• Assert that all messages arrive |
| Simultaneous incoming | • Start the application on three devices |
### 8.3. Cold Integration Testing

This test involves the entire application stack, including the GUI, HelloService, DataFlooder etc. but without the BluetoothWrapper. The Layer2Stub was used instead.

This allowed us to manually assert that the navigation, preferences, menu, animations and notifications all work well with each other, using random user initiated tests. Using this setup, some development and tweaks to the UI were possible without the need for two physically communicating devices.

### 8.4. Functional Testing

The test cases below were used to test the application as a single unit (full integration of all layers). The goal of these tests is to ensure that the application satisfies the functional requirements in Section 1.6.2.

#### 8.4.1. Two Devices

**Setup:** two devices, one of type Samsung Galaxy S2 with Android 4.0.3 (device A) and one of type Samsung Galaxy S3 with Android 4.1.1 (device B).

1. Test Name: Hello Service.

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on one device and clean DB.</td>
<td></td>
</tr>
<tr>
<td>Start the hello service on a single device</td>
<td>The preferences options changed to: “go invisible”</td>
</tr>
<tr>
<td>Open Debug Activity</td>
<td>The Hello message appears with destination broadcast</td>
</tr>
<tr>
<td>Wait for send attempt.</td>
<td>The “attempts remaining” field is decreased.</td>
</tr>
<tr>
<td>Start application on second device</td>
<td>After a while, The Hello message marked as “known by 1 neighbor” and the second device has the first device’s name in its contact list.</td>
</tr>
<tr>
<td>Open the Debug Activity on the second Device</td>
<td>The Hello message of the first device appears in the debug activity of the second device as well.</td>
</tr>
</tbody>
</table>
2. Test Name: Hello Service Single Message

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on two device and clean DB.</td>
<td></td>
</tr>
<tr>
<td>Start the Hello service on a single device</td>
<td>The preferences options changed to: “go invisible”</td>
</tr>
<tr>
<td>Stop The Hello service and restart it</td>
<td></td>
</tr>
<tr>
<td>Open Debug Activity</td>
<td>There is only one Hello message in the Debug Activity.</td>
</tr>
<tr>
<td>Change the interval between Hello preference to 1 second</td>
<td></td>
</tr>
<tr>
<td>Open Debug Activity and wait a few seconds</td>
<td>There is only one Hello message in the Debug Activity.</td>
</tr>
<tr>
<td>Check the debug Activity of the second device.</td>
<td>There is only one Hello message in the Debug Activity (of the first device)</td>
</tr>
</tbody>
</table>

3. Test Name: Hello Service Quit

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 1 device and clean DB.</td>
<td></td>
</tr>
<tr>
<td>Start the Hello service on a single device</td>
<td></td>
</tr>
<tr>
<td>Change the interval between Hello preference to 1 second and Hello cycles to send to 10000</td>
<td>A new Hello message is created each time (check the timestamp) and there is only one message in the debug activity</td>
</tr>
<tr>
<td>Open Debug activity and refresh every few seconds.</td>
<td></td>
</tr>
<tr>
<td>Turn off the Hello Service</td>
<td>The menu option changes to “go visible”</td>
</tr>
<tr>
<td>Open Debug activity and wait for the last Hello message to expire</td>
<td>There is no new Hello message.</td>
</tr>
<tr>
<td>Change the Hello cycles to send to 1</td>
<td></td>
</tr>
<tr>
<td>Turn on Hello Service</td>
<td></td>
</tr>
<tr>
<td>Open Debug activity and wait for the last Hello message to expire</td>
<td>There is no new Hello message.</td>
</tr>
</tbody>
</table>

4. Test Name: Broadcast Test

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices</td>
<td></td>
</tr>
<tr>
<td>Check broadcast on device A</td>
<td>Additional recipients is grayed out</td>
</tr>
<tr>
<td>Write some message and click “send”</td>
<td></td>
</tr>
<tr>
<td>Check Outbox and Debug Activity.</td>
<td>See that the broadcasted message appears. In the outbox – the message has a broadcast icon</td>
</tr>
<tr>
<td>Open inbox in device B.</td>
<td>The message arrived</td>
</tr>
<tr>
<td>Go to debug activity in device B</td>
<td>The message is still broadcasted</td>
</tr>
</tbody>
</table>
Go to compose Activity of device B | Device A appears in the contact list
---|---
Click broadcast on the device B | Both additional recipients and device A in the contact list are grayed out
Uncheck broadcast | Both fields are enabled again
Check contact and click broadcast | Contact is grayed out again (checked).
Send a message and check debug activity and outbox | Only a broadcast message was sent
Go to Inbox in device A | Only a broadcast message is received
Repeat the last step with contact chosen from the Contact picker | Only a broadcast message was sent and received.

