Master's Thesis in Computer Science

A Protocol for Smartphone Ad-Hoc Networks

June 1, 2024

Authors

Troels Lind Andersen
trand19@student.sdu.dk

Viktor Strate Kløvedal
vistr19@student.sdu.dk

Supervisors

Marco Peressotti
peressotti@imada.sdu.dk

Ruben Niederhagen
niederhagen@imada.sdu.dk

SDU University of Southern Denmark

Department of Mathematics and Computer Science
University of Southern Denmark

Abstract

The reliance of digital communication in the modern world makes intentional disruption of such communication a potential tool for governments to suppress civil unrest and protests. In an attempt to combat this, we introduce the Starling protocol, which can be used to establish secure, ad-hoc, peer-to-peer networks using common smartphones, in order to facilitate communication without any other supporting infrastructure than the phones themselves. Together with the protocol, we have implemented a fully working smartphone application to demonstrate its purpose, as well as a purpose-built simulator used to test and measure its performance. We evaluate the protocol both by performing small-scale real life tests, and simulations of its use in protests, and find that it performs very well for group communication.

Resumé

Som brugen af digital kommunikation øges i den moderne verden, kan dette misbruges af regeringer til at undertrykke og begrænse demonstrationer. I et forsøg på at modvirke dette, introducerer vi i denne afhandling Starling-protokollen, som kan bruges til at etablere sikre, peer-to-peer, ad hoc, netværk ved brug af almindelige smartphones og på den måde facilitere kommunikation uden fast infrastruktur. Sammen med protokollen, har vi implementeret en funktionsdygtig smartphone app for at demonstrere protokollens praktiske brug. Derudover har vi udviklet en simulator med det formål at teste protokollen. Vi evaluerer protokollen både på baggrund af praktiske test samt simuleringer af dens brug under demonstrationer, og når frem til at den præsterer fremragende ved gruppekommunikation.

Contents

Chapter 1

Introduction

People are more connected than they have ever been through technology. Smartphones and the Internet is ubiquitous in developed countries and is rapidly increasing in developing countries. The world has become reliant on this connectivity, and it significantly influences the daily lives of most people. However, this reliance can be abused by nation states and governments in order to suppress its people and remain in power.

This could be seen under the civil unrest and protests that arose throughout Iran following the death of Masha Amini in 2022. Within the following week, when the protests had spread across most of Iran, the Iranian government shut down Internet services to make it harder for protesters to communicate and spread awareness of demonstrations. This action together with heavy security presence in Tehran made the government successful in deterring the protesters. [24]

The widespread use of technology such as smartphones also poses risks of mass surveillance. When the phones automatically connect to cell towers, their activity is recorded by the cell carrier which can later be used to determine where the phone has been and when. It is not uncommon for states to use this data as evidence in court cases. This was the case in a protest in Ukraine regarding restrictions on speech and assembly rights which was held in Kiev in 2014. A few days after the protest, several participants received a text on their phone which stated that they had participated in a "mass disturbance". It later became publicly known that the cell phone carrier Kyivstar had shared cell tower information with the government. [42]

These problems give rise to another kind of mobile network known as Mobile Ad-hoc Network (MANET). A MANET [15] is a decentralized network that enables devices to communicate directly with each other to form a self-sustained, ad-hoc network without relying on any existing infrastructure like cellular towers or access to the Internet. The devices on the network work together to forward and route packets for each other. This flexibility make MANETs especially useful for use cases where the traditional network infrastructure is unavailable or has been intentionally disrupted.

Due to smartphones becoming more capable, such as with the introduction of Bluetooth Low Energy (BLE) on more smartphones, it has become more manageable to build MANETs using only the technologies a common smartphone provides. Such technologies have been seen to emerge in different places. One instance was in Hong Kong in 2019 where protests erupted due to a new bill regarding change in the rules for extradition. Many protesters did not trust the government to not censor the Internet, and began using an offline messaging app called Bridgefy. [52]

One of the earliest examples of an offline messaging application for smartphones was

FireChat [37] developed by OpenGarden, with its first public release in 2014. It quickly became popular and was used under multiple conflicts in the following year after its release, even though it originally only supported sending public unencrypted messages. In 2018, it was however discontinued and removed from the app stores.

1.1 State of the Art

This section outlines the current state of the art of offline smartphone messaging applications.

Bridgefy [16] is an application that uses BLE to facilitate offline communication by forming MANETs on smartphones. Although it allows messages to travel multi-hop, it only does so within private group chats, thus messages cannot be forwarded by foreign devices. The application also needs an internet connection to set up an account, which we see both as a potential privacy issue, and it could render the entire application useless in cases with no internet access. It has also been shown to contain severe security vulnerabilities [3]. And after an attempt at rectifying these, the app was shown to still not be secure later by the same authors [2].

The Berty messaging app [10] is a similar but open-source alternative backed by the non-profit organization Berty Technologies. It also uses BLE for offline communication but can also utilize multicast DNS or the Internet when available. Similar to Bridgefy, it does not allow for multi-hop connections, which means that messages cannot be forwarded by foreign devices.

Briar [46] is an open-source Android only mobile application. It takes privacy very far, for example by providing deniability. Briar does so by obfuscating all messages in a way that makes them indistinguishable from random bytes for anyone except the sender and recipient. Although Briar has existed for a while, it has not seen large traction in the real world in the same order as the previously mentioned alternatives have.

A common deficiency for these applications is the lacking support for multi-hop communication. The ability for multi-hop communication could potentially be used to form larger smartphone MANETs. In this thesis we therefore set out to design a new MANET protocol for smartphones that addresses some of these shortcomings, to further advance the possibilities for off the grid smartphone communication.

1.2 The Starling Protocol

In this thesis we introduce the Starling protocol, an open protocol designed for smartphone MANETs, for use during protests and civil unrest where access to traditional means of digital communication is restricted or is unsafe to use due to surveillance. The protocol enables communication for these cases by allowing phones to directly communicate without static infrastructure. It introduces multi-hop connections to increase reach in the network.

Along with the protocol design, we present an implementation of the protocol along with a fully working messaging app for smartphones, to evaluate the protocol in practice. Additionally, a simulator has been implemented to simulate the protocol and evaluate the correctness and performance of the protocol implementation on large scale tests.

We start by analyzing the scenario and establishing the requirements needed for the protocol in chapter 3. In chapter 4 we then discuss and present the protocol design, and then present the implementation in chapter 5. After this, we introduce the simulator in chapter 6 and the implementation of the smartphone application in chapter 7. Lastly, we evaluate the security and efficiency of the protocol in chapter 8.

Chapter 2

Background

One of the main challenges of routing in MANETs, is that nodes are connected through a wireless medium and are often not stationary, and thus the topology of the network will likely change over time. This means that methods of routing in traditional networks with more static topology are not sufficient. Many routing algorithms have been proposed to solve the problem of routing in MANETs, and these can broadly be divided in two main categories; *reactive routing* and *proactive routing.*

This chapter will give an overview of various reactive and proactive routing algorithms discuss pros and cons, while drawing inspiration from [14, 33]. Epidemic routing will also be introduced briefly, as well as various attacks on routing protocols.

This chapter will also provide some background in message synchronization as well as a transport protocol for MANETs. Lastly we briefly present an overview of Bluetooth Low Energy (BLE).

2.1 Proactive Routing

In proactive routing, every node keeps a routing table with entries for the other nodes on the network, and attempts to constantly keep this routing table up to date. This means that when a node wants to connect to another node, it is quite straight forward since the routing table will inform the node of what to do. Maintaining the routing table to ensure that it is up-to-date can require a lot of work for large or dynamic networks and thus creates some amount of overhead.

2.1.1 Destination-Sequenced Distance-Vector

Destination-Sequenced Distance-Vector routing (DSDV) [40] is similar to the Bellman-Ford algorithm for finding the shortest path in a graph. Nodes essentially keep a local routing table, and share this with immediate neighbors regularly or when topology changes occur.

The routing table consists of so-called *distance vectors.* Each node i has a distance vector $\{d^x_{i,j}\}$ for each other node x. These vectors consist of the distances from node x to node i by going through the neighbor j of i. In an ideal setting, this means that any node knows the shortest route to every other node, and thus can send packets and forward packets for other nodes. However, due to the fact that nodes potentially can have stale routing information, loops can occur in the network. DSDV attempts to remedy this by attaching a sequence number when sharing routing information. Nodes can then prioritize the freshest parts of their distance vectors.

Another problem is that nodes moving in the network, could cause a storm of broadcasts when nodes try to update and rebroadcast their routing tables. DSDV addresses this by having nodes wait a bit with broadcasting their routing information, such that they dampen the fluctuations of the routing table. Essentially this attempts to avoid flooding the network when many changes happen, and lets the network settle a bit.

2.2 Reactive Routing

Reactive routing takes another approach and generally has a query/response strategy. In essence, nodes do not know the structure of the whole network and mostly have information about their immediate neighbors. This means that when a node wants to send a packet to another node, a route will have to be found at that time. The clear downside is that the node does not know how to contact the other node, or even if the other node is on the network. On the other hand, this means that there is very little overhead compared to proactive routing, since nodes do not have to constantly maintain routing tables. In environments where the network has a very dynamic topology this might be favorable. Another positive effect of this is that in times of low demand, the network will not be very active, which could positively impact battery life of the participating devices.

Usually reactive routing protocols have the concept of a *route request* (RREQ) as well as a *route reply* (RREP). Route requests are generally packets that are sent when a node wants to communicate with another node on the network. These are then usually continuously propagated throughout the network. In most cases, the route request contains a request ID or some other way of making the route request unique. If a node receives a route request they have seen before, they can ignore this request. This helps avoid cycles, since the route request spreads in a tree-like manner.

For most reactive routing protocols, a route reply is the packet sent as a response when the destination node receives a route request. The nodes can then route the route reply back along the path which was discovered, until the originator is reached.

2.2.1 Dynamic Source Routing

Dynamic Source Routing (DSR) [26] is an example of a purely reactive routing algorithm. In DSR, nodes only perform routing related work when they want to communicate, or when they help forward communication for their neighbors. That is, it is entirely based on demand, which means that it has low overhead compared to proactive routing algorithms. DSR has two main mechanisms.

- **Route Discovery:** When a node wants to send a message but does not know a route to the destination
- **Route Maintenance:** Detecting when a route no longer works, and fixing it

During route discovery, a node creates a route request which consists of a request ID and identifiers for the originator and target node, as well as an empty list. When nodes forward the route request they will append their own identifier to the list. This means that once the target node is reached, the route request will contain a list of the intermediate nodes on the route. The target node then responds with a route reply containing this list. Once the origin node receives the route reply it can store the list of intermediate nodes, called the *source route*, in its route cache. If it wants to communicate with this node again, it can simply create a route request which contains the source route, and the forwarding nodes will forward it along the given route until the destination is reached. While this description relates to how DSR operates with bi-directional links, it also has the ability to operate with networks containing uni-directional links. In that case, the target node will, once it receives the route request, issue its own route request instead of responding with a route reply, thus potentially creating different routes for each direction of communication.

During route maintenance, nodes will discover if a link has been broken. This happens when forwarding a packet along a source route. Each node on the path is responsible for detecting that the next node received its message. This can be done by acknowledgements (either active or passive). If a node does not receive the message after a set number of retries, a *route error* (RERR) message is sent back to the originator, which then invalidates that route in its route cache and will initiate a new route request.

Outside of route discovery and maintenance, nodes can learn of new routes by overhearing other nodes and from those unrelated messages obtain routes that they can cache. An example of this could be a route request containing a source route. Nodes can also used their cached routes (or parts of these routes) to help when receiving a route request. The forwarding node can thus simply return a route reply to the sender rather than forwarding it to the receiver. The reply consist of the concatenation of the route taken so far and the route from the cache. It however needs to check that there is not any duplicates in the concatenation of the two routes.

2.2.2 Ad hoc On-Demand Distance Vector

Ad hoc On-Demand Distance Vector (AODV) routing [39] is based on some of the same principles as DSDV but has a reactive approach instead of a proactive one. This inherently means that AODV does not intend for nodes to have knowledge of the entire network, but only knowledge of the actively used routes. However, AODV still uses distance vectors and routing tables like DSDV. It also uses sequence numbers to guarantee loop-free routing.

AODV uses a broadcast route discovery mechanism somewhat similar to DSR. The main difference is that instead of constructing a list of the intermediate nodes in the packet, the intermediate nodes each record the address of the node they first received the route request from. This makes it possible for a route reply to be routed backwards along the intermediate nodes.

Every node maintains a routing table, which unlike DSDV is not complete. An entry for a node in the routing table consists of the next hop node, a sequence number and a metric (often the number of hops to the node), among other values. The sequence number represents how “fresh” the route is. Route requests also contain a sequence number, which specifies how fresh the originator needs the route to be. When a route request is broadcast, an intermediate node responds with a route reply only if they are the destination node, or if they have a fresh enough entry for the destination node in their routing table. A route entry is deemed as fresh if the sequence number contained in the route request is less than the sequence number in the routing table of the intermediate node.

The route requests are uniquely identified by a sequence number and the destination. This means that nodes can ignore route requests they get from other nodes if they have already handled the same one. There is also a timeout for route requests, such that nodes that weren't on the path between the source and destination eventually can free up the memory related to that request. Similarly to DSR, AODV also monitors for links breaking, and handles it by propagating route errors back along the route.

2.3 Attacks on Routing Protocols

When considering routing protocols, it is important to be aware of whether the nodes on the network can be trusted to follow the protocol. If the assumption is that nodes cannot be trusted, one must consider how an adversary could attack the protocol. This section will present an overview of attacks on reactive routing protocols [28]. We focus on reactive routing protocols since proactive routing protocols generally are very trust based. Nodes can easily create and share malicious routing tables with other nodes, and thus proactive routing protocols are often quite easy to attack. While no specific routing protocol is being considered in this section, many of these attacks are general ideas that can attack different reactive routing protocols.

2.3.1 Black Hole Attack

In a black hole attack, the adversary pretends to know a route to the destination of a route request. This means, that the adversary will respond to any route request they receive, by crafting a corresponding route reply. The goal of an adversary in this attack is to disrupt the attempts from regular nodes in establishing connections, by acting as a black hole and tricking nodes into thinking they have established a connection with the correct node. This attack can be somewhat thwarted by having authentication between the destination and source, since the adversary will not be able to spoof the response from the destination. However, depending on the routing protocol, if intermediate nodes update their routing table they might still be affected. One way of circumventing this, is if the intermediate nodes wait with updating their routing table until a route has been established and the source replies to the destination through the intermediate nodes. Of course, two adversaries could still perform an attack in this scenario, however it doesn't scale as well.

Another possible counter to black hole attacks is the use of baiting. Here some non-adversary nodes, try to detect black hole nodes periodically. The bait is performed by sending a request for a node that does not exist.[1]If any node responds to this request, the node that responded is clearly a black hole and can be added to a deny-list and not be contacted in the future by the originator.

2.3.2 Gray Hole Attack

In a gray hole attack, an adversary pretends to be a normal node and helps establish routes in the network. However, when those routes are established and the nodes want to use them to communicate, the adversary will stop forwarding data between them. This means that the nodes will have to establish the connection again, which of course could end up going through the same node.

[1]The node does however only send it to its neighbors, and thus the max hop count or TTL is 1.

A version of this attack is the jellyfish attack. Where nodes are not dropped but just severely delayed. These kinds of attacks are very difficult to detect, since churn on the network is already high, and it is thus difficult to spot if nodes are being dropped because of an attack or e.g. because a node moved out of range. While it is quite disruptive to the nodes involved, it only works if the adversary is actually on the fastest route between the nodes, and thus it is difficult for an attacker to reach a large portion of the network.

2.3.3 Wormhole Attack

A wormhole attack is performed by two or more adversaries at different locations in the network, who work together to establish an out-of-band connection. They can use this to create what seems like a wormhole to the rest of the network, where traffic can "teleport" to another part of the network. It is thus likely that nodes will establish communication through the wormhole. This will happen quite often since the adversaries have a speed advantage over the route requests propagating normally through the network.

In essence, this is a positive thing for the network, since nodes that would normally go through many hops now go through fewer. However, this can be used by adversaries in an attempt to disrupt the network. As an example, they can perform a gray hole attack, which is more effective than a normal gray hole attack. This is the case because a normal gray hole attack only works when the adversary is on the fastest route between source and destination, and during a wormhole attack, the attackers will be much more likely to be on that route.

2.3.4 Flooding Attack

In general, flooding attacks consist of one or more adversaries sending many packets on the network, with the goal of disrupting it due to congestion. The adversaries can craft arbitrary packets to flood the network with, which significantly increases the number of packets sent through the network. As an example, sending a single route request packet will likely cause many other nodes on the network to repeat this packet, thus amplifying the attack. This scales well for the attacker, since for each route request they send, every other node on the network will try to forward that route request to its neighbors.

Another example of a flooding attack is data flooding, where adversary nodes establish connections between each other on the network. Once these have been established, they begin sending large amounts of data between each other, thus causing congestion on the network. This will affect the nodes on the path quite substantially, but could also have the effect of dividing the network in two along the path.

2.4 Epidemic Routing

The proactive and reactive forms of routing described in section 2.1 and section 2.2, generally make the assumption that there exists a path between the originator and the target node on the network. This however is not always the case. It is not unlikely that a MANET is only partially connected at times. This has lead to the construction of a different type of routing protocols called *epidemic routing* protocols [50]. The core idea behind epidemic routing is that through nodes physically moving around on the network, messages can be delivered between separated parts of the network. This essentially works by buffering messages at intermediate nodes instead of attempting to deliver messages over a live connection. Given enough time of nodes moving around the network, messages become very likely to eventually be delivered.