## 5. Test Name: Broadcast – gray out

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices, and start Hello service on device A</td>
<td>The device appears in the contact list of the device B</td>
</tr>
<tr>
<td>Check broadcast on the device B</td>
<td>The contact and the additional recipients’ text box are grayed out.</td>
</tr>
<tr>
<td>Uncheck broadcast</td>
<td>The contact and the additional recipients’ text box are enabled.</td>
</tr>
<tr>
<td>Check broadcast again</td>
<td>The compose activity appears with the appropriate phone in the additional recipients box, broadcast is not checked.</td>
</tr>
<tr>
<td>Open outbox and press resend to some timed out message.</td>
<td>The compose activity appears with the appropriate phone in the additional recipients box, broadcast is not checked.</td>
</tr>
<tr>
<td>Check broadcast again</td>
<td>The compose activity appears with the appropriate phone in the additional recipients box, broadcast is not checked.</td>
</tr>
<tr>
<td>Open inbox and press reply to any message.</td>
<td>The compose activity appears with the appropriate phone in the additional recipients box, broadcast is not checked.</td>
</tr>
</tbody>
</table>

## 6. Test Name: Unicast Test

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices</td>
<td></td>
</tr>
<tr>
<td>In device A, choose device B from the contact picker and send a message.</td>
<td>Message appears in outbox and debug activity</td>
</tr>
<tr>
<td>Look at the device B’s Inbox</td>
<td>Message appears</td>
</tr>
<tr>
<td>Look at the device A’s Outbox</td>
<td>Ack appears.</td>
</tr>
<tr>
<td>Look at the device B’s Debug activity</td>
<td>The message does not appear there (but the ack does).</td>
</tr>
<tr>
<td>Go to the compose Activity of the device B</td>
<td>The first device appears in the contact list</td>
</tr>
<tr>
<td>Turn off the device A</td>
<td></td>
</tr>
<tr>
<td>In the device B, check the first device and send a message</td>
<td>Message appears in the outbox with “processing” icon</td>
</tr>
<tr>
<td>Wait for timeout</td>
<td>Global timeout indication appears</td>
</tr>
<tr>
<td>Start the device A (and the application)</td>
<td></td>
</tr>
<tr>
<td>Check Resend on Outbox and then click send</td>
<td>The message is delivered successfully (with ack)</td>
</tr>
</tbody>
</table>
7. **Test Name: Unicast amount Test**

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices</td>
<td></td>
</tr>
<tr>
<td>Start the Hello service on device B</td>
<td>device B appears in the device A’s contact list</td>
</tr>
<tr>
<td>Choose device B from the contact picker and send the message “hello”</td>
<td></td>
</tr>
<tr>
<td>Clear contact picker field</td>
<td></td>
</tr>
<tr>
<td>Choose device B from the contact list and send the message “hello”</td>
<td>Both messages appear in device B’s inbox</td>
</tr>
<tr>
<td>Now choose the device from both contact picker and contact list and send a message</td>
<td>The message arrives only once</td>
</tr>
</tbody>
</table>

8. **Test Name: Reply Test**

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices</td>
<td></td>
</tr>
<tr>
<td>Start Hello Service on device A</td>
<td>After a while, the device appears in the contact list of the device B.</td>
</tr>
<tr>
<td>In device B, check the device A and send a message</td>
<td>Message appears in the inbox of device A.</td>
</tr>
<tr>
<td>In device A, Click the message and send reply</td>
<td>Compose activity opens with the source’s number.</td>
</tr>
<tr>
<td>Send the message</td>
<td>Message arrives normally</td>
</tr>
<tr>
<td>Send a broadcast message to device A from device B</td>
<td></td>
</tr>
<tr>
<td>Open Inbox on device A</td>
<td>Message arrives normally</td>
</tr>
<tr>
<td>Click the message and send reply</td>
<td>Compose activity opens with the source’s number.</td>
</tr>
<tr>
<td>Send the message</td>
<td>Message arrives normally</td>
</tr>
<tr>
<td>In the device A, press “back” twice</td>
<td>The application becomes hidden (Only 1 activity in call stack)</td>
</tr>
</tbody>
</table>

9. **Test Name: Ack after global timeout**

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices</td>
<td></td>
</tr>
<tr>
<td>Turn routing service off on device B</td>
<td>Menu options changes to “Turn router on”</td>
</tr>
<tr>
<td>Send a message to the device B</td>
<td>Message appears in the inbox of the second device. Ack message appears in the Debug Activity</td>
</tr>
<tr>
<td>Wait for global timeout in device A</td>
<td>The icon in the outbox changes to a “global timeout” icon</td>
</tr>
<tr>
<td>In device A, Click the message and press “resend”</td>
<td>The Compose activity reopens with the same message and destination</td>
</tr>
<tr>
<td>Click Send</td>
<td>Both sent message appear in the outbox</td>
</tr>
</tbody>
</table>
of device A:
One with global timeout,
One with “processing” icon.

Turn on the routing service on second
device

In the device A’s outbox, the Icon of both
messages changes to “Ack”.