Each node on the network has a buffer containing messages to be forwarded to other nodes. This can be its own messages as well as messages from other nodes. As a way of identifying the messages to other nodes, each message is given an identifier based on its hash value which determines its placement in a hash table. From this, nodes can construct a *summary vector* which is a bit field indicating which entries in its hash table contain a message. The summary vector serves as a concise way for nodes to let other nodes know which messages they are storing.

When two nodes meet they can perform an *anti-entropy session.* This consists of sharing their summary vectors with each other to deduce which messages that one node has that the other one does not. The nodes can then request these messages from each other. There can off course be collisions between hashes of message when constructing the summary vector, however a collision just means that a node does not get a message that the other node has. If collisions are somewhat infrequent, this does not have a large effect on the delivery rate.

Every node has a limit to how many messages they choose store in their buffer, and thus must prioritize messages. It can choose to do this by dropping the oldest messages in its buffer. This of course has an adverse effect on message delivery, so choosing a large enough buffer is important, as otherwise eventual message delivery becomes improbable.

2.5 Message Synchronization and Ordering

Message synchronization and ordering is important when considering how to correctly combine two partial sequences of messages into a single sequence whilst maintaining a correct order of the messages. What the term 'correct' means in this case is of course up to interpretation, however in this section we focus on causal ordering [32]. Causal order essentially means that no message can come before any message that was known by the sender when the message was created.

2.5.1 Raynal-Schiper-Toueg

The Raynal-Schiper-Toueg (RST) algorithm [44, 32], solves the problem of causal ordering, by associating each message M with a log of which messages that are causally before M. It does so by maintaining an $n \times n$ matrix clock $SENT$ for the number of nodes in the network n. Where each entry $SENT_i[j,k]$ corresponds to the number of messages j has delivered to k as known by i. Each node also maintains a $DELIV$ vector, where $DELIV[k]$ represents the number of messages sent by k that have been delivered at this node.

Every time a node i sends a message to some node j, the current $SENT_i$ matrix is sent together with the message, in order for j to know the causal location of the message. When a message M, together with the associated matrix clock ST, arrives at node i, it will be added to a buffer to first be delivered when all causally prior messages have been delivered. This is the case when $DELIV[x] \geq ST[x,i]$ for all nodes x.

2.5.2 Scuttlebutt

The scuttlebutt [51] algorithm attempts to solve the problem of causal ordering in a way that sends much less metadata, by instead restricting the order in which messages are sent instead.

Let $G = \{p, q, ...\}$ be a set of nodes forming a group on the network. Each node p maintains a local state $\mu_p \in \mathit{State} = \mathit{Node} \to \mathit{NodeState}$, which maps nodes to a node state. A node state $\mathit{NodeState} = \mathit{Message} \to (\mathit{Content} \times \mathit{Version})$ maps message identifiers to its contents and an associated version number.

A node p starts with an empty state, that is $\mu_p(i) = \bot$, for every node i. The state will over time become populated when new messages are created and nodes synchronize with each other.

When a node p wants to send a message, it is first created in the local state of p, such that $\mu_p(p)(m) = (c, v)$ where m is a randomly generated message identifier and c is the contents of the message. The version number v of the message will be 1 more than the largest version number seen. If μ_p is empty, v will take the value 1. After the message has been created, it will automatically be propagated to the other members of G when nodes come in contact with each other and synchronize their states.

Let $\max(\sigma)$ be the largest version number of any message in $\sigma \in \mathit{NodeState}$. When two nodes p and q come in contact with each other to synchronize, p starts by computing a digest $d_p \subseteq \mathit{Node} \to \mathit{Version}$ that maps each known node n to the latest version for any message. That is, $d_p(n) = \max(\mu_p(n))$, for each known node n, and all unknown nodes are mapped to 0.

p now sends d_p to q, such that q can compute a set of deltas

$$\Delta^{q \to p} \subseteq (\mathit{Node} \times \mathit{Message} \times \mathit{Content} \times \mathit{Version}), \tag{2.1}$$

that combined entails every message q knows of that p does not know of yet.

$$\Delta^{q \to p} = \{(n, m, c, v) \mid \mu_q(n)(m) = (c, v),\ v > d_p(n)\} \tag{2.2}$$

Only messages with a larger version number than the digest is included to ensure that only new messages are sent.

The deltas in $\Delta^{q \to p}$ are now sent back to p and p can update μ_p to include the new messages. Along with the deltas, q will also send a digest d_q in the same manner as sent by p, such that p can compute and send back $\Delta^{p \to q}$.

2.6 Ad-hoc Transfer Protocol

The Ad-hoc Transfer Protocol (ATP) [47] is a modification of the TCP protocol [43] to make it more suited for ad-hoc networks. Some of the problems with TCP for ad-hoc networks are slow starts. When TCP starts up, the window size is very small, and it grows exponentially until the suitable throughput is reached. Since connections in ad-hoc networks can be very short-lived, it is better to attempt to utilize the connection fully from the start.

TCP uses loss based congestion control where the window size is adjusted when packets are dropped. ATP attempts to improve this by utilizing information from the network and piggybacking the information along with the data. Each intermediate node maintains two parameters, Q_t an exponential average of the *queueing delay* experienced by packets traversing that node, and T_t an exponential average of the *transmission delay* experienced by the head-of-line packet at that node. The receiver collects the parameters and periodically sends an acknowledgement (ACK) with the information back to the sender. The sender can then use this information to calculate the optimal throughput.

In TCP acknowledgements work by sending cumulative ACKs back and when the sender window is full, no more data will be sent until an ACK is received. Unlike this ATP only sends ACKs when the receiver is missing packets, to request the missing packets to be resent. This reduces the overall traffic on the network.

2.7 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a technology for creating wireless personal area networks [49, 13]. BLE provides low power consumption compared to classic Bluetooth while keeping the same range. At the physical layer, it operates on the radio band at 2.4000 to 2.4835 GHz with 40 channels, where 3 of these are reserved for advertising and the remaining are used for data.

The BLE stack uses several layers at the link level. The Attribute Protocol (ATT) layer, defines a client-server architecture. The Generic Attribute Profile (GATT) is an encapsulation of the ATT layer, where servers can define profiles which describes what kind of data they provide. This data in a GATT profile is organized into services, which can be publicly advertised using a service UUID. The data contained in a service is further organized into characteristics. BLE defines some standard services and characteristics for specific uses (e.g. heart rate sensors), but also allows for the server to provide custom services and characteristics. A characteristic consists of some properties, a descriptor and a value. Some of the commonly used properties are *Readable*, *Writable* and *Notifiable*. Here, *Readable* means that clients can read from the characteristic, and *Writable* means that clients can write to the characteristic. The *Notifiable* property means that clients can subscribe to receive notifications when the server writes to the characteristic.

To establish a connection within a BLE network, a device needs to periodically send out advertising packets. Other devices scanning the network for advertising packets can then discover and connect to the advertising device. The device that initially sends the advertising packets is referred to as the *peripheral*, and the connecting device is the *central*. After connecting, the peripheral and central agree on a Maximum Transmission Unit (MTU). For standard smartphones, the MTU is often around 512 bytes. Once connected, the peripheral and central can use the GATT layer to act as client and server, to e.g. exchange data through characteristics.

Chapter 3

Requirements

This chapter will analyze the potential scenarios as well as the users and threat actors within them. This analysis will inform the construction of the final requirements for the protocol.

3.1 Analysis of Scenario

The primary scenario we are considering is demonstrations and protests in areas with monitored, restricted, or no internet access. This scenario will usually consist of large groups of people gathered outside in a city. The group might be somewhat stationary at a square, or could be marching through the streets of the city. In both cases individuals are not expected to necessarily be stationary, and might move alone or along with a smaller group of friends. Individuals in the protest could be interested in retaining the ability to communicate digitally with other members of the protest, even in the case where internet is unavailable.

3.1.1 Analysis of Potential User Groups

This section outlines some potential groups of users that could be affected by an internet shutdown during a protest.

Protest participant. The main user group we see, is simply a regular person who is participating in a protest. This person would likely want to communicate and keep in touch with their friends during this. At some point during the protest, they might lose track of parts of their friend group and would like to still be in contact with them. The person might also want to stay informed on the general progression of the protest, and receive information from the organizers of the protest.

Protest organizer. Another core potential user group we see are the people in charge of organizing the protest. These people are helping coordinate and organize the protest or simply just spreading the word. They might of course also need to be able to communicate with friends, but they likely would also want an ability to reach and communicate with a substantial portion of the participants in the protest. Through this, they could coordinate with other people and help provide information related to the status of the protest.

Journalist. An auxiliary and smaller user group that we consider are journalists covering the protest. Such a journalist would likely want to get information from protest participants to later publish for a larger audience. The journalist might have sources participating in the protest, that they want to communicate with. They might also meet new sources that they want to keep in contact with throughout the protest.

One common theme between the three user groups is that they most likely want to stay anonymous. They would not like for their online activity to be linked with their real life identity, and they would want their communication to stay confidential. These threats will be studied further in section 3.1.3. A common assumption we make for the three user groups, is that they would have access to a smartphone or similar device, but not necessarily have access to a safe and unrestricted internet connection.

Based on the analysis in this section, we see that the protocol must be able to facilitate one-to-one communication. That is, it must allow two participants on the network to send messages to each other, and must do this without the use of static infrastructure. It can thus only rely on a network consisting of mobile devices.

From the analysis we also see that the protocol must facilitate group communication. It must allow for participants to create groups and invite other participants to the group, wherein the members can then exchange messages.

3.1.2 Threat Actors

We consider the major threat actors to be governments in nation states that want to deter protests and demonstrations by shutting down or restricting digital communication infrastructure. These threat actors could consist of various government agencies such as law enforcement, intelligence services and the military. We expect the threat actors to be advanced adversaries, which have the resources and motivation to attempt to attack alternative ways of digital communication such as MANETs. The goal of the threat actors could be to gain information about who is sending data on the network to whom, and the contents of this data. They might even want to impersonate people on the network in an attempt to infiltrate groups and extract further information. Another goal could be for them to simply disrupt the network completely.

In relation to the capabilities of the threat actors, we assume that they have access to large computational power, but not quantum computers. This means that we consider them incapable of breaking contemporary cryptographic standards. We also assume that they in general do not have a global view of the network, and instead only have a local view. However, we do accept that threat actors could plant multiple adversaries in a protest to attempt to approximate a global view with high effort. We also assume that the threat actors could place multiple adversary nodes on the network in an attempt to attack or disrupt it. The threat actors might also have access to technology that can jam network activity entirely, however we assume this jamming to be localized to limited regions and not be persistent.

3.1.3 Threats

The following threats are considered when specifying the requirements of the protocol, in order to minimize the potential damage imposed by such threats.

Data leakage. The threat of data leakage entails leakage of private data on the network, for instance if data is not sufficiently encrypted. The extent an adversary is able to compromise encryption is the key factor when minimizing the possibilities for data leakage.

Intelligence gathering. The threat of intelligence gathering entails getting information about the network, but is not necessarily the private data exchanged on the network. Instead, routing metadata from the network could be enough to decrease anonymity. With enough exposed metadata, an adversary might be able to build a social graph of the users on the network. Even if the social graph does not leak personal information about an individual, the relationship between nodes is still a compromise of anonymity.

Impersonation. An adversary might be able to impersonate other users on the network and spread misinformation. Impersonation should never be possible without an explicit identity compromise. If an adversary were to compromise the identity of another user, the potential damage the adversary should be able to cause, should be as limited as possible.

Disruption. The network should be resistant to disruption such that it avoids becoming unusable for communication. Disruption could be caused by an adversary by overloading the network with traffic in order to cause too much congestion for genuine traffic to pass. An adversary should also not be able to craft malicious network packets that cause harm or drain computational resources of the receiving nodes or the network in general.

3.2 Security Requirements

It is crucial that the application is secure by design, as security flaws could result in serious consequences for the intended users of the platform. This section describes the security requirements needed in order to prevent or minimize the threats mentioned in section 3.1.3.

Confidentiality. *Messages can only be read by communicating parties.*
Confidentiality of messages sent between communicating parties must be guaranteed. This is needed in order to prevent the threat of data leakage.

Authentication. *Users prove they are who they say they are.*
Authentication between communicating parties must be guaranteed. However, other nodes in the network who help forward messages between contacts do not have to be authenticated. Without proper authentication, users risk the threat of impersonation.

Integrity. *Messages cannot be altered.*
Integrity of messages sent between communicating parties must be guaranteed. It is a prerequisite for trusting the authenticity of messages. Without integrity, users might risk impersonation, and it could make the network susceptible for disruption.

Availability. *The network should always be available.*
It must be hard for adversaries to disrupt the network. It might be infeasible to completely prevent disruption of the network, but as long as the required effort in doing so scales

linearly with the amount of disruption, we still consider the network to be resistant to disruption.

Anonymity. *Users cannot be identified.*
It is crucial that users of the technology cannot be identified and that network data cannot later be used as proof of their presence. Therefore, device IDs must be rotated regularly, and personal information must never be shared without proper encryption. Anonymity is necessary in order to prevent the threat of intelligence gathering.

Chapter 4

Protocol Design

This chapter describes the Starling protocol while motivating and discussing the design choices. The technical specification of the protocol can be found in appendix A.

The protocol is split up into layers similar to the OSI model [25] used by the Internet. There are four layers in the protocol: packet, network, transport, and application layer. Each layer encodes the data for the next layer in the stack. While the protocol specifies these four layers, a link layer is needed to enable communication between pairs of nodes.

4.1 Link Layer

The link layer can in principle be implemented on top of various communication forms, as long as peers can communicate bidirectionally. The channel may be insecure and public. The link layer should attempt in best-effort to deliver packet in-order, however packets may arrive out of order, be dropped or corrupted. Each node should be able to differentiate between their own peers and identify them for the duration of a session. This could be achieved using MAC addresses or similar. One node does not need to be identifiable across different nodes. For example a node may choose to identify themselves with a different address for each peer in order to increase anonymity.

One of the prime candidates for the link layer is Bluetooth Low Energy (BLE) since it is supported by most Apple and Android smartphones. It is also very flexible since it can run while the phones are in background mode and automatically connect when the phones near each other. The combination of its widespread support and flexibility makes it a great candidate for our use case. We have thus chosen to focus our protocol design around BLE. The biggest drawbacks of using BLE for the link layer is its limited bandwidth and no proper multicast in most smartphones. The lack of general multicast makes route requests flooding more expensive, since each route request must be forwarded individually to the other peers. However, the benefits far outweighs the drawbacks compared to the other available smartphone technologies.

4.2 Packet Layer

The packet layer is responsible for splitting up and bundling packets into a fixed byte window, so they can be sent over the underlying link layer. As an example, the size of BLE packets, also referred to as maximal transmission unit (MTU), can be negotiated between devices but as stated in section 2.7 it is usually limited to around 512 bytes for

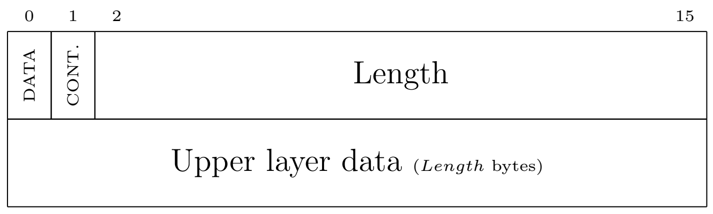


Figure 4.1: Binary format of the packets on the Packet layer.

smartphones. This means that if an upper layer wants to send a packet that is longer than this, it must be sent over the link layer in multiple packets.

The format of packets in the packet layer can be seen in fig. 4.1. Each packet starts with a 16 byte header. The first bit indicates whether or not this packet contains data. In almost all cases, this is 1. If the bit is 0, the rest of the data is discarded. The second bit is the continuation flag. If this flag is 1 it means that the data continues into the next packet, and thus should be combined with this one. If the continuation flag is 0, it means that the data contained in this packet is the last data of the upper layer packet. Following these two bits, is a length field, which indicates how many bytes of data are included in this packet (disregarding the header).

Essentially, an upper layer packet is split into however many packet layer packets is needed. If more than one packet layer packet is needed, all except the last of these packets has their continuation bit set to 1. The receiever of these packets can then stitch the packets back together to reconstruct the original upper layer packet.

4.3 Network Layer

The network layer is responsible for finding end-to-end routes between two *contacts* on the network, potentially through multiple intermediate nodes. The term contact is used to refer to nodes that have authenticated and linked with each other, the term will be used throughout the rest of the thesis. How nodes link with each other to become contacts will be described in section 4.3.3. The general goal of the network layer is to establish an encrypted and authenticated session, through which the two contacts can communicate.

4.3.1 Routing

When deciding which routing protocol to use, it is important to choose one that aligns with the requirements specified in chapter 3. Due to how dynamic the network topology is expected to be, we choose to use a protocol of the reactive routing type. Within this category our main focus is DSR and AODV, however we specify a custom routing protocol that draws inspiration from both of these. We will now discuss problems that DSR and AODV have in relation to our requirements.

One of the major problems with both DSR and AODV is in relation to anonymity, since nodes need an address which is used for routing in the network. When a node wants to establish a session with another node, they have to include the address of the node within the route request. This generally breaks anonymity, and could allow for a threat actor to create social graphs.

Problems with DSR. A problem specific to DSR is that since route requests contain a list of the current path, the packet is not a fixed size but can grow substantially. Due to the focus of BLE as the link layer, it is more efficient to stay within the max packet size of BLE. This is because route requests are likely to take up a big part of the network traffic, and reducing its size will potentially reduce overhead. The fact that route lists are included in route requests also gives a potential attacker a lot of power in regards to modifying the list, and potentially creating loops or unnecessarily long routes.

Problems with AODV. One of the problems with AODV is that intermediate nodes are allowed to reuse old routes if they are deemed fresh enough, when replying to a route request. This means that the target node is not actually reached, and thus cannot authenticate itself in the route reply. This opens up for black hole attacks, since adversary nodes could reply to all route requests with bogus route replies.