10. Test Name: Activity Ordering

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open the Inbox activity in a device with many messages</td>
<td>All messages are ordered by date</td>
</tr>
<tr>
<td>Open the Outbox activity in a device with many messages</td>
<td>All messages are ordered by date</td>
</tr>
</tbody>
</table>

11. Test Name: Compose Activity – Data save

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open compose Activity</td>
<td></td>
</tr>
<tr>
<td>Choose a contact from the contact picker, write some text in the text box field, check another contact in the contact list (not broadcast)</td>
<td></td>
</tr>
<tr>
<td>Open Inbox and press back</td>
<td>The contact and the text still appear</td>
</tr>
<tr>
<td>Open outbox and press back</td>
<td>The contact and the text still appear</td>
</tr>
<tr>
<td>Open debug activity and press back</td>
<td>The contact and the text still appear</td>
</tr>
<tr>
<td>Rotate the screen</td>
<td>The contact and the text still appear</td>
</tr>
<tr>
<td>Press back and reopen activity</td>
<td>Nothing is checked, the additional recipients’ text box is empty, the message field says: “your text here”.</td>
</tr>
<tr>
<td>Check broadcast, enter some text and choose a contact from the contact picker.</td>
<td></td>
</tr>
<tr>
<td>Open Inbox and press back</td>
<td>The additional contact is grayed out, so is the contact list. Broadcast still checked.</td>
</tr>
<tr>
<td>Open outbox and press back</td>
<td>The additional contact is grayed out, so is the contact list. Broadcast still checked.</td>
</tr>
<tr>
<td>Open debug activity and press back</td>
<td>The additional contact is grayed out, so is the contact list. Broadcast still checked.</td>
</tr>
</tbody>
</table>

12. Test Name: Activity rotating

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open the Inbox of device A with some messages in it (send messages from B if needed)</td>
<td></td>
</tr>
<tr>
<td>Rotate the screen</td>
<td>The messages still appear</td>
</tr>
<tr>
<td>Open the Outbox of device A with some messages in it (send messages to B if needed)</td>
<td>The messages still appear</td>
</tr>
<tr>
<td>Rotate the screen</td>
<td>The messages still appear</td>
</tr>
</tbody>
</table>
Open the Debug activity of device A with some messages in it (start Hello service if needed)

Rotate the screen

The messages still appear

### 13. Test Name: Layer 2 connection status

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open the application on two devices, and start the Hello service on device A</td>
<td>The device appears in device B's contact list</td>
</tr>
<tr>
<td>Open the debug activity on both devices</td>
<td>Both devices display “L2 connections: 1”</td>
</tr>
<tr>
<td>Wait for one of the devices to start discovery until “not accepting” appears in Debug activity</td>
<td>Both devices display “L2 connections: 1”</td>
</tr>
<tr>
<td>Turn routing service off on both devices</td>
<td>still “L2 connections: 1”</td>
</tr>
<tr>
<td>Turn off Hello service</td>
<td>still “L2 connections: 1”</td>
</tr>
<tr>
<td>Turn router on and Wait for Debug Activity on both sides to be empty (all en route message will be expired or sent)</td>
<td>still “L2 connections: 1”</td>
</tr>
<tr>
<td>Turn routing service off on both devices</td>
<td>still “L2 connections: 1”</td>
</tr>
<tr>
<td>Kill the application on one device</td>
<td>The second device displays: “L2 connections: 0”</td>
</tr>
<tr>
<td>Start the application again on the device B, and send a message from device A to device B</td>
<td>Message arrives</td>
</tr>
<tr>
<td></td>
<td>Both Debug activity display: “L2 connections: 1”</td>
</tr>
</tbody>
</table>

### 14. Test Name: Compose Activity display issues

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start the application on 2 devices.</td>
<td></td>
</tr>
<tr>
<td>Make sure that each device knows the phone number of the other device a priori (it exists in their phone contacts)</td>
<td></td>
</tr>
<tr>
<td>Send a message from device A to device B</td>
<td>Device A is added to the application’s contact list with the name from the phone</td>
</tr>
<tr>
<td>Wait for an Ack on the device A</td>
<td>Device B is added to the application’s contact list with the name from the phone</td>
</tr>
<tr>
<td>Clear the DB in the device B</td>
<td>The application contact list is cleared as well</td>
</tr>
<tr>
<td>Send a broadcast message from device A</td>
<td>As before, the device is added to the application’s contact list in device B with the name from the phone</td>
</tr>
<tr>
<td>Change the application phone number on both devices to unfamiliar numbers</td>
<td></td>
</tr>
<tr>
<td>Send a new message from device A to device B</td>
<td>Device 1 is added to the contact list of device 2, only phone number is displayed (only once)</td>
</tr>
<tr>
<td>Wait for an ack</td>
<td>Device 2 is added to the contact list of</td>
</tr>
</tbody>
</table>
15. Test Name: No Passkey request

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to Bluetooth settings on both devices.</td>
<td></td>
</tr>
<tr>
<td>Make sure the devices are not paired (unpair if needed).</td>
<td></td>
</tr>
<tr>
<td>Send a message from one device to the other</td>
<td>The message arrives, no popup (passkey or pincode request) appears</td>
</tr>
</tbody>
</table>

8.4.2. Three Devices

**Setup:** three devices, two of type Samsung Galaxy S2 with Android 4.0.3 (devices A and B) and one of type Samsung Galaxy S3 with Android 4.1.1 (device C).