Generally the idea from AODV of storing parts of the route locally in the intermediate nodes on a path, is quite resistant to tampering with the path since the control over the route is more distributed than in DSR. An intermediate adversary node can of course always choose to not forward packets, however this essentially has the same effect as if they disconnected from the network, which the protocol should be able to handle.

4.3.2 Routing Protocol Design

The specification for the network layer of the Starling protocol can be found in appendix A.6.3, it is presented here with a focus on routing. The network layer consists of four packet types as briefly described below.

RREQ A route request is flooded through the network in order to find contacts. It contains a *request ID*, a *time to live (TTL)*, an ephemeral public key, as well as a *contact bitmap* encoding the intended recipients.

RREP A route reply is propagated back after a route request finds a matching contact. It includes among other fields, a randomly generated *session ID*, an ephemeral public key, an authentication tag, as well as the request ID from the corresponding route request. It can also optionally contain piggybacked application data.

SESS A session packet is used to carry data between two contacts that have established a session. It includes the session ID, an authentication tag, the length of the data it carries as well as the actual encrypted data.

RERR A route error is propagated through a route in order to inform the nodes along it, that the connection no longer exists. It only includes the session ID of the broken session.

When some node i wants to communicate with another node j on the network, it needs to establish a session between the two. Node i can attempt to do this by sending a route request packet (RREQ) to its neighbors. The request ID contained in the route request is randomly generated and serves as a way of identifying this route request on the network.

The neighbors that receive the route request will decrement the TTL and forward the requests to their neighbors. Each node maintains a *routing table*, in which they make an entry for the route request containing the request ID along with the address of the *source node*, which is the node it received the request from. If a node receives a route request with a request ID they already have in their routing table, they will ignore the request.

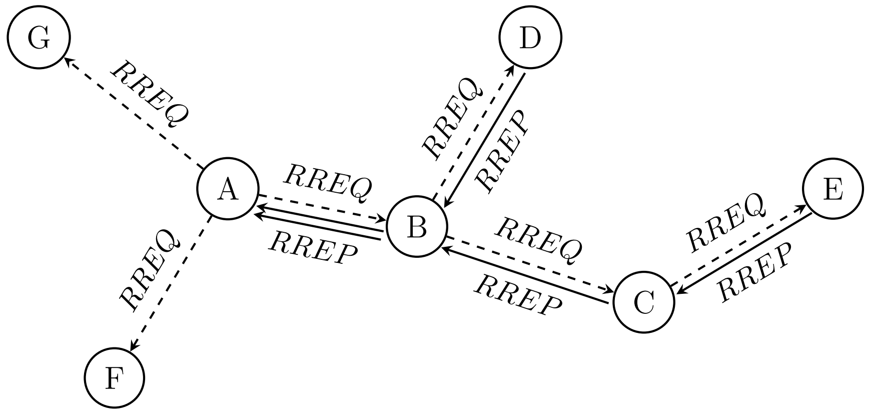


Figure 4.2: Flooding of a route request in the network layer.

This essentially creates a tree structure in the network for each route request, where any node in the tree knows from which node they received the request.

Nodes will continue spreading the route request until its TTL reaches 0. If node j at any point receives the route request, it will identify that it is an intended recipient using the contact bitmap. How this works will be addressed in section 4.3.5. Node j can then respond by sending a route reply packet (RREP) to the node that it received the route request from. j crafts a route reply by generating a session ID as well as an ephemeral key, and includes the request ID from the route request. If j has data it wants to send along with the route reply, it includes this as piggybacked data. Before responding with the route reply, node j creates a new entry in its *session table* which among other fields includes the session ID and the source node, in order to associate future packets with this particular session. When the intermediate nodes receive the route reply, they also create a new entry in their session table, which includes the session ID along with the two neighbors on the path – the node they initially received the route request from and the one they received the route reply from.

Figure 4.2 illustrates a possible scenario, wherein A has sent a route request in an attempt to establish sessions with D and E. A has transmitted the route request to each of its neighbors B, G, and F. Only B has neighbors other than A, so it forwards the route requests to D and C, after which C further forwards it to E. Both D and E have received the request and from the included bitmap and they each deduce that they are an intended recipient. This causes them to respond with a route reply which they send back along the path in reverse. A receives the two route replies and the two sessions are established.

Once a session has been established the end-point nodes can send encrypted data to each other using session packets (SESS). How the data is encrypted will be described in section 4.3.4. The session packets can be sent by either end-point node and are forwarded along the established path by the intermediate nodes.

If a connection between two nodes along the path is broken at any point, these two nodes should each send a route error packet (RERR) in the opposite direction. The route error packets are then forwarded along each half of the path, and nodes on the path remove the entry for that session from their session tables. When the end-points receive the route error they each know that the session is broken, and can attempt to establish a new session by issuing new route requests.

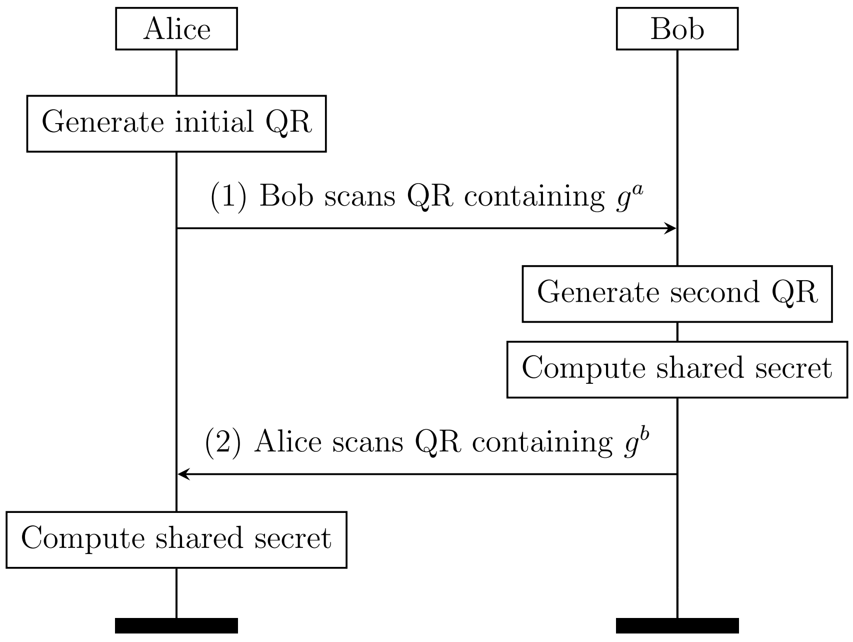


Figure 4.3: Sequence diagram showing the process of linking contacts using two QR codes.

4.3.3 Linking

For two nodes to establish a secure session, they need to be contacts first. This involves that they agree on a shared *contact secret*, which is both used for authentication and as part of the encryption process.

Suppose that two people, Alice and Bob, want to become contacts. The general idea is to use Diffie-Hellman key exchange [18] to obtain the contact secret. Alice generates a public key g^a and Bob generates public key g^b, which they exchange with each other. The protocol technically allows the key exchange to take place through any medium, however we propose a key exchange through QR codes. This version of the contact linking process is illustrated in fig. 4.3.

If Alice is the initiator, she generates a QR code which contains g^a, and this QR code is shown on her device. Bob then scans this code to obtain g^a, after which Bob generates a QR code containing g^b. Once Alice scans it, the key exchange has been performed and both parties can calculate the contact secret as g^{ab}.

One of the reasons for choosing QR codes on devices as the medium for key exchange is to ensure authentication. Since each party has obtained the public key of the other party directly from their device, it severely limits the possibilities of an attacker. For example, a man-in-the-middle attack would be very difficult to execute, since the attacker would have to convince each party to scan the attackers QR codes instead of each others.

We choose to use the X25519 ECDH standard [8] for performing the Diffie-Hellman key exchange, as the sizes of the keys are relative small while still maintaining a high security level of at least 128 bits. The algorithm is also very fast, which together with the small keys makes it very suitable for this use case.

Groups

Groups also use the concept of a contact secret, however the process of establishing it is different from when linking between two contacts. To create a group, a node simply generates a random contact secret. This contact secret serves both as a membership token and as part of the encryption key for messages in the group. A node invites another node to the group simply by sending the contact secret of the group to them. This of course requires that both nodes are already contacts with each other, such that the message is encrypted and the contact secret of the group is not leaked.

This system for groups is a bit primitive and lacks some features. Any node that has obtained the contact secret of the group is seen as part of the group and can read messages in the group. This means that there is no way to revoke membership of a group from a node. The network layer also does not provide the ability to differentiate between members of the group in an authenticated manner. It is up to the upper layers to provide this if needed (see section 4.6.1).

4.3.4 Secure Session Establishment

The goal of the session establishment is to obtain a confidential and authenticated channel between any two parties, say Alice and Bob, without leaking any identifying information to the rest of the network.

Let a, b be static private keys of Alice and Bob respectively, with $A = g^a$, $B = g^b$ as the corresponding public keys. It is assumed that Alice knows the public key of Bob and vice versa from the linking process, as described in section 4.3.3. This means that they both have the static contact secret. In addition to the static keys, both Alice and Bob generate an ephemeral key pair for each session. Alice generates $(x, X = g^x)$ and Bob generates $(y, Y = g^y)$. The ephemeral public keys are shared between Alice and Bob during the session establishment, since they are included in the route request and the route reply. With the ephemeral keys they can generate the *ephemeral secret* for the session, as Y^x from the perspective of Alice and X^y for Bob.

Both the contact secret and ephemeral secret are used to construct the *session secret* (s), using the Unified Model [11, 5], which is used for encryption and authentication. For the instance where Alice is the initiator of the route request, the session secret is calculated as $s = KDF\left(B^a \mid\mid Y^x\right)$ from the perspective of Alice, and as $s = KDF\left(A^b \mid\mid X^y\right)$ from the perspective of Bob.

Here $||$ is the concatenation operator. The key derivation function (KDF) is used for deriving proper key material from another entropy source. In this case we use it to ensure we get a session key of the correct size, that combines the entropy from both the contact secret and the ephemeral secret. For this protocol the HKDF scheme [31] is used for key derivation.

The shared session key will be used for all encryption of all session packets exchanged between the two parties for the duration of this session. The AES-GCM [34] standard is used for symmetric encryption. AES is the defacto standard for symmetric encryption, which makes it an obvious choice. Using the Galois/Counter Mode (GCM) as the mode of operation, provides authentication and integrity along with the confidentiality provided by AES itself.

Analysis of the scheme

As mentioned, the design of the session establishment scheme uses the Unified Model [11, 5]. One shortcoming of this scheme is that it does not provide *key-compromise impersonation*. Suppose that Alice and Bob have linked by performing a Diffie-Hellman key exchange. Now imagine that some malicious user Eve manages to compromise the long-term private key of Alice. Clearly, Eve can now impersonate Alice when communicating with Bob, however without the property of key-compromise impersonation, Eve is also able to impersonate Bob when communicating with Alice.

This scheme provides implicit key authentication, which ensures that each party can guarantee that no other entity than the other party knows the session secret. However, it cannot guarantee that the other party does in fact know it. This means that it does not guarantee *forward secrecy*, where even if the static private key is compromised after the session has ended, the data exchanged over the session is still safe.

Imagine a case where Alice and Bob want to establish a session on a path with a malicious intermediate node Eve. Alice sends a route request along, when Eve receives it, she replaces the ephemeral key in the route request with one where she knows the private key, before forwarding it along to Bob. When Bob receives the message, he unknowingly computes a session secret based on the ephemeral key of Eve. However, Eve cannot compute the session secret as she does not know the static private key. She can however hold on to the encrypted messages in hope of later compromising the static private key, at which point she will be able to read the message. This breaks forward secrecy.

Bob can however obtain forward secrecy if he waits to send any confidential data until an authenticated message from Alice has been received first. That way *explicit key authentication* is obtained, where each party is guaranteed that the other party possesses the session secret. And with that forward secrecy is also obtained.

This scheme has potential for improvement, which is out of scope for this project. We believe that the double ratchet algorithm developed for the Signal protocol [41] and used by many of the major messaging platforms including Signal itself, WhatsApp [53] and Facebook Messenger [19], could potentially be used as the session establishment scheme, in order to further increase the security of the protocol.

Anonymity

It is important to ensure that no identifying information is revealed during the key exchange, to keep the anonymity of the participating parties. This means that the route request must encode the recipient(s) in a way where none other than the recipient can know if the packet is targeted them. This is achieved by the contact bitmap which will be presented in section 4.3.5. Likewise, when a route reply is initiated, no identifying information of Alice or Bob must be leaked. When a route reply is initiated, all identifying information is encrypted using AES-GCM with the session secret, and the receiver of the route reply must attempt to decrypt the message. They do this by computing what would be the session secret s for each static contact secret that they know of. If the decryption is successful for a given contact secret, they know that it was this contact that responded to their route request.

Both the route request and the route reply also contain an ephemeral public key, however this does not leak anything since these are randomly generated for each new session.

Algorithm 4.1 Primitive approach for encoding a contact bitmap.

```
function PrimitiveContactBitmap($\mathcal{C}$)
    CB[$0..n-1$] $\leftarrow$ a new array with all elements initialized to 0
    for each $c \in \mathcal{C}$ do
        $i \leftarrow h(c)$
        CB[$i$] $\leftarrow 1$
    end for
    return CB
end function
```

4.3.5 Contact Bitmaps

To achieve anonymity, the network layer has no notion of a network addresses, in the sense that route requests do not include some address of the intended receiver. Instead, the mechanism used for indicating which contact a node wants to establish a session with is what we call a *contact bitmap*. This mechanism allows us to encoded multiple different contacts anonymously in a single route request. This leads to higher efficiency, since nodes can establish sessions with multiple contacts from a single route request.

Each route request contains a contact bitmap which is used to identify whether the receiver of the request is encoded as an intended receiver by the sender. If they are, a route reply will be sent back in order to establish a new session. The challenge is to encode many contacts in a small amount of bytes and in a way that is anonymous and also resistant to various attacks.

Primitive approach

A primitive approach for accomplishing this could be to use some derivative of the contact's shared secret to generate a hash. This hash can then be used to find a single bit in the bitmap which should be set to 1. When a node then receives a route request, they can go through each contact in their contact list and compute the corresponding index of the bit. If the bitmap has this bit set, that indicates that they might be an intended receiver. This can of course be a false-positive due to collisions, however if this rarely happens it is not a big problem, since proper authentication will be done later anyway. If the bit in the bitmap is not set, then the route request was not meant for them.

A more precise definition of this primitive version of a contact bitmap will be given now. Let $\mathcal{C}$ be a set of contacts that this node has linked with. Let $h(c)$ be a function which maps each contact $c \in \mathcal{C}$ to an index i such that $i \in \{0 \mathinner{..} n-1\}$ where n is the size of the contact bitmap. The function $h(c)$, uses a derivative of the contact secret to ensure that the mapping is not publicly known. The function PrimitiveContactBitmap for computing this version of a contact bitmap can be seen in algorithm 4.1.

One problem with this approach is that it requires the contact bitmap to be quite large if we want to avoid false-positives. It can be made more efficient by using the concept of Bloom filters [12]. A Bloom filter works by the same idea, except multiple hash functions (eg. 5) are used for each element. A node would in this case need to identify whether the 5 indexes that the hash functions produce are all set to 1, and only in this case will the node respond. The idea behind this is to minimize the risk of collisions, and thus minimize the required space usage.

Attacks

Fingerprinting is one of the major problems with the simple approach. Since the shared secrets always hashes to the same bits, bitmaps from the same device will often look very similar. This allows an attacker to easily recognize individual nodes from their contact bitmap and thus would allow them to build social graphs of the network.

This could be partially solved by seeding the hash function and using a new seed for each bitmap. The route request could include some seed along with the bitmap to allow the receiver to seed their hash function correctly. The idea here is that that multiple bitmaps from the same sender will be completely different even if they contain the same encoded contacts, since the bits are hashed differently each time. It would however still be possible to count the number of contacts to get a rough fingerprint instead. Our proposal attempts to mitigate fingerprinting even further.

Another attack vector is the idea of a reply attack. Reply attacks could be performed by an adversary by setting all bits to 1 in the bitmap. This would mean that every node that receives the route request will identify itself as a possible receiver, since all of their contacts would match with the given bitmap. Every node that received the route request would thus respond with a route reply, which would add a lot of unnecessary traffic to the network. Our proposal also attempts to mitigate this attack.

Our proposal

Our proposal uses a modified version of a Bloom filter. Instead of inserting a 1 at every index the hash functions produce, we alternate between setting the bit as 1 and 0. Then the recipient can calculate the same bits and check if all of them match in the bitmap.

The main problem with this approach is that two contacts might produce two different bit values for the same index in the bitmap. That is, one might require a bit to be 1 and the next requires the same bit to be 0. These conflicts mean that only a limited amount of contacts can be encoded in a bitmap at once. However, the risk of collisions can be made low enough that the bitmaps still function well (see handling conflicts and performance on page 27). A contact could also have a collision with itself, since multiple hash functions are used. However, this is simply solved by choosing the next available bit instead of the one specified by the hash function.

In order to avoid fingerprinting, we use a selection of hash functions as described under attacks on page 25, and include a seed in the route request which indicates which hash function was used. In practice, the seed sent along with the route request is the request ID. To avoid an adversary detecting how many contacts we have by analyzing the bitmap, we simply randomize the bits that are not specified by the encoding process. That is, if a bit is not set to either 0 or 1 after encoding all contacts, this bit will be randomly set to 0 or 1.

Our proposal also curbs the reply attack significantly, since setting every bit in the bitmap to 1 will not have any effect. This is the case because no one will reply since every node will need some of the indexes they are looking at to be 0. The best an attacker can do is to set a random bitmap of 1's and 0's, however this doesn't yield a good attack as long as we choose the number of hash functions used for insertions to be appropriate. As an example, using 12 hash functions (12 bits per contact) for each insert will mean that a false positive will only happen at a rate of $\frac{1}{2^{12}} < 0.03\%$ per contact.

We propose the function ENCODECONTACTBITMAP which can be seen in algorithm 4.2. This function encodes contacts into a contact bitmap. Here, the first argument $\mathcal{C}$ repre-

Algorithm 4.2 Encoding a contact bitmap.

```
function EncodeContactBitmap($\mathcal{C}, s$)
    CB[0..$n-1$] ← a new array with all elements initialized to nil
    for each $c \in \mathcal{C}$ do                                  ▷ Outer loop
        $T \leftarrow$ ContactBits($c, s$)
        for $j \leftarrow 0$ to $m - 1$ do
            $i \leftarrow T[j]$
            if CB[$i$] $\neq$ $nil$ and CB[$i$] $\neq j \mod 2$ then
                continue Outer loop                    ▷ Conflict with another contact
            end if
        end for
        for $j \leftarrow 0$ to $m - 1$ do
            $i \leftarrow T[j]$
            CB[$i$] $\leftarrow j \mod 2$                             ▷ Set contact bit
        end for
    end for
    Set remaining $nil$ bits in CB to random values
    return CB
end function
```

Algorithm 4.3 Computing the bits for a given contact.

```
function ContactBits($c, s$)
    T[0..$m-1$] ← a new array with all elements initialized to nil
    for $j \leftarrow 0$ to $m - 1$ do
        $i \leftarrow h_j(c, s)$
        while $T$ contains $i$ do
            $i \leftarrow i + 1 \mod n$                ▷ Possible conflict within contact
        end while
        T[$j$] $\leftarrow i$
    end for
    return $T$
end function
```

sents a list of contacts, where the order of the contacts determines their priority, since contacts early in the list are more likely to be successfully encoded than later entries. The second argument s is the seed. The function $h_j(c, s)$ maps a contact c and a seed s to the corresponding index in the contact bitmap. Here the subscript j indicates that h_j is a function selected from a family of such functions. These functions are essentially hash functions in the way that they must be one-way functions. The variable m is the number of hash functions to use for each contact. The algorithm utilizes the function ContactBits as seen in algorithm 4.3. It selects the m bits that should set to either 0 or 1 when inserting a single element. If there is a conflict within these m bits, this is handled by selecting the next available bit. After the call to ContactBits, the EncodeContactBitmap algorithm goes through the returned list of indices and checks if these are in conflicts with any of the already assigned bits in the contact bitmap. If there are no such conflicts, the bits at the indices in the list are assigned to alternatingly 0 and 1.

Algorithm 4.4 Decoding a contact bitmap.

```
function DecodeContactBitmap($BC, \mathcal{C}, s$)
    $L \leftarrow \emptyset$
    for each $c \in \mathcal{C}$ do                              ▷ Outer loop
        $T \leftarrow$ ContactBits($c, s$)
        for $j \leftarrow 0$ to $m - 1$ do
            $i \leftarrow T[j]$
            if CB[$i$] $\neq j \mod 2$ then
                continue Outer loop                          ▷ Discard current contact
            end if
        end for
        $L \leftarrow L \cup \{c\}$                                  ▷ Found contact
    end for
    return $L$
end function
```

Handling conflicts

As mentioned, with this modification to bloom filters, we are no longer guaranteed that we can insert an element because it might collide with another element at one or more indexes in the bitmap. The simple solution is just to skip one of the contacts that are in conflict and use a "best effort" approach. Alternatively the contacts that were in conflict can be attempted to be inserted in another route request.

If the contact bitmap is deemed to not contain enough contacts or to not contain the right contacts, a node can simply choose to execute the algorithm again with a new seed. This can be done in order to get a more reliable result since having multiple unlucky attempts in a row is less likely. It is up to the node to decide how many retries it should perform. It is important to note that retries will only add more computation for the sender, and not for the recipients for whom the seed will be known, and they will be able to match the contact efficiently.

The function in algorithm 4.4 shows how a receiver can decode a contact bitmap, to identify whether they might be a receiver. As part of the route request, the receiver both has access to the bitmap array BC as well as the chosen seed s. From their own state, they also know which contacts they have linked with. This is represented by the set $\mathcal{C}$. It returns a set L which contains the contacts that are associated with the contact bitmap. As can be seen, the time complexity for decoding a contact bitmap is linear in the size of $\mathcal{C}$, since m is a fixed constant.

Encoding performance

When designing the bitmap, there are multiple parameters that must be chosen to ensure that it is usable, robust and provides anonymity. These parameters also affect how efficient it is to encode contacts into a bitmap. One of the big decisions is the size of the bitmap n. That is, the number of bits in the bitmap. The larger n is the less chance of collision, but a larger n of course also increases the size of route request. A choice of $n = 2048$ bits (256 bytes) has been made to be able to keep route requests within the 512 byte max size of a Bluetooth packet, while still allowing space for other fields in the route request packet. Another parameter to consider is the size of the seed. The size of the seed does

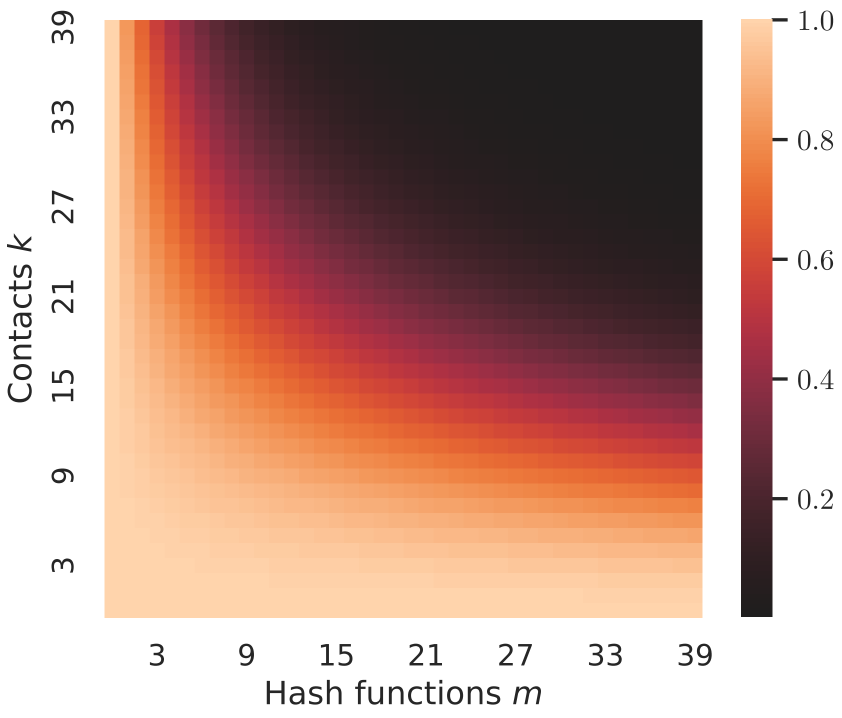


Figure 4.4: Heatmap showing the chance of a collision when inserting k contacts in a bitmap with m hash functions used for each contact.

not affect the chance of collisions, but it has to be large enough such that a single seed is not used multiple times. A seed size of 8 bytes has been chosen for the protocol.

We also need to consider the number of hash functions m to be used for each insert. This decision needs to strike a balance between being resistant to reply attacks, while still allowing us to insert enough elements. That is, we want the false positive rate to be low enough that it does not affect the network negatively. At the same time, we want to maximize the parameter E, which represents how many contacts we are expected to be able to encode in a given bitmap. In a way, this can be seen as an evaluation of how well the parameters we have chosen perform.

The effects on E for different choices for m has been visualized in the heatmap seen in fig. 4.4. We essentially consider how varying the number of bits m to be inserted and varying the number of contacts we attempt to insert k, affects the chance of the inserts being successful. If the chance is low, we will have to try many times, and if it is high, we will have to try few times. It fundamentally represents the expected amount of work to encode the contacts in the bitmap. For construction of the heatmap, the following expression, eq. (4.1), was used as a pessimistic estimate of the chance of successfully encoding k number of contacts in the bitmap.

$$\prod_{i=0}^{k-1} \frac{n - i\frac{m}{2}}{n} \tag{4.1}$$

While the heatmap seems to indicate that m should be as small as possible, we of course need to avoid setting it too low, as to avoid false positives. If a node receives a random bitmap, the probability of a false positive is $\frac{1}{2^m}$ for each of the contacts that the node has. This means that whenever a node receives a route request, there is a probability of $|\mathcal{C}|\,\frac{1}{2^m}$ of a false positive. A false positive would of course mean that the node would simply send a wrong route reply, which is not a problem as long as it only happens to a limited number of nodes. We choose m to be 12 for the protocol. We see this as a good middle ground, where it is not too difficult to encode a reasonable amount of contacts in a single bitmap, while still keeping the rate of false positives low. For example, a node would be expected to encode 25 contacts in a bitmap in 3 attempts. Also, with this choice of m, a node with 80 contacts would have its probability of false positive be less than 2% for a given bitmap.

Anonymity

One of the main features regarding anonymity in the version of contact bitmaps given in EncodeContactBitmap, is that fingerprinting is eliminated. That is, it is infeasible to infer whether two contact bitmaps have been created by the same node. Each node uses a new seed each time they construct a new bitmap which ensures that they are different. Also, the problem from PrimitiveContactBitmap where it is easy to identify roughly how many contacts a node has, has also been avoided. This is the case since nodes assign both 0 and 1 to bits when encoding contacts into the bitmap, and that the remaining bits are randomly assigned.

Another feature related to anonymity is that with contact bitmaps, nodes do not have a network wide address. They of course need to have a link layer address used for their neighbors, however this address is not required to stay the same. In fact, most smartphones rotate their BLE address every 15 minutes or so to increase anonymity. Instead of addresses, this notion of only allowing the node that knows the contact secret to identify that the route request was meant for them, means that the route request does not leak information about the intended receiver.

4.4 Transport Layer

The transport layer becomes relevant once a session has been established on the network layer. All data sent over the session is managed by the transport layer. The responsibility of the transport layer is to provide reliability, such that if a packet is dropped, an attempt will be made to resend it. It is also the responsibility of this layer to terminate the session after a timeout if the other party suddenly stops responding. Lastly it guarantees that data packets are delivered in a first-in-first-out (FIFO) order.

The layer consists of two packet types as described below. For more details refer to the specification in appendix A.6.4.

DATA A data packet carries data to the recipient.

ACK An acknowledgement packet, acknowledges one or more of the other party's data packets.

4.4.1 Data Packets

A data packet carries application level data together with a sequence number. When a participant sends a data packet, the associated sequence number is computed as one larger than the sequence number from the previous packet they sent.

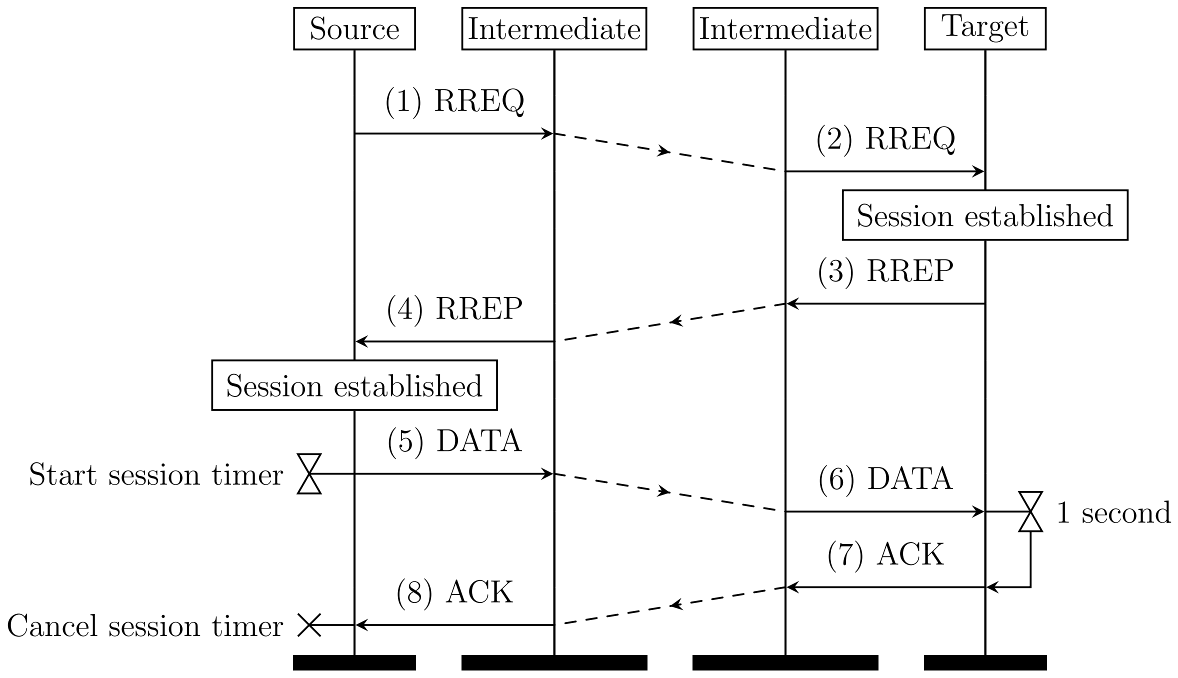


Figure 4.5: Sequence diagram showing the process of establishing a session, and how the session can be used to for data communication afterwards.

When a data packet is received, the recipient will wait to deliver it if the sequence number is more than one larger than the previously delivered packet. This ensures that data packets are delivered in a FIFO order. If a packet cannot be delivered yet, it is stored in a buffer. Once a new packet arrives and can be delivered, it can cause other packets in the buffer to be delivered if they have sequence numbers immediately following it.

4.4.2 Data Acknowledgement

When a data packet is received, the *acknowledgement timer* (ACK timer) will be started if it is not running already. This timer times out after 1 second and once it times out, an ACK packet is sent back to the sender of the data packet. The ACK packet contains the highest sequence number of any data packet received so far, along with a list of missing sequence numbers. That is, all sequence numbers smaller than the highest, where the associated data packet has not been received yet. When a new data packet is later received, the timer will be started again and the process continues.

When an ACK packet is received, the data packets mentioned in the list of missing sequence numbers will be retransmitted. In order to do this, the sender must keep a list of data packets in transit. When a data packet has been acknowledged by an ACK packet, it can be removed from the in-transit list.

If an ACK packet has not been received after t_{SESS} seconds from a data packet has been sent then the session will be closed as it is considered to have timed out. This is kept track of by the *session timer*. The duration of t_{SESS} must be longer than one second, and should be chosen such that it is a bit longer than the expected round trip time plus one second. Once a session times out, a route request can be sent in an attempt to establish a new session. If the ACK packet is received before the session times out the session timer is reset, as seen fig. 4.5 where the data packet (5) starts the session timer, which is canceled when the ACK packet (8) is received. However, in the case that there are still other data packets in transit that have not been acknowledged yet, the timer should not be cancelled.

As mentioned in section 4.3, a route reply can piggyback extra data (third packet in fig. 4.5). The transport layer uses this to attach the first DATA packet of the participant in the route reply. The purpose of this is first to let the participant include relevant data (if they have any) as soon as possible. This is especially useful for carrying digests when performing synchronization as will be described in section 4.6. If the participant does not have any relevant information to include, sending an empty DATA packet, has the benefit of facilitating a timeout to cancel the session if the route reply is never received.

4.4.3 Congestion Control

As described in section 2.6, transport protocols for ad-hoc networks have to handle some aspects of the network differently than a traditional transport protocols such as TCP. One example of this is congestion control.

Due to time limitations it was decided that congestion control was to be omitted as a feature of the protocol. It is our belief that congestion control is less important for our protocol than transport protocols for traditional networks. If the network is congested in certain areas, new routes would naturally avoid these areas since the route requests would not be propagated very quickly through these. This would lead to the creation of routes that might not take the most direct path, but would instead go through nodes that have the capacity to handle the traffic. Of course this argument only relates to newly created routes, but since the network topology is expected to be very dynamic, sessions will most likely not be long-lived. This means that nodes are likely to lose connection and reestablish a session through a different route, especially if they experience packet loss due to congestion.

While congestion control was not prioritized, we present a short discussion of the problems and ideas related to it. A practical problem with congestion control for the protocol is the limited information about queuing and transmission delays available when the protocol is running on an average smartphone. Thus, the strategy ATP has for congestion control, where intermediate nodes keep track of delays, is not viable if these delays are not possible to extract from the device. Congestion control could of course be done in a loss based fashion, where the two end-nodes control the number of packets they send based on the number of acknowledgements they receive. However, as mentioned in section 2.6 this approach risks under-utilizing the potentially quite brief connection between the nodes.

4.5 Application Layer

The application layer is responsible for sending the application specific data over a session. The layer is in essence very simple. It defines the first byte to specify what kind of data is being sent, such that the recipient knows how to interpret the data. Currently, only two values have been reserved.

Application specific packet `0x01`. This value is used to specify that the format of the packet is outside the scope of the specification, and it is therefore up to the implementation to identify and handle the packet properly. It is recommended that the specific implementations add their own way to identify the packets from other packets, for example by starting the packet with a small sequence of bytes that signify the packet type.

Synchronization packet `0x02`. This value specifies that the rest of the packet should be interpreted as a synchronization packet, as is described in the specification in appendix A.10. This protocol extension describes a common way to implement message synchronization, which also supports groups.

The intention with this structure is that new extensions could be added to the protocol specification in the future. If a common use case emerges, an extension to the specification can be added and a byte identifier can be reserved in order for multiple implementations to use it.

4.6 Synchronization Extension

In order to ensure that all messages will eventually be delivered, it is not enough to solely rely on the underlying session provided by the protocol. A session only provides reliability as long as it is active. When it is broken, all messages in transit will be forgotten and thus are not guaranteed to be delivered. To solve this, a message synchronization layer is introduced that builds on top of the underlying sessions, which ensures that after a session breaks, any undelivered messages will be retransmitted whenever a new session is established. The specification for the synchronization extension can be seen in appendix A.10.