16. Test Name: Presentation Test

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices (named A, B), while the application is not running in C</td>
<td>Verify That the message appears In: A’s outbox and B’s Debug activity</td>
</tr>
<tr>
<td>Send a message from A to C</td>
<td></td>
</tr>
<tr>
<td>Turn off the application in device A and turn on the application in device C</td>
<td>The message arrives to device C. Device C’s debug activity contains a new Ack Message, and it is transferred to B.</td>
</tr>
<tr>
<td>Turn off the application in device C and turn on the application in device A</td>
<td>The ack arrives back to A.</td>
</tr>
</tbody>
</table>

17. Test Name: Multiselection

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on all three devices and Start Hello service on all three devices.</td>
<td>After a while, all 3 devices appear in each other’s contact list.</td>
</tr>
<tr>
<td>In device A: choose B and C from the contact list and send them a message</td>
<td>Both messages arrive</td>
</tr>
<tr>
<td>In device B: choose the numbers of A and C from the contact picker and send them a message</td>
<td>Both messages arrive</td>
</tr>
<tr>
<td>In device C: choose A from the contact list and B from the contact picker and send them a message</td>
<td>Both messages arrive</td>
</tr>
</tbody>
</table>

18. Test Name: Self broadcast

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices (named A,</td>
<td></td>
</tr>
</tbody>
</table>
B), while device C is turned off.

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send a broadcast message from A.</td>
<td>Verify That the message appears in: A’s outbox and B’s Debug activity</td>
</tr>
<tr>
<td>Turn off device A and turn on device C</td>
<td>The message arrives to device C. Device C’s debug activity contains a new Ack Message, and it is transferred to B.</td>
</tr>
<tr>
<td>Turn off device B and turn on device A</td>
<td></td>
</tr>
<tr>
<td>Look at device C’s debug activity</td>
<td>After a while, the broadcast message is known by 2 neighbors</td>
</tr>
<tr>
<td>Look at device A’s Inbox</td>
<td>The message does not appear</td>
</tr>
</tbody>
</table>

19. Message arrives twice

<table>
<thead>
<tr>
<th>Step</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start application on 2 devices (named A, B), while device C is turned off.</td>
<td></td>
</tr>
<tr>
<td>Turn off Routing Service in Device B</td>
<td>Message Arrives</td>
</tr>
<tr>
<td>Send a message from A to B</td>
<td></td>
</tr>
<tr>
<td>Turn off device B and turn on device C</td>
<td>The message appears in C’s Debug Activity</td>
</tr>
<tr>
<td>Turn off device A and turn on device B</td>
<td></td>
</tr>
<tr>
<td>Open device C’s debug activity</td>
<td>The message appears with “know by 2”.</td>
</tr>
<tr>
<td>Open device B’s Inbox</td>
<td>The message appears once.</td>
</tr>
<tr>
<td>Open device B’s Debug Activity</td>
<td>The message does not appear</td>
</tr>
<tr>
<td></td>
<td>The toast for this message in device B is shown only once!</td>
</tr>
</tbody>
</table>

8.5. Performance Testing

In each of the following tests:

- 10 messages are sent at once to a single neighboring device.
- The returning ACK datagrams are recorded.
- Each test was repeated at least 10 times.

From the returned ACKs we can measure both the one-way and the round trip latency. The round trip latency is straightforward: we measure the time between hitting the send button and the incoming layer 3 event of the expected ack.

Measuring the one-way latency is a bit more difficult: The ACK datagram is generated at the destination when the message is received. At that point the ACK datagram’s timestamp is also decided. We can therefore deduce the one way latency by subtracting the start time from the ACK’s timestamp. This measurement is less reliable as the two device’s clocks are not in perfect sync. They were synched manually, to a confidence margin of about 100ms. This kind of synching was impossible for more than two devices, and for this reason this measurement is ignored in some tests.

In analyzing the measured latencies, we distinguish two different statistics.
- The latency distribution of all the messages in all the repetitions of the test
- The latency distribution of the first message of each repetition in the test. This distribution is consistent with the predefined performance targets.

In the following tables we describe the test scenarios as well as the predefined performance traits that are being tested. The ‘results’ column takes into account only the latency of the first out of the 10 messages in each repetition of the test.

The tables are followed with relevant graphs with the full latency distribution of each test. The performance goals (Section 1.6.3) are also shown on the graphs.