Another role of the synchronization layer is to ensure that messages are ordered correctly. This is more apparent for group communication where group members can communicate back and forth with only a small subset of the entire group, and they must be able to synchronize all new messages when members meet.

The RST algorithm, described in section 2.5.1, proposes a solution that allows arbitrary messages to be sent in any order. This is achieved by sending a full matrix clock together with each message. This adds a lot of extra metadata, especially for large groups, since an $n \times n$ matrix will have to be sent along with each message, where n is the number of known members of the group when the message was created, from the perspective of the message sender. Improvements to RST that reduce the amount of the metadata have been proposed [23, 32], however it is still a major factor for this family of algorithms.

Some aspects of the extra metadata needed in RST could also be beneficial as it can provide detailed information about the message which could be displayed to the user. It can for example be used to infer that some messages exist but they have not been received yet. Having the flexibility of being able to send messages in any order might also be beneficial, like in situations with a lot of messages to synchronize, the latest messages could be prioritized and sent first as they might be most relevant to the user.

The Scuttlebutt algorithm, described in section 2.5.2, takes a different approach where it restricts the order that messages must be delivered, allows for much less metadata to be sent along with each message. This compromise is very desirable, since bandwidth is a major concern when designing the protocol.

With Scuttlebutt, messages must be delivered by the oldest version numbers first. This is not an issue since the full message history is stored locally on the device and keeps having relevance long after the message was first received.

Although the proposal does not maintain a matrix clock, a local one can be incrementally built and maintained from each digest. This is done, by updating the row in the matrix clock associated with the sender of the digest. The matrix clock will not be as up-to-date as with RST, since transitive updates to the clocks will not be sent along. Maintaining a matrix clock like so, still gives some relevant information to the user such as which members of a group are known to have received some message.

4.6.1 Group Authentication

When communicating in a group, participants use the same shared secret to encrypt the content of their messages using AES-GCM. This alone provides confidentiality, integrity, but also authentication in the sense that only someone who knows the shared secret (someone in the group) could have written the message. However, this still allows members to impersonate each other within the group. To prevent this, each member of the group has their own public and private key, which is used to sign the messages they send, such that other members of the group can verify the sender.

This produces a need for a mechanism for distributing the public keys within the group. However, this can be done as part of the message synchronization layer by including the public key of the node associated with each delta $\Delta^{q \to p}$. Even though this approach works, it can be inefficient, since the same public key is sent along with each new message from the same node n. However, q can determine if p already knows about n from the digest. By only sending the public key of n when $d_p(n) = \perp$, keys are only shared once for each node.

We chose to use the Ed25519 standard [9] for computing the digital signatures, since the size of the public keys and the generated signatures are small while still maintaining a security level of at least 128 bits. Having small public keys is convenient, but small signatures is even more crucial since these are sent along with each message.

4.6.2 Protocol Design

The synchronization layer is inspired by the Scuttlebutt gossiping algorithm, described in section 2.5.2. It uses vector and matrix clocks as well as anti-entropy gossiping to spread messages throughout a group. The main idea is that when Alice sends a route request to Bob, Bob will piggyback a vector clock to the route reply packet. Alice can then use the vector clock to calculate a set of deltas containing the new messages that Bob has not received yet. Alice will then send back her own vector clock along with all the deltas in order for Bob to catch up with Alice. When Bob receives the vector clock from Alice, he can in the same manner calculate the deltas for Alice and send them to her. When all the deltas have been delivered Alice and Bob are fully caught up with each other. If the session is broken before all the deltas have been delivered, the messages will simply be delivered later when a new session is established.

The synchronization extension introduces two new packet types.

SYNC-PULL A pull packet is sent to request deltas from the other party. It contains the digests of the sender used to calculate the deltas. Additionally the sender's public key is included along with a signature of the packet contents.

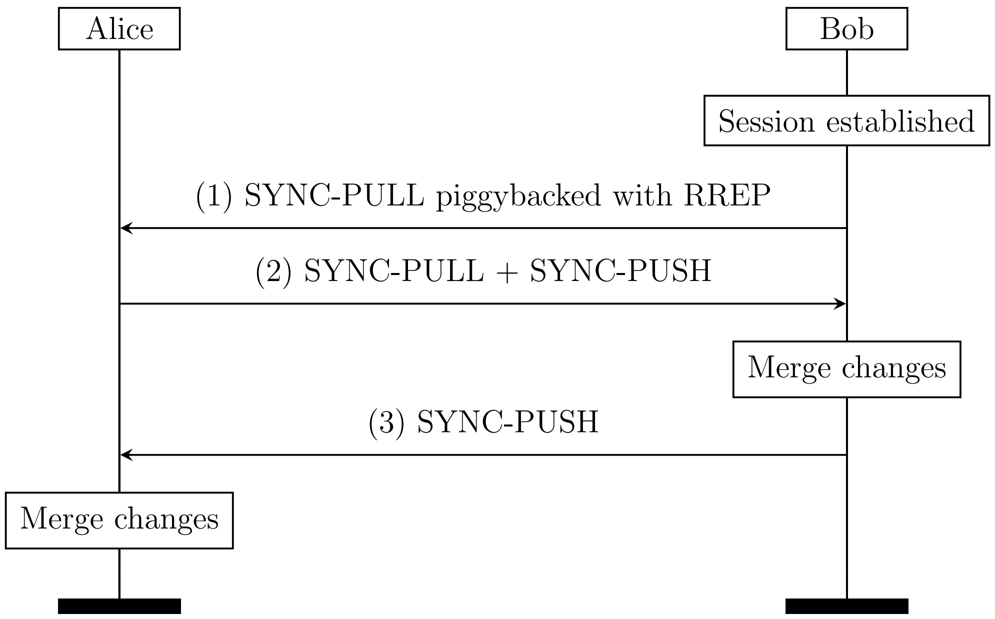


Figure 4.6: Typical synchronization flow after a session has been established.

SYNC-PUSH A push packet is sent as a response to a pull packet. It contains deltas with new messages. Additionally the sender's public key is included along with a signature of the packet contents.

Figure 4.6 shows the typical flow for synchronization between two parties Alice and Bob. When Bob has received a route request from Alice and a session has been established, he will send back a route reply with a new pull packet piggybacked along with it (1). When Alice receives the route reply, she simultaneously sends two packets back to Bob (2). First a push packet containing the deltas between her local state and the digest contained in the piggybacked pull packet. She also sends her own pull packet in order to also receive a delta from Bob. Lastly, when Bob receives the two packets, he can merge the deltas from the push packet and send back a push packet (3) as a reply to the newly received pull packet.

Incremental updates

After the initial synchronization, both parties are fully caught up and no more messages are needed to be exchanged. However, if a new message is created or received, their state is now out of sync, and they will have to resynchronize. In the case where they are still connected by the same session as used for the last synchronization, instead of performing a new full synchronization an incremental update is instead performed.

If the local state of one of the parties changes, they simply create a new push packet containing the deltas of the new changes and sends it to the other party. That way, everything after the initial synchronization can be sent directly to the relevant recipient(s).

Chapter 5

Protocol Implementation

We have implemented the Starling protocol in the Go programming language [20] as a generic library that can be integrated with any environment. This allows us to run the protocol on actual devices, by integrating the protocol with an environment that interacts over Bluetooth with other devices. However, it also enables us to run the exact same code in a simulated environment using the simulator, which is described more in-depth in chapter 6. The source code of the protocol implementation can be found online.[1]

The consumer of the library constructs a new instance of the protocol by calling the *NewProtocol* function shown in listing 5.1. The protocol instance is then used to interact with the environment, eg. by calling *OnConnection* when a new device has connected or *ReceivePacket* when a packet has been received.

A *device* interface (see listing 5.2) must be provided when constructing a new protocol instance. It defines how all the events the library can produce are handled. Essentially this serves as the way for the protocol to interact with the environment. For instance if the *BroadcastRouteRequest* method is called, the protocol will construct a route request and send it to its peers. This will in the end result in a call to the *SendPacket* method of the device.

The library is structured such that each layer is implemented as a separate Go package. Each layer uses the layer just below it, such that the application layer imports and uses the transport layer and so on. This allows us to test a specific layer without the added complexities of the higher levels added on top.

An example of this could be when the device receives a packet and hands it to the

[1] https://github.com/starling-protocol/starling

Listing 5.1: A summary of the *protocol* methods.

```go
type Protocol struct {
  dev         device.Device
  options     device.ProtocolOptions
  application *application_layer.ApplicationLayer
}

func NewProtocol(dev device.Device, options *device.ProtocolOptions) *Protocol {...}
func (proto *Protocol) OnConnection(address device.DeviceAddress) {...}
func (proto *Protocol) OnDisconnection(address device.DeviceAddress) {...}
func (proto *Protocol) SendMessage(session device.SessionID,
                                   message []byte) (device.MessageID, error) {...}
func (proto *Protocol) BroadcastRouteRequest() {...}
func (proto *Protocol) ReceivePacket(address device.DeviceAddress, packet []byte) {...}
```

Listing 5.2: An excerpt of the *device* interface.

```go
type Device interface {
  SendPacket(address DeviceAddress, packet []byte)
  MessageDelivered(messageID MessageID)
  MaxPacketSize(address DeviceAddress) (int, error)
  ProcessMessage(session SessionID, message []byte)
  ReplyPayload(session SessionID, contact ContactID) []byte
  SessionEstablished(session SessionID, contact ContactID, address DeviceAddress)
  SessionBroken(session SessionID)

  // some methods have been omitted...
}
```

Listing 5.3: A summary of the packet layer.

```go
type connection struct { encoder *PacketEncoder, decoder *PacketDecoder }

type PacketLayer struct {
  dev         device.Device
  options     device.ProtocolOptions
  connections map[device.DeviceAddress]*connection
}

func NewLinkLayer(dev device.Device, options device.ProtocolOptions) *PacketLayer {...}
func (link *PacketLayer) OnConnection(address device.DeviceAddress) {...}
func (link *PacketLayer) OnDisconnection(address device.DeviceAddress) {...}
func (link *PacketLayer) ReceivePacket(sender device.DeviceAddress, packet []byte) [][]byte
↪  {...}

func (link *PacketLayer) SendBytes(address device.DeviceAddress, data []byte) bool {...}
func (link *PacketLayer) BroadcastBytes(data []byte) {...}
```

protocol by calling the *ReceivePacket* method on the protocol instance. That call will first be forwarded to the application layer. The application layer will then call the transport layer in order to handle the packet from the perspective of the transport layer. The transport layer will similarly call the network layer which lastly will use the packet layer to decode the packet. Once the packet has been decoded into bytes, the network layer can decode these bytes into a network packet. If the result is a network session packet, the resulting data will first be decrypted and then returned to the transport layer. Similarly, the transport layer will decode the decrypted session data to a transport packet. If this results in a transport data packet, that will be returned back up to the application layer, which decodes the data and handles it appropriately.

5.1 Packet Layer

The packet layer is responsible for encoding and sending packets, an overview of the implementation of the packet layer is seen in listing 5.3. It keeps track of each active connection through the *OnConnection* and *OnDisconnection* methods. A *connection* contains an encoder for outgoing packets, and a decoder for ingoing packets.

When the *SendBytes* method is called, the packet is first encoded using the encoder, and then handed to the *SendPacket* method of the device from listing 5.2. The *BroadcastBytes* internally calls *SendBytes* once for each connection in the *connections* map. This simulates multicast by sending the message to all currently active connections.

Listing 5.4: A summary of the *PacketEncoder* and *PacketDecoder* classes.

```go
type PacketEncoder struct {...}
func NewPacketEncoder(packetSize int) *PacketEncoder {...}
func (encoder *PacketEncoder) EncodeMessage(message []byte) error {...}
func (encoder *PacketEncoder) PopPacket() []byte {...}

type PacketDecoder struct {...}
func NewPacketDecoder() *PacketDecoder
func (decoder *PacketDecoder) AppendPacket(packet []byte) error {...}
func (decoder *PacketDecoder) ReadMessage() ([]byte, error) {...}
```

5.1.1 Encoding and Decoding

The objective of the packet layer is to split up packets that are too large to be sent over the link layer as a single packet, and combine them again at the receiving end. The module contains an encoder and a decoder class as seen in listing 5.4. Raw bytes can be written to instances of the two classes, and the encoded or decoded messages can then be extracted afterwards.

Encoder. A new packet encoder can be constructed by providing a max packet size. The constructed encoder can then be used to encode packets into packets of max size *packetSize*. The raw bytes of a packet can be written to the encoder using the *EncodeMessage* method. In order to get the resulting encoded packet the *PopPacket* method is called. This will return the first packet contained in the packet encoder. Multiple smaller packets can be encoded into one encoded packet, if they do not exceed the packet size of the encoder.

Decoder. A packet decoder does the reverse operation of the packet encoder, it reads encoded packets using the *AppendPacket* method. Afterwards the *ReadMessage* method can be called in order to extract the original packet(s).

5.2 Network Layer

As described in section 4.3, it is the responsibility of the network layer to establish secure sessions between two contacts on the network. The network layer provides a few public functions that are available to the transport layer, such as the *SendData* and the *ReceivePacket* methods. It also has events that it can emit to inform the transport layer when certain things happen. Both the public functions and events can be seen in listing 5.5. For example, the *SessionEstablished* and *SessionBroken* event, are called to inform the above layers that a session has been established or broken with the given ID to the given contact. It also has a constructor *NewNetworkLayer*, that an upper layer can use to get an instance of the network layer. This constructor needs the device, the options of the protocol as well as an implementation of the *NetworkLayerEvents* interface.

5.2.1 Handling Packets

Once the network layer receives a packet from the transport layer via the *ReceivePacket* function, it is its responsibility to handle it according to the protocol specification. It inspects the packet and uses the packet type field to deduce whether it is a route request, Listing 5.5: Some of the public functions and events of the network layer.

```go
func (network *NetworkLayer) OnConnection(address device.DeviceAddress) {...}
func (network *NetworkLayer) OnDisconnection(address device.DeviceAddress) {...}
func (network *NetworkLayer) ReceivePacket(sender device.DeviceAddress, packet []byte)
↪  []SessionMessage {...}
func (network *NetworkLayer) DeleteContact(contact device.ContactID) {...}
func (network *NetworkLayer) SendData(session device.SessionID, data []byte) error {...}

type NetworkLayerEvents interface {
  SessionEstablished(session device.SessionID, contact device.ContactID, address
  ↪  device.DeviceAddress, payload []byte, isInitiator bool)
  SessionBroken(session device.SessionID)
  ReplyPayload(session device.SessionID, contact device.ContactID) []byte
}

func NewNetworkLayer(dev device.Device, events NetworkLayerEvents, options
↪  device.ProtocolOptions) *NetworkLayer {
  layer := &NetworkLayer{
    dev:          dev,
    events:       events,
    options:      options,
    packetLayer:  packet_layer.NewLinkLayer(dev, options),
    requestTable: make(RequestTable),
    sessionTable: make(SessionTable),
  }
  return layer
}
```

route reply, route error or session packet. In the case where it is a route request with an unseen request ID, a new *RequestTableEntry* is created. The structure of the *RequestTable* can be seen in listing 5.6. Here, *SourceNeighbour* is the address of the node that it received the route request from, and the *EphemeralKey* is a newly generated Ed25519 public key.

If the contact bitmap indicates that this node is an intended receiver, multiple things happen. We can calculate the session secret (as described in section 4.3.4), generate a random session ID, and create an entry in the *SessionTable*, as seen in listing 5.6. The *SessionTable* stores among other things, the session ID, contact ID and the session secret. The fields *SourceNode* and *TargetNode* are used to identify which neighbors the node should receive and forward to. For endpoint nodes, only one of these is set while the other is nil. For the node that initiated the route request, only the *TargetNode* would be

Listing 5.6: The request and session tables.

```go
type RequestTable map[RequestID]*RequestTableEntry
type RequestTableEntry struct {
  RequestID           RequestID
  SourceNeighbour     *device.DeviceAddress
  EphemeralPrivateKey *ecdh.PrivateKey
}

type SessionTable map[device.SessionID]*SessionTableEntry
type SessionTableEntry struct {
  RequestID       RequestID
  SessionID       device.SessionID
  Contact         *device.ContactID
  SourceNeighbour *device.DeviceAddress
  TargetNeighbour *device.DeviceAddress
  SessionSecret   []byte
}
```

set. In this case, since the node is the receiver of a route request, only the *SourceNode* field is set. Once the node has created the entry in its session table, it can construct a route reply packet which includes potential piggybacked data from upper layers. The piggybacked data is obtained by emitting a *ReplyPayload* event, to which the upper layer can reply with data it wants to be attached. When the route reply is constructed, it can be sent back along the path to the initiator. The intermediate nodes along the path will construct new entries in their session tables as they receive the route reply. Intermediate nodes can of course not fill the contact ID and session secret entries, but only need to store the session ID as well as the source and target nodes, so they can forward packets. Even though they have the names source and target, intermediate nodes can of course forward packets in both directions.

Once the other endpoint node receives the route reply, it will also create an entry in its *SessionTable*, as well as informing the transport layer with a *SessionEstablished* event. Since the session has been established, upper layers can call the *SendData* function on the network layer to send data over the given session. This will create a session packet, which will be forwarded along the path in the network, in the same manner as the route reply.

If any node along the path receives a call to its *OnDisconnection* function with an address that is either their *SourceNode* or *TargetNode* for a given session, they will remove their entry in the *SessionTable* and send a route error packet along the other half of the route. Once a node receives a route error packet, it will also delete the corresponding entry in its session table and forward it on. When an endpoint node receives the route error packet, it will emit a *SessionBroken* event to the transport layer.

5.2.2 Contact Bitmap

The implementation of the contact bitmap is its own separate package, but is used heavily by the network layer. It has functions for encoding and decoding contact bitmaps, equivalent to algorithm 4.2 and algorithm 4.4. It also has an implementation of the ContactBits function as described in algorithm 4.3. This implementation, as following the description in listing 5.7, is worth discussing, since it shows how the indices in the contact bitmap are actually calculated in practice. As described in section 4.3.5, a family of hash functions for mapping between contacts and bit indices is needed. This family could be constructed using techniques such as concatenation or masking, however these all require that the hash functions are run multiple times, one time for each bit. Instead, the choice was made to use a single HMAC function [30], and use different sections of the hash as different indices. This was mainly chosen due to efficiency. Essentially a SHA256 HMAC function is created with the contact secret of the contact to be encoded. This hash function is used on the seed to obtain a single hash with a length of 64 bytes. The bitmap has a length of 2048 bits, and thus we need $\log_2 2048 = 11$ bits to index into it. Since we need to find 12 indices in the bitmap, we can simply split the hash up in pieces of 16 bits and use the lowest 11 bits of each piece as the index. This is done in line 11 of the listing.

5.3 Transport Layer

The responsibility of the transport layer, as described in section 4.4, is to facilitate reliability of a network layer session. This entails resending packets that are dropped and making

Listing 5.7: A function for calculating the relevant bit-indices of for a contact.

```go
func ContactBits(seed Seed, secret device.SharedSecret) ([]int, error) {
  if len(secret) != 32 { ... }
  bits := []int{}

  hasher := hmac.New(sha256.New, secret)
  if _, err := hasher.Write(seed.Bytes()); err != nil { return nil, err }

  hash := hasher.Sum(nil)

  for i := 0; i < 12; i++ {
    bitIndex := int(binary.LittleEndian.Uint16(hash[i*2:]) & 0x7FF)
    for slices.Contains(bits, bitIndex) {
      bitIndex = (bitIndex + 1) % 0x7FF
    }
    bits = append(bits, bitIndex)
  }

  return bits, nil
}
```

sure packets are delivered in the order they were sent. An overview of the structure of the transport layer can be seen in listing 5.8.

An instance of the *TransportLayer* structure can be constructed using the *NewTransportLayer* function. It requires an implementation of the *TransportEvents* interface among other things. This interface provides callbacks for events emitted by the transport layer. The *SessionEstablished*, *SessionBroken* and *ReplyPayload* are simply forwarding the events emitted from the underlying network layer. The *MessageDelivered* event is specific for the transport layer, and is emitted when a packet that was earlier sent has now been acknowledged.

The transport layer exposes the *SendMessage* function, that when called will send a message on a session given by its session ID. When called, a new data packet will be sent along the chain of intermediate nodes towards the recipient. The packet will be stored temporarily until an acknowledgement packet is later received indicating that the data packet has been delivered. If the sequence number of the packet is among the missing sequence numbers of the acknowledgement packet, the packet will be resent.

Like with the other layers, the transport layer also exposes a *ReceivePacket* function, which passes the packet bytes down to the network layer. If the network layer decodes it into a session packet that contains data, the decoded data will be returned to the transport layer. Here, it will be decoded as either a data packet or an acknowledgement packet, and handled appropriately.

5.3.1 Session State

When an event is emitted from the network layer indicating that a new session has been established, the transport layer will save a *SessionState* (shown in listing 5.9) in a map, using the session ID for the key. Since the session allows for bidirectional communication, the state is organized into a *SenderState* and a *ReceiverState*, holding the state related to the sending and the receiving part of the session respectively.

The sender state keeps track of everything related to sending on the session. It maintains the sequence ID to use for the next data packet to be sent. Every time a new data packet is sent, this value is incremented by one. It also keeps track of all data packets

Listing 5.8: Some public functions and events of the transport layer.

```go
type TransportEvents interface {
    SessionEstablished(session device.SessionID, contact device.ContactID,
                       address device.DeviceAddress, payload []byte, isInitiator bool)
    SessionBroken(session device.SessionID)
    ReplyPayload(session device.SessionID, contact device.ContactID) []byte
    MessageDelivered(messageID device.MessageID)
}

func (tr *TransportLayer) ReceivePacket(address device.DeviceAddress,
                                        packet []byte) []TransportMessage {...}
func (tr *TransportLayer) SendMessage(sessionID device.SessionID,
                                      message []byte) (device.MessageID, error) {...}

func NewTransportLayer(dev device.Device, events TransportEvents, options
↪   device.ProtocolOptions) *TransportLayer {
    transport := TransportLayer{
        dev:           dev,
        events:        events,
        options:       options,
        networkLayer:  nil,
        outbox:        []outboxMessage{},
        sessionStates: map[device.SessionID]*SessionState{},
    }

    transport.networkLayer = network_layer.NewNetworkLayer(
        dev, &networkEvents{transport: &transport}, options)

    return &transport
}
```

that are awaiting an acknowledgement, i.e. they have not been confirmed to have been delivered yet. This is needed such that if an acknowledgement packet is received which includes a missing sequence ID, the data packet with that ID can be found and resent.

Lastly, it has a boolean indicating whether the timeout timer is activated. When it is, a timer is running in the background, counting down for the session to time out. This timer is started when a new data packet is sent and the timer is not already running. When the timer runs out, it will check if the data packet has been delivered and acknowledged in the meantime, and break the session if it has not. If it has, the timer will potentially be restarted if a new data packet has been sent and not yet delivered while the timer was running.

Similarly, the receiver state keeps track of everything related to receiving on the session. It maintains the latest sequence ID that has been received. When a data packet is received, it can be delivered immediately if it has a sequence ID exactly one larger than the latest sequence ID and there are no other missing packets with a sequence ID lower than it. When the data packet is delivered, the latest sequence ID is updated to match it. If a data packet is received with a sequence ID larger than the latest sequence ID, that means that the packet has been received out of order and the transport layer must wait for the earlier packets to be delivered before this one can be delivered. It does so by adding the data packet to the *awaitingDelivery* list. Every time a new data packet has been delivered, the awaiting delivery list is checked to see if it contains the data packet with the next sequence ID and deliver it if it was found. Multiple packets awaiting delivery can be delivered at once, if their sequence IDs are right after each other.

If a data packet is received out of order, the missing sequence numbers, up to this one, are added to the *missingSeqs* list. When they are later received they will be removed

Listing 5.9: The structure of the session state in the transport layer.

```go
type ReceiverState struct {
  ackTimer         bool
  latestSeq        SequenceID
  missingSeqs      []SequenceID
  awaitingDelivery []*DATAPacket
}

type awaitingACK struct {
  message    outboxMessage
  sequenceID SequenceID
  timestamp  time.Time
}

type SenderState struct {
  timeoutTimer   bool
  awaitingACKs   []awaitingACK
  nextSequenceID SequenceID
}

type SessionState struct {
  sendLock    sync.Mutex
  receiveLock sync.Mutex
  sessionID   device.SessionID
  sender      *SenderState
  receiver    *ReceiverState
}
```

again. This list will be attached in acknowledgement packets in order to request the missing packets to be resent.

The *ackTimer* attribute works much in the same way as the *timeoutTimer* does. When a data packet is received the ack timer is started if it is not already running. When the timer finishes, an acknowledgement packet will be sent. It will contain the latest sequence number and the missing sequence numbers stored in the receiver state as of the time the timer finishes. If another data packet has been received while the timer was still running, the timer will be restarted. Otherwise, the timer will first start again when a new data packet is received.

5.4 Application Layer

The application layer serves to allow different applications to send their application specific messages on the network. It is quite thin in the sense that while it has many public methods and events, it mostly just delegates these to the lower layers for them to handle. It has the constructor *NewApplicationLayer* which is used to obtain a new instance of the application layer. This takes a *Device* interface and a *ProtocolOptions* structure, and constructs the new application layer, which involves also constructing a corresponding transport layer for it to use. Here, the *ProtocolOptions* structure represents the options for the protocol, and thus defines what behavior the protocol expects for this layer. One example of an option is the `EnableSync` option, which is a boolean indicating whether the synchronization extension is enabled or not.

Application layer packets are quite simple, since they simply consist of the application data, with a single byte prepended to it. This *packet type* byte determines the type of application data. Currently, only two types of application data are defined, but this could be expanded in the future. The packet types are defined in section 4.5. Packets are

Listing 5.10: Overview of some of the core structs and types of the synchronization extension.

```go
type Version uint32
type NodePublicKey [32]byte

type Model struct {
  Digests    Digests            `json:"digests"`
  PrivateKey ed25519.PrivateKey `json:"private_key"`
  PublicKey  NodePublicKey      `json:"public_key"`
  NodeStates ModelNodeStates    `json:"node_states"`
  Type       ModelType          `json:"type"`
}

type Digests map[NodePublicKey]*Digest
type Digest struct {
  Nodes      DigestNodes `json:"nodes"`
  MaxVersion Version     `json:"max_version"`
}
type DigestNodes map[NodePublicKey]Version

type Delta struct {
  PublicKey      NodePublicKey
  Version        Version
  Value          []byte
  AttachedSecret []byte
  Signature      Signature
}

type ModelNodeStates map[NodePublicKey]NodeState
type NodeState map[Version]Message
type Message struct {
  Value          []byte              `json:"value"`
  Signature      Signature           `json:"sig"`
  AttachedSecret device.SharedSecret `json:"attached_secret"`
}
```

simply decoded by looking at the packet type byte and then handling the rest of the data according to that type.

For *application specific packets*, the data is simply sent off to the device for it to handle. This happens by calling the *ProcessMessage* function of the *Device* associated with the application layer. For application specific packets, the implementor of the *Device* interface is responsible for decoding and processing the data if needed.

If the packet instead is a *synchronization packet*, the data is handled by the synchronization extension. How this extension is implemented will be described in section 5.5. For the application layer to handle synchronization packets, the `EnableSync` options has to be given as part of the *ProtocolOptions*.

5.5 Synchronization

The synchronization extension has been implemented as its own separate package. It provides a constructor and a some public functions such as *ReceiveSyncPacket* and *NewMessage* for interacting with it. Practically, it receives calls to these from the application layer. It also has events that it can emit, to inform the application layer when certain things occur. The synchronization implementation is responsible for keeping track of the messages that have been received from different contacts. It does this by maintaining a map from contacts IDs to *models*, where a model is a structure containing the relevant data for a chat. It also keeps track of which active sessions exist, and to which contacts these sessions belong.

5.5.1 Model

The model structure seen on listing 5.10, includes a *PrivateKey* and a *PublicKey* which is ones own identity in the model. New messages will be signed with this private key in order for others to verify ones identity. The structure also includes a list of digests and a model type, which indicates whether the model holds a group of members or a conversation of two people.

The *NodeStates* map holds all messages sent and received in the model. It is a multi level map of the form $(\mathit{PublicKey} \times Version) \mapsto \mathit{Message}$, where *PublicKey* is the public key of a node in the model, and $Version$ is the associated version number of the message. This maps to a *Message* structure which contains *Value*, the actual contents of the message, and a *Signature* generated by the sender in order to authenticate the message. Lastly, a message can optionally contain an *AttachedSecret* with a contact secret of a group that allows recipients of the message to join that group.

5.5.2 Digest

A *Digest* structure (listing 5.10) contains a map $\mathit{PublicKey} \mapsto Version$, that maps public keys to the highest version number ever received locally from the node with the given public key.

A *Digests* map $\mathit{PublicKey} \mapsto \mathit{Digest}$ is stored in the model, which maps a public key to a local perspective of the digest of the given public key. This essentially models a matrix clock. Whenever a digest is received from another node, that digest is stored under the public key of the node, in the *Digests* map. Although the *Digest* map is strictly speaking enough to follow the protocol specification, the *Digests* map provides extra information which can for example be used to deduce which messages have been received by which nodes.

5.5.3 Delta

A *Delta structure* (listing 5.10) contains the same attributes as the *Message* structure, but it also includes the public key of the sender and version number of the message. A delta is used to encode messages in a push packet. A list of deltas are included in each push packet which describes the messages that are synchronized.

5.5.4 Handling Packets

The synchronization extension introduces two packets, a pull and a push packet.

Pull Packet

The pull packet contains the digest of a node. When a pull packet is received for the first time over a session, synchronization for the given session is registered with the public key of the sender. The registration means that future updates on the model will trigger a new push packet to this node.

After the session has been registered, the appropriate digests are updated. The digest associated with the public key of the sender of the pull packet in the *Digests* attribute of the model, will be updated to become the new digest from the pull packet.

Lastly, an event informing the upper layers that the synchronization state has changed is fired.

Push Packet

When a push packet is received from another node, the deltas included in the packet will be merged in the *NodeStates* map of the *Model*. This is done by using the public key and version number in the delta as keys into the *NodeStates*. Of course, the signatures of each of the messages are verified to check that the messages are authentic. A new *Message* is created and saved in the map, with the value, signature and attached secret from the delta.

Lastly, the upper layers are notified about the state changes and a push packet with changes will be sent to each registered session.

5.6 Tests

In order to make the implementation correct, we have focused heavily on automated testing of the code in the form of unit tests.

We have three major kinds of automated tests. Traditional unit tests, which test individual functions and enclosed functionality. Large manual tests of the network and transport layers, calling the methods of the layer in the expected order for some common scenario. And lastly tests utilizing the simulator to test that the protocol events work together as expected and more advanced coordination work. For example simulating multi-hop session establishments or group synchronization in a large network. The simulator tests will be described later in section 6.7.2.

5.6.1 Unit Tests and Fuzzing

Packet encoding and decoding is a very isolated task which makes it easy to test. A lot of effort has been put into testing encoding and decoding, since an error here could potentially allow an attacker to send maliciously crafted packets that could conceivably cause a panic and crash the application.

In order to test against maliciously crafted packets that break the implementation, we utilize the builtin fuzzing functionality of the Go testing framework.[2] A fuzzing test in Go works similar to a traditional unit test, but instead of providing the parameters manually, a set of starting parameters, called the corpus, is given. The fuzzer will then automatically generate new parameters by making changes to the corpus. The fuzzer then analyzes the code paths taken for the given parameters, and attempts to mutate the parameters in order to reach as many paths through the code as possible.

Listing 5.11 shows an example of a fuzz test that first encodes a route reply and then decodes it again in order to test that encoding and decoding the packet, does not change the packet. First it adds two examples to the corpus, providing the parameters that make up the route reply: the request ID, session ID and the payload of the route reply as well as

[2]See the official documentation `https://go.dev/doc/security/fuzz/`.

Listing 5.11: Fuzzing test for route reply encoding and decoding.

```go
func FuzzCodingRREP(f *testing.F) {
  f.Add(int64(1234), 1, 2, []byte{})
  f.Add(int64(1234), 1, 2, []byte("payload"))

  f.Fuzz(func(t *testing.T, seed int64, _reqID int, _sessID int, payload []byte) {
    random := rand.New(rand.NewSource(seed))
    reqID := network_layer.RequestID(_reqID)
    sessID := device.SessionID(_sessID)
    ephemeralPrivate, err := ecdh.X25519().GenerateKey(random)
    assert.NoError(t, err)

    rrep := newRREP(t, random, reqID, sessID, ephemeralPrivate, payload)

    assert.Equal(t, reqID, rrep.RequestID)
    assert.Equal(t, sessID, rrep.SessionID)
    assert.Equal(t, *ephemeralPrivate.PublicKey(), rrep.EphemeralKey)
    assert.Len(t, rrep.Cipher, 16+len(payload))
    assert.Len(t, rrep.Nonce, 12)

    encoded := rrep.EncodePacket()
    assert.Len(t, encoded, 65+len(payload)+16)

    decoded, err := network_layer.DecodeRREP(encoded)
    assert.NoError(t, err, "failed to decode RREP packet")

    assert.Equal(t, rrep, decoded)
  })
}
```

a seed used for seeding the ephemeral key generation. These parameters are only used to initialize the fuzzer and new values for the parameters will be generated by mutations of these. The test constructs a new route reply, asserts that all its attributes are as expected. Then it encodes it, asserts that the encoded bytes are of the expected length. Lastly, it decodes it again and asserts that the decoded packet is equal to the original one.

The example shown in listing 5.11 only attempts to decode validly constructed route reply packets. It is still important to test that the decoder fails as expected when given invalid bytes as input. The fuzz test shown in listing 5.12 tests this. The parameters for the fuzzer is only a sequence of bytes. For the corpus a valid encoding of a route reply is added. Additionally, a random byte sequence is added to act as invalid input. The test can expect very little when decoding the route reply, since arbitrary bytes are given as input, both valid and invalid inputs can be given. The only thing the test can expect is therefore that the function will not cause a panic.

5.6.2 Transport Layer Testing

Since the transport layer is quite complicated because of the timeout and acknowledge timers, a framework has been made for testing the layer with very fine control. In order to construct an instance of the transport layer, a test mock has been made for the device interface (see listing 5.2), and a mock of the transport events interface. The two mocks generally work in the same way. All events emitted by the interface is stored in a stack, such that they can be popped in a controlled order and asserts can be inserted in-between.

All the unit tests for the transport layer require a connection between two nodes. The *setupConnection* function seen in listing 5.13 shows how this is done. The first half of the function, constructs two transport layers, *transportLayerA* and *transportLayerB*, along

Listing 5.12: Fuzzing test for decoding invalid route replies.

```go
func FuzzDecodingRREP(f *testing.F) {
  random := rand.New(rand.NewSource(1234))
  ephemeralPrivate, err := ecdh.X25519().GenerateKey(random)
  assert.NoError(f, err)
  rrep := newRREP(f, random, network_layer.RequestID(456), device.SessionID(654),
↪  ephemeralPrivate, []byte{})
  f.Add(rrep.EncodePacket())

  invalid_rrep := [50]byte{}
  if n, err := random.Read(invalid_rrep[:]); n != 50 || err != nil {
    f.Fatal()
  }
  f.Add(invalid_rrep[:])

  f.Fuzz(func(t *testing.T, bytes []byte) {
    assert.NotPanics(t, func() {
      network_layer.DecodeRREP(bytes)
    })
  })
}
```

with the associated mocks for the devices and the transport layer events. Both transport layers create a contact with the same contact secret, using the *DebugLink* function.