### 8.5.1. Two Devices

<table>
<thead>
<tr>
<th>ID</th>
<th>Test</th>
<th>Description</th>
<th>Tested Traits</th>
<th>Results (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cold Start Unidirectional</td>
<td>Devices are freshly started with no preexisting Bluetooth connection</td>
<td>21,23,27</td>
<td>One Way Average Latency: 14170 One Way 90th Percentile: 15993 Round Trip Average Latency: 15263 Round Trip 90th Percentile: 16655 Packet loss: 0%</td>
</tr>
<tr>
<td>2</td>
<td>Cold Start, target busy Unidirectional</td>
<td>One device is idly broadcasting Hellos. Other device enters the room with the enqueued datagrams.</td>
<td>1,3,7</td>
<td>One Way Average Latency: 17457 One Way 90th Percentile: 25202 Round Trip Average Latency: 18964 Round Trip 90th Percentile: 27507 Packet loss: 0%</td>
</tr>
<tr>
<td>3</td>
<td>Warm start Unidirectional</td>
<td>Devices are allowed to exchange Hellos. Then one device sends the messages.</td>
<td>22,24,27</td>
<td>One Way Average Latency: 695 One Way 90th Percentile: 1206 Round Trip Average Latency: 2391 Round Trip 90th Percentile: 4618 Packet loss: 0%</td>
</tr>
<tr>
<td>4</td>
<td>Cold Start Exchange</td>
<td>Same, with both devices trying to unicast</td>
<td>21,23,27</td>
<td>One Way Average Latency: 17451 One Way 90th Percentile: 25129 Round Trip Average Latency: 20589 Round Trip 90th Percentile: 29963 Packet loss: 0%</td>
</tr>
<tr>
<td>5</td>
<td>Warm start Exchange</td>
<td>Same, with both devices trying to unicast</td>
<td>22,24,27</td>
<td>One Way Average Latency: 639 One Way 90th Percentile: 1031 Round Trip Average Latency: 3586 Round Trip 90th Percentile: 5195 Packet loss: 0%</td>
</tr>
<tr>
<td>6</td>
<td>Cold Start Interference</td>
<td>Same, with a nearby incompatible discoverable</td>
<td>21,23,27</td>
<td>One Way Average Latency: 14978 One Way 90th Percentile: 16713 Round Trip Average Latency: 15863 Round Trip 90th Percentile: 17911</td>
</tr>
</tbody>
</table>
The following four graphs describe the latency distribution in each of the tests, for the cases:

- Total message latency distribution, one way.
- 1st message latency distribution, one way.
- Total message latency distribution, round trip.
- 1st message latency distribution, round trip.

The target requirements were met in all cases.

It should also be noted that the batch of ten datagram were all initialized with very close timestamps (millisecond differences). The ten datagrams were then enqueued in the router. It was apparent during the experiments that once the first datagram made it to the other device, the other nine datagrams immediately followed. This manifests itself in the graphs in the following ways:

- Total message latency distribution graphs are ‘divided’ to ten mostly linear parts, corresponding to each of the repetitions of the test. In each repetition, once the first datagram is sent, the following datagrams are processed and sent in linear time.
- 1st message latency distribution graphs are only made up of ten data point. These are a better representation of the stochastic property that we are measuring, i.e. the time required to create a Bluetooth connection. The data points are connected linearly for visualization properties, but it is important to note that this is just an interpolation.
Figure 23 - Two Device One Way Latency Distribution
Figure 24 - Two Device One Way Latency Distribution (First Message In Batch)
Figure 25 - Two Device Round Trip Latency Distribution
Figure 26 - Two Device Round Trip Latency Distribution (First Message In Batch)
8.5.2. Three Devices

All three device tests consist of a travelling node connecting two remote nodes.

<table>
<thead>
<tr>
<th>ID</th>
<th>Test</th>
<th>Description</th>
<th>Tested Traits</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Cold Start</td>
<td>Devices are freshly started with no preexisting Bluetooth connection</td>
<td>25,28</td>
<td>Average cold hop latency: 14702 90\textsuperscript{th} percentile cold hop latency: 14564 End to end packet loss: 0%</td>
</tr>
<tr>
<td>8</td>
<td>Warm Start</td>
<td>Each pair in the line has a connection</td>
<td>26,28</td>
<td>Average warm hop latency: 1648 90\textsuperscript{th} percentile warm hop latency: 2842 End to end packet loss: 0%</td>
</tr>
</tbody>
</table>

In the three device cold start scenario (test seven above), the round trip time until an ack is received is comprised of two cold hops and two warm hops (the ACKs travel back on open connections). For this reason, in order to measure the ‘per cold hop’ latency, we measured the one way latency and divided by two.

On the other hand, measuring the warm hop latency was done by dividing the round trip time by four.

For both the cold and warm scenarios, the target requirements were met.

In the following graphs, legend entries tagged as ‘1\textsuperscript{st}Ack’ indicate that the measured time is the latency until the first ACK datagram is received (out of ten).
Figure 27 - Three Device Relay, Per Hop Latency Distribution
8.5.3. Many Devices

This experiment was done with a group of several friends, helping us to test the application over pizza (see Beta Testing Section 8.6). 6 devices in total were available.