Each transport layer starts by calling the *OnConnection* method (line 26-27), providing the address of the other node. This informs the layers that the nodes now have a connection and can communicate with each other on their respective addresses. Next, *transportLayerA* calls *BroadcastRouteRequest* (line 28) to send a route request to B. The *ReceivePacket* is then called on *transportLayerB* (line 31) to simulate that B receives the route request packet from A. This call will result in a route reply being sent on the mocked device of B. This packet is now handed over to A by a call to its *ReceivePacket* method (line 34).

Now, a route request has been sent to B and a route reply has been sent back to A. The route reply piggybacks data from the transport layer and is thus the first packet to be acknowledged. When A received the route reply, the acknowledgement timer was started. Since this is a testing environment, instead of starting an actual timer which is very hard to control and test, the mocked device instead pushes the action that should be run when the timer ends to a stack. This allows us to simulate the timer ending at at controlled point in the test. The acknowledgement timer is then ended by a call to *ExecuteNextDelayAction* on the mocked device of A (line 37). This will result in an acknowledgement packet being sent along to B. The acknowledgement packet is handed to B by a call to *ReceivePacket* (line 40). Lastly, the timeout timer of B is ended by a call to *ExecuteNextDelayAction* (line 43). Since the acknowledgement has been received and no new data is received at this point, the timeout timer does not break the session, and it will not be restarted.

Both A and B now have an active and acknowledged session with each other. The end of the function simply collects all mocks and related information of the two nodes into the structures *nodeA* and *nodeB*, which the function returns to be used for further testing.

Packet Drop Test

The *setupConnection* function in listing 5.13 is used as the starting point for multiple unit tests, one of which is the *TestSingleDataPacketDrop* test seen in listing 5.14.

After the setup of *nodeA* and *nodeB*, the test starts off by *A* sending a message (line 7). The packet is now popped from the device but never delivered to *B*, this simulates the packet being dropped (line 11). Instead, *A* sends a second message, which is handed to *B* by a call to *ReceivePacket* (line 14-15). The method returns the list *receivedMessage* of messages to be delivered. We assert this list to be empty, since the second packet can only be delivered once the first one has been delivered.

Next, we simulate the acknowledgement timer of *B* ending (line 20), which will result in an acknowledgement packet being sent to *A*. This packet will include the sequence number 1 in its missing sequence numbers list. When the acknowledgement packet is received by *A* (line 21), this will trigger the first packet to be resent. When this packet is received by *B* (line 24), both messages will be delivered, which is checked by asserting that *receivedMessages* contains two messages (line 25). Lastly the order of the messages is asserted to be correct by asserting that the data field of the messages match the original strings (line 26-27).

Listing 5.13: Setup a session between two nodes in a testing environment.

```go
type TestNode struct {
  address        device.DeviceAddress
  transportLayer *transport_layer.TransportLayer
  dev            *testutils.DeviceMock
  contact        device.ContactID
  session        device.SessionID
}

func setupConnection(t *testing.T, addressA device.DeviceAddress, addressB device.DeviceAddress)
↪  (*TestNode, *TestNode) {
  seed := rand.NewSource(rand.Int63())
  random := rand.New(seed)
  sharedSecret := bytes.Repeat([]byte{0x01}, 32)
  protoOptions := device.DefaultProtocolOptions()
  protoOptions.DisableAutoRREQOnConnection = true

  devA := testutils.NewDeviceMock(t, random)
  transportLayerA := transport_layer.NewTransportLayer(devA, transportEvents{devA},
  ↪  *protoOptions)
  contactA := devA.Contacts.DebugLink(sharedSecret)

  devB := testutils.NewDeviceMock(t, random)
  transportLayerB := transport_layer.NewTransportLayer(devB, transportEvents{devB},
  ↪  *protoOptions)
  contactB := devB.Contacts.DebugLink(sharedSecret)

  assert.Equal(t, contactA, contactB)

  transportLayerA.OnConnection(addressB)
  transportLayerB.OnConnection(addressA)
  transportLayerA.BroadcastRouteRequest()

  // RREQ
  transportLayerB.ReceivePacket(addressA, devA.PopLastPacket())

  // RREP
  transportLayerA.ReceivePacket(addressB, devB.PopLastPacket())

  // Delay for sending ACK of RREP
  devA.ExecuteNextDelayAction()

  // ACK for RREP
  transportLayerB.ReceivePacket(addressA, devA.PopLastPacket())

  // Delay for ACK timeout
  devB.ExecuteNextDelayAction()

  nodeA := NewTestNode(addressA, transportLayerA, devA, contactA, devA.Sessions[0])
  nodeB := NewTestNode(addressB, transportLayerB, devB, contactB, devA.Sessions[0])
  return nodeA, nodeB
}
```

Listing 5.14: A test where node *A* sends two packets, but the first is initially dropped. Node *B* will notice this and request the first packet be resend.

```go
func TestSingleDataPacketDrop(t *testing.T) {
  addressA := device.DeviceAddress("1000")
  addressB := device.DeviceAddress("2000")
  nodeA, nodeB := setupConnection(t, addressA, addressB)

  // A sends first packet.
  nodeA.transportLayer.SendMessage(nodeA.session, []byte("Hello from A"))
  assert.NotEmpty(t, nodeA.dev.PacketsSent)

  // Packet is dropped, and B does not receive the message.
  nodeA.dev.PopLastPacket()

  // A sends second packet, and this is received by B (however not delivered yet).
  nodeA.transportLayer.SendMessage(nodeA.session, []byte("This is the second message"))
  receivedMessages := nodeB.transportLayer.ReceivePacket(addressA, nodeA.dev.PopLastPacket())
  assert.Empty(t, receivedMessages)

  // B's ack timer times out, and it replies with an ACK
  // containing the sequence numbers of the missing packets.
  nodeB.dev.ExecuteNextDelayAction()
  nodeA.transportLayer.ReceivePacket(addressB, nodeB.dev.PopLastPacket())

  // A responds by resending the initial message.
  receivedMessages = nodeB.transportLayer.ReceivePacket(addressA, nodeA.dev.PopLastPacket())
  assert.Equal(t, len(receivedMessages), 2)
  assert.Equal(t, receivedMessages[0].Data, []byte("Hello from A"))
  assert.Equal(t, receivedMessages[1].Data, []byte("This is the second message"))
}
```

Chapter 6

Simulator

The main scenario that the Starling protocol targets is big protests with many people, as described in chapter 3. This presents a problem, since the scenario would be difficult to fabricate for real life tests. As a way to evaluate how the protocol would fare in a realistic scenario, a simulator has been created to simulate and evaluate the protocol. It is also useful as a tool when developing the protocol, since important decisions can be made on a more informed basis. The source code for the simulator can be found online.[1]

6.1 Design Decisions

The simulator has been implemented as its own standalone package in Go. It can simulate environments consisting of nodes sending and receiving packets while moving around in the plane. This allows for testing MANET protocols in custom scenarios. The choice of building a simulator from scratch instead of using an existing one such as the popular ns-3 [45], has some advantages and disadvantages. One of the main advantages is that since the simulator and the Starling protocol implementation is written in the same programming language, it is very easy to integrate the two. This allows us to set up quite involved unit tests of the protocol using the simulator, which is useful for catching mistakes during development. It also means that we can use the same implementation of the protocol both for running on the physical devices as well as in the simulator, which helps ensure the validity of the implementation. Another advantage is that we can set up complex movement models for the nodes on the network, to simulate real life scenarios as close as possible. One of the main problems with a custom simulator is making a accurate simulation of the physical layer and link layer. The simulator simulates Bluetooth Low Energy (BLE), however it does so with some major simplifications. This will be described in section 6.4.

The scenario to be modelled, is at its core a network of nodes transmitting and receiving packets to and from each other. This can quite fittingly be modelled using *discrete event simulation*, since the major occurrences can be seen as discrete events happening at specific times. When a node wants to send a packet, it creates an event for sending it at a given time, and the other node receives it at another time after some delay. Another possibility is continuous simulation, where changes in the system are described by a set of differential equations. This would mostly make sense for only simulating the behavior of nodes on a macro scale. However, this does not really make sense for simulating individual

[1] `https://github.com/starling-protocol/simulator`

nodes sending and receiving packets.

When designing a simulator, one of the fundamental decisions to make, is how to model the progression of time. There are generally two categories within discrete event simulation; *Next-event time progression* and *incremental time progression*. Next-event time progression means that the simulator can skip straight through to the next event after it has processed the current event, no matter how far ahead in time it is. Incremental time progression, on the other hand, means that time is divided into slices, and all events happening within a slice are seen as happening at the same time.

One aspect of our problem that does not follow the discrete event structure, is the movement of the nodes on the network. Since nodes move around, the nodes within another node's vicinity (its neighbors) will change over time. There needs to be events informing the nodes on the network, of nodes that are now within reach, *connections*, and nodes that are now no longer in reach, *disconnections*. These events, while discrete, happen due to the continuous movement of nodes on the plane. Due to this, we have chosen to use a hybrid between next-event time progression and incremental time progression. We do this by discretizing the movement of the nodes to only happen at certain events scheduled regularly. This has the downside of limiting the granularity of the movement of the nodes to the time increment value. However, since nodes generally move at a relatively slow speed, we see this as a fine trade off.

6.1.1 Event Queue

Let an event E be defined by $E = (t, s, e, d)$, where t is the time, s is the sequence number, e is the event type, and d is other associated data of the event. When a new event is created, it is pushed to the event queue of the simulator, which is a priority queue with (t, s) as the key. A priority queue was chosen to make it easy to dequeue the next event. The keys are ordered lexicographically, such that the event with the lowest time is chosen first, and if two events have the same time, the one with the smallest sequence number is selected. Since the simulator has to potentially handle many nodes, which each can create many events in a short time span, the event queue has to be quite efficient. For this reason, it has been implemented using a heap.

6.2 Structure

The simulator is centered around a main *Simulator* structure, which is responsible for keeping track of the state of the simulation as well as handling time progression and events. The *Simulator* can be created with its constructor, *NewSimulator*. Among other fields, it has a *time* and a *currentSequenceNumber* field. The time field keeps track of how long in milliseconds that the simulation has progressed, and the *currentSequenceNumber* field is used to give unique sequence numbers to events. The simulator also maintains the locations of nodes and whether nodes are within range and thus are connected to each other.

The simulator has public methods for adding nodes to the simulation, starting the simulation and running the simulation, among other things. The simulator has been built such that it can be used, not just for testing our protocol, but such that other MANET protocols could easily be tested. For someone to use the simulator, they need to provide an implementation of the nodes on the network. The implementation must follow the *Node* interface, as will be presented in detail in section 6.2.1. This interface serves as a

Listing 6.1: The *Node* interface and the *NodeArguments* structure.

```go
type Node interface {
  OnConnect(peer Peer, id NodeID)
  OnDisconnect(peer Peer, id NodeID)
  OnReceivePacket(peer Peer, packet []byte, id NodeID)
  OnStart(sim NodeArguments)
  ID() NodeID
  OnTerminate()
  TransmissionBehavior() TransmissionBehavior
}

type NodeArguments struct {
  Data      func() *map[string]interface{}
  UpdateID  func(newId NodeID)
  DelayBy   func(func(), time.Duration)
  Now       func() time.Time
  Terminate func(error)
  Log       func(message string)
  Loggers   []Logger
}
```

way for the simulator to inform the node when certain events happen, such as connections or disconnections to other nodes.

For each *Node* that is added to the simulation, the simulator constructs an *InternalNode* structure. This structure contains a lot of information on the node such as the current location, its nearby peers, the internal ID of the node, as well information about its movement. As the simulation progresses, these internal nodes move through space where different events can happen to them, such as gaining and losing peers. The list of *InternalNode* structures is the main way the simulator keeps state of the nodes.

6.2.1 Usage

As mentioned previously, the simulator can be used for modelling various different node behaviors. These node implementations have to follow the *Node* interface, as seen in listing 6.1. Here, the *OnConnect* and *OnDisonnect* inform the node that it has gained a new peer or lost a peer respectively. The *NodeID* is equivalent to the BLE address that the node broadcast, whereas the *Peer* structure has a public method *SendPacket* that allows the node to send packets to the peer. The *OnReceivePacket* method simply delivers a packet to the node that has been sent by the given peer. The *ID* and *TransmissionBehavior* methods stand out, since they are not to inform the node, but serve as a way for the simulator to extract information from the node. *TransmissionBehavior* will be discussed in section 6.4. The interface also contains *OnStart* and *OnTerminate* methods which inform the node that the simulation is starting or ending, and allows the node to initialize or clean up respectively. *OnStart* also gives the node a *NodeArguments* structure (see listing 6.1), which mainly consists of callback functions. These functions will not all be presented in detail, but generally they allow the node to either request data from or inform the simulator. The *Terminate* function can be called by a node if it experiences an error it cannot recover from, or to simply terminate the simulation because some test goal has been reached. Another function is *Log*, which is useful for using the logging system of the simulator.

Node implementations following the *Node* interface, can be added to the simulator by the *AddNode* method on the simulator. Once the nodes have been added, the simulation can be run by using the *Update* method. It takes as an argument *updateUntilTime* which represents the time that the simulation should be run until. When the *Update* method is called, the simulator goes through the events on the event queue that come before *updateUntilTime* and removes them from the queue as well as executes them in the correct order. The different types of event will be presented in section 6.2.2. Some events can cause other events to be added to the queue. Once there are no more events to execute within the given time, the function will return, however a new call to *Update* with a later time argument will continue the simulation from the point in time it stopped. The *Update* method can return an error, which indicates that the simulator has terminated with an error.

6.2.2 Events

While the events on the event queue has different types and different associated data, they all inherit from the *BaseEvent* structure. This simply has a time, a sequence number and a parent event. An overview of the different events will be given now.

TIMESTEP This event serves as the basic granules of time for node movement. When this event happens, the location of every node on the network is updated according to the movement profile of the node (see section 6.5). Once a TIMESTEP event is executed, a new one is added to the event queue with a delay of 10 milliseconds. This essentially means that node movement are updated every 10 milliseconds. A TIMESTEP event often results in CONNECT and DISCONNECT events being added to the event queue as well.

CONNECT A CONNECT event is emitted whenever two nodes that were previously not neighbors, come into range of each other. Once the event executes, the two implicated nodes receive an *OnConnect* call to inform them, and the simulator updates its internal data structure for peer connections.

DISCONNECT A DISCONNECT event happens when two nodes that are connected come out of reach of each other. The *OnDisonnect* method is called on both nodes to inform them of the event.

ADD_NODE The ADD_NODE event is put on the event queue when the *AddNode* method on the simulator is called. The *AddNode* method has the time at which the node should be added as one of its arguments. At the given time, the node is created in the simulator and CONNECT events will be issued to nearby nodes.

SEND_MSG This event is created whenever a node calls the *SendPacket* method on a *Peer*. It includes information about the packet, such as the data and the receiver. The reason why the sending of the packet might be delayed and not executed immediately is to model the buffer of messages at the sending node. If the node is sending multiple packets, these need time to transmit. Once the event is executed, a new RCV_MSG event is added to the event queue with some transmission delay. Delays will be presented in more detail in section 6.4.

RCV_MSG As mentioned, a RCV_MSG event is created by a SEND_MSG event. Once the RCV_MSG event is popped from the event queue, the packet associated with the event is delivered to the receiver via the *OnReceivePacket* method. An edge case is that the packet is dropped if the nodes are no longer within reach of each other.

DELAY The DELAY event serves as a way for a node to delay execution of an arbitrary piece of code. When created, the event is given a function, which is executed once the event is popped from the event queue. An example of when this is useful, is when a node has a timer, and needs to perform a certain action once it expires. Since the simulation time is disconnected from real time, if the node performed a wait of 1 second this wait would take 1 second of real time, whereas a DELAY event in the simulator would take 1 second of simulation time. A node can add a delay event by calling the *DelayBy* function from its *NodeArguments*.

TERMINATE This event determines the time at which the simulation should terminate. It allows for the simulator to terminate gracefully, by calling the *OnTerminate* method for all nodes. The TERMINATE event can also have an error associated with it, that is returned once the event occurs.

6.3 Reproducibility

Since we want to be able to use the simulator as a testing framework (see section 6.7.2), it is important that we can rely on the simulator to always produce the same result if it is given the same input. That is, we expect the simulator to ensure reproducibility. This way, we can set up complex tests and trust them to execute in the same way every time. It also allows us to reproduce bugs, which is useful during development.

While this property is nice, we of course need some way of obtaining randomness, e.g. to allow the nodes to move in varying random patterns or to simulate random transmission delays. We do this by using a pseudo-random number generator provided by `math/rand` in the Go standard library. This allows for seeding the number generator such that the sequence of numbers it produces is consistent across different simulations. If a consistent run is needed, the generator is given the same seed every time, and if a random run is needed, a random seed is given instead. The simulator can thus have stochastic behavior but still be consistent across runs.

This of course requires that every time randomness is needed, the same generator is used. The simulator achieves this by taking the random generator object *random* as an argument in its constructor, and utilizing this whenever needed. However, this is not enough, since the behavior of the nodes and the events they produce must of course also be the same for each run. That is, we require the node to also guarantee reproducibility for the result of the simulation to be consistent. This can be achieved by also passing the *random* generator to the nodes and using it consistently. As will be described in section 6.3.1, the node implementation for the Starling protocol does not currently live up to the criteria of consistent behavior. While this is the case, the simulator itself still provides this feature if used with another node implementation which does live up to these criteria.

6.3.1 Implementation Challenges

We will now discuss some of the problems related to guaranteeing that node behavior is consistent, which we discovered when implementing the nodes for the Starling protocol.