<table>
<thead>
<tr>
<th>ID</th>
<th>Test Description</th>
<th>Tested Traits</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>One to many Cold Start</td>
<td>21,22,27</td>
<td>Round Trip Average Latency: 21706</td>
</tr>
<tr>
<td></td>
<td>Devices are in the same room, one device unicasts 10 datagrams to different devices</td>
<td></td>
<td>Round Trip 90th Percentile: 30920</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packet loss: 6%</td>
</tr>
<tr>
<td>10</td>
<td>One to many Warm Start</td>
<td>22,28</td>
<td>Round Trip Average Latency: 2094</td>
</tr>
<tr>
<td></td>
<td>Devices exchange Hello before unicasting the 10 datagrams</td>
<td></td>
<td>Round Trip 90th Percentile: 3507</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packet loss: 1%</td>
</tr>
<tr>
<td>11</td>
<td>Line relay warm start</td>
<td>26,28</td>
<td>Per Hop Average Latency: 2132</td>
</tr>
<tr>
<td></td>
<td>10 datagrams are unicast to a single device, 3 hops away.</td>
<td></td>
<td>Per Hop 90th Percentile: 3750</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Packet loss: 0%</td>
</tr>
</tbody>
</table>

The one-to-many scenario (tests 9, 10) involved uncasting each batch of 10 messages to different devices in the room. Cold starting this scenario meant resetting each device before each test (our performance test activity supports such ‘clearing’).

The one way latency measurement was completely unreliable on the one-to-many scenario due to clock synchronization issues. We therefore only measured the round trip times.

In the one-to-many scenario, the target requirements were met by the first message of each batch of ten. However, looking at the latency distribution of all messages, only the minimal requirement was met.

When performing the Line Relay scenario (test 11), we set the devices up in a way such that each relaying device sees only two neighbors. It became apparent that a cold start scenario would take too long to test for, and we had a limited timeframe on which we could test on many devices. Therefore, we decided to only test a warm scenario.

As bad luck would have it, one device’s battery went out, and we were only able to complete 7 out of the intended 10 repetitions of this warm relay test.

In the warm line relay scenario, the target performance requirements were met.

In the following graphs, legend entries tagged as ‘1st Ack’ indicate that the measured time is the latency until the first ACK datagram is received (out of ten).
Figure 28 - Many Devices, Round Trip Latency Distribution
Figure 29 - Four Device Relay, Latency Distribution
8.6. Beta Testing

At the final stages of development, it became necessary to test the application in a real life environment:

- Many devices
- Different OS versions and device models
- Incompatible devices that only make themselves discoverable by Bluetooth

In such a setting we wanted to test our application for stability, functionality and performance.

To this end, we invited some friends over to a pizza and Android party, where they got free pizza as a token of appreciation for their participation.

In this event, we:

- Turned on Bluetooth discoverability on all devices and made sure this did not disrupt communications between two compatible devices.
- Installed the application on every compatible device, and allowed our guests to play around with the application.
- Followed the protocol for performance testing the ‘Many Devices’ test scenarios (performance tests 9-11, see 8.5.3).

9. Future Work

9.1. Performance Optimizations

See Section 5.3.6 for unimplemented router optimizations.

9.2. Complex Payload Type

It is possible to encode any sort of data on the datagram’s payload string (which can be replaced with the more general ‘byte array’ type). A key-value system would enable the storing and retrieving of the data.

For instance, it is possible to bundle identifying information, in addition to the source address, such as a display name.

One might also bundle a small audio file in the datagram. This will enable simple ‘push to talk’ capabilities (assuming we improve our latency results, see 9.6).

A very useful feature would be to bundle GPS coordinates with each datagram, including the periodic Hello datagrams. This, alongside a simple activity, will allow the user to see where his friends are located. This could be immensely useful in the various situations in which this ad hoc network is needed: finding a person in a crowd or in a disaster area.
9.3. Multicasting

The data flooding routing scheme is well suited for a simple implementation of multicasting. As it is, all datagrams reach all the devices in the network. The designated device simply picks up the datagrams meant for it by matching its own ID with the on the datagram’s global destination.

Broadcasting was implemented simply: a special ‘broadcast’ address is picked up by all devices in the network.

It is therefore easy to extend this mechanism for multicasting. The destination address should simply contain a label identifying the group. The devices that belong to said group should pick it up from the stream alongside the other datagrams.

In code, this means replacing the layer 3 member ‘myDeviceID’ with a list of labels to look out for. The list will include the device ID but also the labels of the groups to which the device belongs. Additional code should also allow the user to choose which labels he subscribes to, perhaps with a new activity.

It should be mentioned that collisions would be possible between labels and actual IDs (Phone numbers). This can be prevented by either

- Prefixing all addresses with some indication of the address type (“unicast:55555”, “Multicast:55555”).
- Adding a special ‘multicast’ datagram type.

Finally, the visibility feature should be extended. The user would not want to see every single online node on the network, but rather only those nodes in his group. This can be realized with the following steps:

- The Hello datagrams should contain a list of labels to which the device is subscribed. This can be realized with the bundling suggested above.
- The Compose activity should allow filtering contacts by checking one of the labels to which the device is subscribed to.
- The database scheme must be updated with a new table to hold the address/label mappings and their age.

9.4. Connectivity to External Networks

This application is relevant where there is no mobile infrastructure, or it has been damaged or overloaded. In most scenarios, the lack of mobile service is localized - service may be available a few blocks over.

It would therefore be of use to utilize the ad hoc network to transfer the messages to nodes that do have available mobile service. These nodes would then act as gateways, to transfer messages from the ad hoc network to the mobile network/internet and vice versa.
9.5. Authentication and Security
The user is currently able to ‘change’ his phone number, thus sniffing datagrams meant for other users. This can be averted by authenticating the phone number with an SMS, similarly to how the ‘whatsapp’ instant messaging application authenticates its users. The user enters his phone number, and an SMS is automatically sent to it. The application should intercept it to verify if it is the expected SMS.

Even with this mechanism in place, the data is all stored as plaintext in the DB. In order to fully secure this information, it must be encrypted by either a preshared key or a public key. As a partial solution, we may simply ‘scramble’ the text in the DB in such a way that only an expert user will be able to decipher it (e.g. shifting all the chars in the strings with a constant). This is only marginally better than plaintext but would make eavesdropping that much more difficult to the layman.

9.6. Alternative Layer 2 Implementations
It is possible to swap the Bluetooth implementation with some more modern variations, such as Wi-Fi Direct. This may provide us with better range, quicker discoveries, etc.

9.7. Optimization of Routing Protocol Parameters
Our data flooding routing protocol make use of several time constants, namely:

- Minimal time between discoveries
- Maximal time between discoveries
- Local timeout
- Flooding time

These were manually tweaked, based on our grasp of how they interact (see Preferences in Section 6.7). It is possible to empirically optimize these constants by doing more thorough testing, or change them dynamically to respond to the ad hoc network's load and a device's mobility.

Bibliography
Appendix A – User Manual

The Ad-Hoc SMS network application allows simple text messaging via Bluetooth. It allows passing a text message as long as there is territorial contiguity of devices running the application between the sender and the receiver.

The user may choose to send a message specifically for one / several destinations, and he may also choose to broadcast certain messages to the entire network.

Installation

Install the application normally and run it. On the first time you run the application, a setup screen will greet you. Fill in your phone number, including the international prefix, and continue.

After inserting the phone number, the main screen will open. On the next time you run the application, the main screen will be opened immediately.

Main Screen (Compose)

The main screen is the first screen that shows when the application is started (after the first time). This screen will be used for composing the messages and selecting the recipients.
Compose your message in the text box instead of ‘Your text here:’

Recipients may be chosen in several ways:

- Your device automatically remembers devices that have recently been active on the ad hoc network. They will appear in a contact list right below your text. If their phone numbers are listed in your phone’s normal contacts data, their name will appear beside their phone number. Tick one or more recipients from this list to choose them as recipients.
- If a friend of yours does not appear on the contact list, it means he has not been active recently, but it is still possible that he is available if you send him a message. The top text box is meant for ‘Additional Recipients’. You can manually insert phone numbers that do not appear in the provided contact list.
- Clicking the ‘Phone Contacts’ button will open you phone’s contact picker. Selecting a contact from there will fill in his number in the ‘Additional Recipients’ text box.
- If you tick the ‘Broadcast’ recipient, your message will be received by all the devices available on the network. All the other methods to select recipients will be grayed out to remind you that this option is on, and that there is no need to individually select further recipients.
Finally, after choosing recipients and inserting the wanted text, pressing the “Send” button will send the message.

On the bottom of the screen you will find two buttons: Inbox and Outbox (Note: the Debug and Test buttons should not appear on the production version). These will lead you to dedicated Inbox/Outbox screens.

**Inbox**

The inbox allows you to see incoming messages either meant specifically for you or broadcasted. Tapping any message will open the reply dialog, enabling you to quickly write a reply to the sender. Clicking “OK” will re-open the main screen (Compose), where the “additional recipients” field will already hold the phone number of the sender:
Outbox

The outbox allows you to see outgoing messages (both broadcasts and messages for specific recipients). A broadcasted message will appear once in the outbox. A message for multiple recipients will appear several times, one time per each recipient.

On the right sides of each message, you can see one of the status icons below:

- A Broadcast icon (_broadcasting) indicating that the message was broadcasted to all possible devices in the networks. There will be no indication whether this message was successfully transmitted to any device.
- A successful transmission icon (✓) indicating the message was successfully transmitted to the desired recipient and a positive feedback was received by the sender.
- A global timeout icon (❌) indicating that the global timeout period (may be changed in the preferences windows, see below) had passed, without any indication that the message reached the desired recipient. Please note that this does not mean the message was not received necessarily.
- A processing icon (○) indicating that none of the 2 states above is valid. Meaning, there was no indication of successful transmission, nor the global timeout had passed. It is likely that the message is still in transit!

Tapping any timed-out message will open the resend dialog, enabling you to quickly resend the message to the desired destination. Clicking “OK” will re-open the main screen (Compose), where the “additional recipients” field will already hold the phone number of the original recipient and the text field will hold the original text.

Notifications
There are 2 types of events that initiate a special notification, even if the application is not running in the foreground.

- Incoming Message
- Global timeout

Both events will be notified in 3 ways:

- A special sound will be played.
- A text message appearing in the bottom of the screen, describing the event:
A notification icon of the letter B (for Bluetooth) in the top left corner of the screen. Scrolling down the notification bar will open the Android notifications menu, where more details will be presented (how many timeouts, how many incoming messages). Tapping the “incoming message” notification row will open the Inbox. Tapping the “global timeout” notification row will open the outbox.

Menu
When in the main compose screen, pressing the Android menu button will bring up the application’s menu. The application’s menu consists of three options:
Go Visible / Invisible
When the device is in a ‘Visible’ state, it broadcasts periodic ‘Hello’ messages to let the rest of the network know he is online. For instance, other users will be able to see the device in their compose screen’s contact list. The device will eventually go back to the invisible state after a predefined amount of time that can be set in the preferences. This is done in case you forget to switch it off manually, which may cost you in battery life.

It is important to note that any message that you send while ‘invisible’ will alert the other users that you are online. So the ‘invisible’ state does NOT guarantee invisibility. Only the opposite is true: the ‘Visible’ state guarantees visibility.

Turn Router On/Off
Turning the router off means you will still receive, but not send or forward, messages. This feature is meant to be used if you are travelling with your device between two distant groups of people using the application. When you travel between them, your device carries messages meant for the other group!

While in transit, you may choose to switch the router off to preserve battery life: there is no one to send the messages to anyway!

Preferences
The preference screen allows you to customize some options of the application.
Main Preferences

- **Phone Number**: change your own phone number. The application recognizes messages that are meant for you using this number. Outgoing messages are stamped with this number as well, so that the recipients know who the sender is. It is recommended that you enter your number with its international prefix.

- **Visibility Definitions**: change how this device announces itself to other devices, and how it interprets other devices announcements.
  - **Broadcast Interval**: when your device is in the visible state, this preference controls how frequently ‘Hello’ messages will be broadcasted.
  - **Visibility Duration**: when you go to the visible state, your device will automatically go back to the invisible state after this much time has elapsed. This is done to save battery life in the event that you forget to go back to the invisible mode.
  - **‘Who’s Online?’ Period**: Devices that are considered ‘online’ are shown on the compose screen’s contact list. Online devices are devices that have been active (i.e. have sent messages) at least as recently as the time set in this preference.

- **Advanced**: read further
Advanced Preferences

Advanced preferences affect how your device routes messages to other devices by configuring several parameters. You are advised not to change any advanced preference. If you do, you can always choose the ‘Restore Defaults’ Option to restore them to the preset settings.

- Restore Defaults: Restores all the settings to their original presets, except for the phone number.
- Global Timeout: When you send a message to another person, we cannot be certain that he received the message until he sends back an ‘acknowledge’ message, which is not shown to you explicitly. If this acknowledge message is not received after the amount of time specified in this preference your original message is considered lost, as indicated by the red X in the outbox, as well as a notification. On very large networks that span several sites, you may choose to increase this amount.
- Time to Live: Your message may travel through many devices before it reaches its destination. However, we would not want it to travel forever. This preference limits the number of ‘hops’ you messages are allowed to make.
- Re-flooding: When the application sends or forwards a message, it does not try only once. Rather, every few seconds your device looks for new neighbors and forwards your message to them (if they do not already have a copy).
  - Re-flood Interval: The amount of time waited between any two attempts to forward a message to new neighbors. It is important that this is less than the Minimum Rest Time (see below).
  - Re-flood Time: Your device will keep trying to forward each message for this long a time. This time should be long enough to ensure your message has a good ‘footing’ in the network, and that it will still be able to advance even if a few devices leave the network.
Bluetooth Discoveries: Discovery is the process of looking for new neighbors. This is costly in terms of battery use, and also makes your device unavailable for incoming messages from new neighbors.

- Maximal Rest Time: If your device can still be useful without discovering (it still has messages to send to its currently known neighbors), then it will not discover. However, over time this may cause the device’s known neighbors to be very out of date, so we enforce a discovery after this much time has passed since the last one.

- Minimal Rest Time: If your device has nothing to do with its current neighbors (they already got all the messages that your device holds), then it will try to discover new neighbors, but only if it has been this long since the last discovery.

Delete Options – See below

Delete Options

- Clear Inbox: Deletes all your incoming messages, unless they are still being forwarded (broadcasts).
- Clear Outbox: Deletes all your outgoing messages, except those that are still being forwarded.
- Clear Entire Database: Deletes all the known messages, including messages that are only passing through.