One problem we faced, is that when iterating over a map in Go, the order of the iteration is not guaranteed. This can introduce random behavior if the order of the iteration affects other parts of the simulation. This was solved by constructing a utility function that takes a map as well as the random generator and returns a list of the keys of the map in random order.

Another problem is that concurrency can introduce randomness in the order of execution of code. This can be solved by avoiding concurrency when not necessary, and otherwise by using the *DelayBy* function of the simulator if delays are needed.

The last and by far largest of the problems, is that our node implementation requires cryptographically random number generation when performing some cryptographic operations such as key generation. One strategy for solving this could be to use cryptographically secure generators for real use cases, and just provide a pseudo-random generator when using the protocol with the simulator. However, this does not work in practice.

We use functions from the `crypto/ecdh` package to generate key pairs when linking, and this implementation explicitly states that the returned key is not deterministically decided by the random generator given. This problem is amplified by the fact that the protocol uses the contact secret obtained when linking for encoding contact bitmaps, which means that the nodes encoded in a contact bitmap can vary depending on the value of the contact secret. This essentially means that with the current node implementation we cannot expect the simulator to give consistent results. This could potentially be solved by simulating the cryptographic primitives, but we did not look further into this.

6.4 Modeling Physical and Link Layer

The main focus when deciding how to model both physical layer and the link layer has been to keep it simple. The complexity of modelling the details of these lower layers means that it would be less efficient, and since we want the simulator to be able to handle many nodes, we generally choose to use a simplified model. We also evaluate the protocol by performing real life tests with physical devices, and thus we feel the need to model details on both the physical layer and the link layer are not essential.

6.4.1 Transmission Range and Delay

One of the major choices regarding the physical layer, is how signal propagation is modelled. The simulator has been implemented such that it is given a range in its constructor, which decides the maximum for how far nodes can send and receive data from each other. It is then up to the nodes to choose how to model the details of transmissions within this range.

A node must provide a structure following the *TransmissionBehavior* interface, which has a single function *Transmission*. This function is given the coordinate of the two nodes as well as the packet, and returns how much delay it should experience or whether it is dropped entirely. It is called whenever the node attempts to send a packet to another node. The simulator has a few different implementations for this interface included. Some are quite simple and have a fixed delay and a random chance of dropping packet. The one mainly used for testing the Starling protocol will be presented in section 8.2.

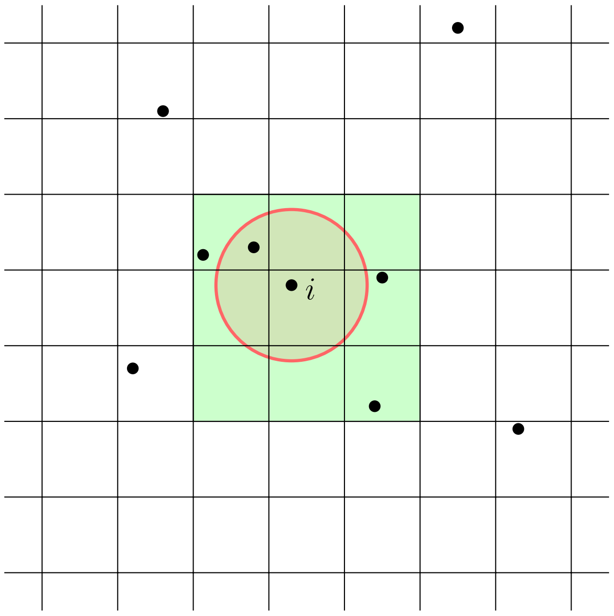


Figure 6.1: An illustration showing how space is divided into regions. The circle shows the range of node i, and the green squares show the squares that have to be considered when computing peers for node i.

Optimization

Computing whether a node is within a given range of another node is quite simple, since it simply requires calculating the euclidean distance between the location of the two nodes and checking whether this is less than the transmission range of the simulator. However, since the simulator should be able to handle many nodes, it needs to be efficient with how these range checks are performed.

Naively, we need to perform a range check for each pair of nodes. If the simulation contains n nodes, the number of checks to be performed is then in the order of $\mathcal{O}(n^2)$. Since this check is performed many times every second of the simulation, it is one of the main limiting factors when it comes to efficiency of the simulator.

Due to this, the simulator uses a strategy intended to minimize the number of required checks. It does so by dividing the plane into regions consisting of squares with length equal to the maximum transmission range of the nodes. Each node is assigned to the region that it is currently within. The simulator keeps a list for each region, that contains the nodes within it.

We are guaranteed that a node cannot connect to a node which is more than one square away from it (including diagonals), since the length of the squares are the maximum range. If we take the perspective of some node i, we only need to perform the range check for nodes that are within i's own region as well as the 8 regions around it. This can be seen illustrated in fig. 6.1, where the red circle represents the range of node i. The green area represents the regions where we need to perform range checks.

It is important to note that this optimization technically does not improve the worst case running time, in the case where all the nodes are within the same few nearby regions. In that case it likely performs worse, due to the overhead of updating the list of nodes for each region. On the other hand, if nodes are spread out, the amount of needed range checks are dramatically reduced. The amount of range checks to be performed could be expressed in the like $\mathcal{O}(kn)$, where k represents the maximum amount of nodes within a

3 by 3 region area.

6.4.2 Link Layer

The link layer, while intended to be comparable to BLE, has also been simplified quite substantially. There is no concept of advertising and actively scanning for other devices and discovering their services as in BLE. Essentially, if two nodes come within range of each other, they are both notified and can then send packets to each other. One aspect of the link layer that is modelled, is that each node has a buffer of limited size for outgoing packets. First of all, this means that a node needs to finish sending the first packet in the buffer before the second packet is sent, which essentially limits its throughput. It also means that nodes can potentially end up dropping packets before they are even sent, if their buffer is full.

6.5 Movement Model

One of the core aspects of MANETs is how they handle the physical movements of the nodes on the network. It is one of the main differentiating factors between MANETs and other categories of networks. How the nodes move is thus of great importance when designing the simulator and when evaluating the protocol. The simulator allows for nodes to have different movement models, by requiring that nodes supply a structure that follows the *NodeMovement* interface. This interface has a function for retrieving the initial position of the node, and a function *RegisterMovements* that supplies movement instructions. The movement instruction simply states the location a node is moving towards, and the time it will take for it to reach this destination. The simulator queries this function every time a node reaches its destination, to obtain information regarding the new destination.

The simulator must have movement models which are representative of how devices would move around in a real setting. One much used movement model is the *random waypoint* model, as first used in [27] for their simulations. The random waypoint model has come to serve as the standard for MANET research, and thus is very useful when comparing different results. However, it has been criticized as giving misleading results, since time-average results can depend considerably on how long time a test runs [54]. It could also be criticized for not modelling the movement of individuals realistically.

Since the Starling protocol is designed for a specific scenario, it makes sense to consider a movement model which closely models this scenario. One way of doing this is to simulate a crowd to gather the movement data for the nodes in the simulator. This strategy was used for evaluating the protocol and the details of the crowd simulation will be presented in section 8.2. The crowd simulation outputs the location of each node at different time steps. This is simply converted to a list of movement instructions, which the simulator can use to move the nodes around appropriately.

6.6 Visualization

As part of the implementation of the simulator, a visualization module has been developed. This module allows for a visualization of the movement of the nodes on the network, and even can be used to visualize what types of packets nodes are sending to each other. It has

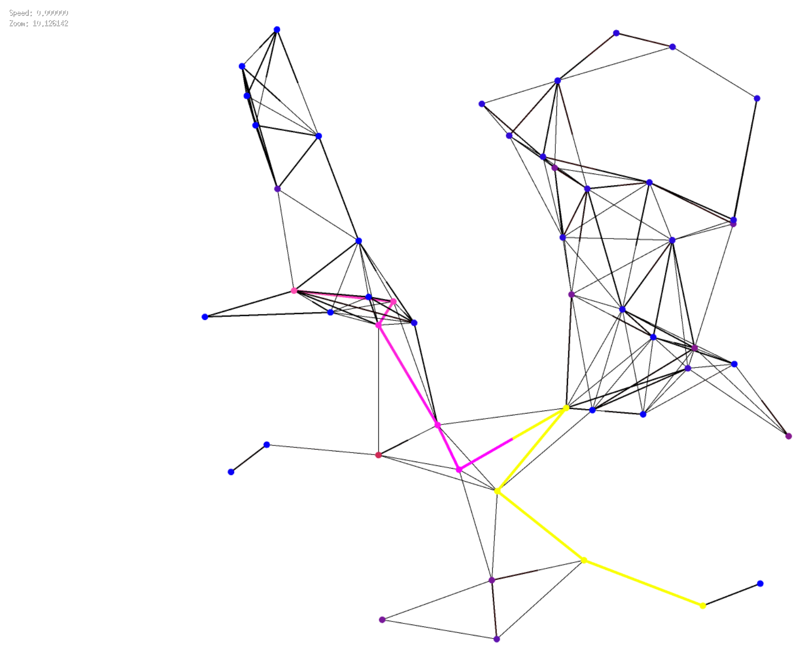


Figure 6.2: A screenshot from a visualization of a simulation.

been implemented using the Ebitengine game engine [22] written in Go. The visualizer has been very useful during development of the simulator and the protocol. Its primary use has been to verify that nodes move like we expect and that things generally seem to work as intended.

When the visualizer is initialized, it is given a instance of a *Simulator* and is responsible for using the simulator to run the simulation in the same manner as described in section 6.2.1. It allows for zooming and panning on the plane, as well as changing the simulation speed. A screenshot from the visualizer can be seen in section 6.6. Nodes are represented by dots, and the current connections they have with other nodes are represented by lines. The color of the node and connections can be used to indicate the state of it. In the given example, yellow indicates an active session along the connections, and purple indicates that a session packet is currently being forwarded along the path.

Nodes communicate their current state by the *Data* function of the *NodeArguments* object. This gives access to a map where the node can store arbitrary data, which the visualizer can read. If a node is transmitting a route reply, it can thus notify the visualizer which can then colors the node and the connection. Each node only controls half of the line representing the connection to a peer. This helps to show transmission delays and dropped packets, since the other half of the line will first change color once the packet is received.

6.7 Starling Node

So far, this chapter has mostly described the simulator in general terms, and how it could be used to simulate nodes following any Mobile Ad-hoc Network (MANET) protocol. This section will present the implementation of nodes following the Starling protocol specifically.

Once the Starling node is initialized, it creates a new instance of *Protocol*. It also creates an instance of *NodeDevice*, which is a structure that implements the *Device* in-

Listing 6.2: A selection of functions from the Starling node implementation.

```go
func (n *Node) OnConnect(peer simulator.Peer, id simulator.NodeID) {
  n.peers[idToAddr(id)] = peer
  n.proto.OnConnection(idToAddr(id))

  for _, scenario := range n.scenarios {
    scenario.OnConnect(n, idToAddr(id))
  }
}

func (n *Node) OnReceivePacket(peer simulator.Peer, packet []byte, id simulator.NodeID) {
  n.proto.ReceivePacket(idToAddr(id), packet)

  for _, scenario := range n.scenarios {
    scenario.OnReceivePacket(n, packet, idToAddr(id))
  }
}
```

terface. The Starling node lives up to the *Node* interface as given earlier in listing 6.1. A few functions from the implementation can be seen in listing 6.2. The node is quite simple and in some aspects it mostly serves as a binding layer between the simulator and the protocol. The fact that the Starling node simply forwards most events to the protocol for it to handle, means that the Starling nodes running in the simulation will act very similarly to physical devices running the protocol. For example, in the *OnConnect* function, it simply stores the peer with the given ID, and performs a call to the protocols *OnConnection* function to inform it that the node has a new peer. If the protocol then in response needs to send a packet to this new peer, it will use the *SendPacket* function on *NodeDevice*.

The Starling node also has functions for linking nodes and creating groups. For linking two nodes, it simply takes the two nodes as input and essentially simulates the linking process by calls to protocols *LinkingStart* and *LinkingCreate* methods. For groups it takes a list of nodes, creates a group secret, and calls the *JoinGroup* protocol function for every node. These functions allows a user to simply create multiple Starling nodes and link them, after which they can be added to the simulator and the simulation can be started. That simulation would of course be rather uninteresting, since none of the nodes will be prompted to send data packets by default. This problem is solved by the concept of scenarios which will be introduced now.

6.7.1 Scenarios

Scenarios are used when we want to orchestrate scenarios where nodes intend to send messages to each other. We define the *Scenario* interface as seen in listing 6.3, as a way to affect the behavior of the nodes. As seen earlier in listing 6.2, whenever nodes receive calls to certain functions, they notify the scenarios that have been added to the node.

Scenarios can be added to the node by using the *AddScenario* method, which allows the scenario to react to events. The Starling node implementation contains many scenarios and a few will be presented here. An example of one such scenario, is the *ScenarioSession*. This scenario is given a contact and when its *OnStart* function is called, it attempts to establish a session with this contact. It does so by sending route requests every so often until it is succesful. It also contains a list of subscenarios of the type *ScenarioSendData* which are activated once the session is established. Following this, future events will thus

Listing 6.3: The *Scenario* interface.

```go
type Scenario interface {
  OnStart(node *Node)
  OnConnect(node *Node, address device.DeviceAddress)
  OnDisconnect(node *Node, address device.DeviceAddress)
  OnLog(node *Node, message string)
  OnReceivePacket(node *Node, packet []byte, address device.DeviceAddress)
  OnReceiveData(node *Node, data []byte, session device.SessionID)
  OnSessionEstablished(node *Node, session device.SessionID, contact device.ContactID, address
  ↪  device.DeviceAddress)
  OnSessionBroken(node *Node, session device.SessionID)
  OnProcessMessage(node *Node, session device.SessionID, message []byte)
  OnMessageDelivered(node *Node, messageID device.MessageID)
  OnSyncStateChanged(node *Node, contact device.ContactID, stateUpdate sync.Model)
  OnTerminate(node *Node)
}
```

Listing 6.4: An example of using scenarios to compose node interactions.

```go
nodes[a].AddScenario(
  node.DelayScenario(sessionA.SendDataScenario("ping"), sendDelay))

nodes[b].AddScenario(
  node.EventScenario(sessionB.ReceiveDataEvent("ping")).
    OnEvent(sessionB.SendDataScenario("pong")),
)

nodes[a].AddScenario(node.EventScenario(sessionA.ReceiveDataEvent("pong")).
  OnEvent(node.ActionScenario(func(node *node.Node) {
    statLogger.LogStatEvent(statistics.NewPingPongSuccess(nodes[a], nodes[b]))
  })))
```

be forwarded to these subscenarios.

Another type of scenario is *ScenarioEvent*, which provides a way for activating subscenarios based on when certain criteria are met. These subscenarios, can also include subscenarios of their own, which leads to an ergonomic way of composing complex scenarios. This can help to model communication between the nodes, as seen in listing 6.4. Here it is assumed that the two nodes have been linked and both have a *ScenarioSession*. First, node A is assigned a *DelayScenerio*, which essentially just sends a ”ping” message at a certain delay. Node B is given an event scenario which waits until the node receives a message with the content ”ping”, and then replies with a ”pong”. Lastly, node A logs a success when it receives the ”pong”. All of the scenarios are specified and given to the node before the simulation is started.

6.7.2 Simulator Tests

As mentioned earlier in this chapter, the simulator has been useful for validating that the Starling protocol behaves as expected. One aspect of this is that we can use the simulator to construct tests for the protocol, which can be used to verify that everything works after making new changes to the protocol. The scenario concept has been used heavily in the creation of these simulator tests.

To create a test, one needs to create an instance of the *Simulator* as well as the relevant Starling nodes. If the test involves the two nodes sending messages to each other, they need to be linked and located within range of each other. The relevant scenarios can then be added to the nodes, to specify how they are supposed to behave. After this, the simulation can be started and run a fitting amount of time. Once the simulation has been run, we need to assert that the things we expected to happen have occured. This can be done by querying the scenarios. An example of this is the *DidTrigger* method for the *ScenarioEvent*. Thus, at the end of the test, we can assert that certain events have triggered, e.g. that a message was received.

Chapter 7

Application

The protocol has been implemented as a smartphone application to test how the specification will work in practice. The app has been implemented to run both on iOS as well as Android, in order to show that the technology can work across different platforms. The app uses the Bluetooth Low Energy (BLE) technology as it is made to run in the background and is supported by both iOS and Android.

As a proof of concept we decided to make the application as a general purpose messenger app, where users can link and become contacts in order to chat with each other. Each chat has a message history where the user can read the entire conversation from start to end. Users can create group chats and invite members, where everyone's messages are combined into a single group conversation. The source code for the application can be found online.[1]

7.1 Architecture

The mobile application is made up by multiple components that work together. The overall structure can be seen on fig. 7.1. At the top we have the *View* component, which handles the user interface. The user interface is built using the React Native framework [35]. This allows us to implement the view component only once and run it on both platforms. Events are sent back and forth between the *View* and the *Adapter* component. For instance, when the user sends a message, that will cause the view to send an event to the adapter which can then pass it further on to the *Protocol* and *Bluetooth* components. Likewise, when a Bluetooth packet is received with a message, it will cause the adapter

[1] `https://github.com/starling-protocol/starling-messenger`

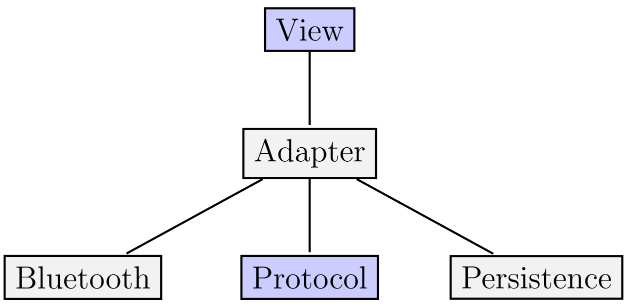


Figure 7.1: General architecture of the mobile application. Components in blue are cross-platform.

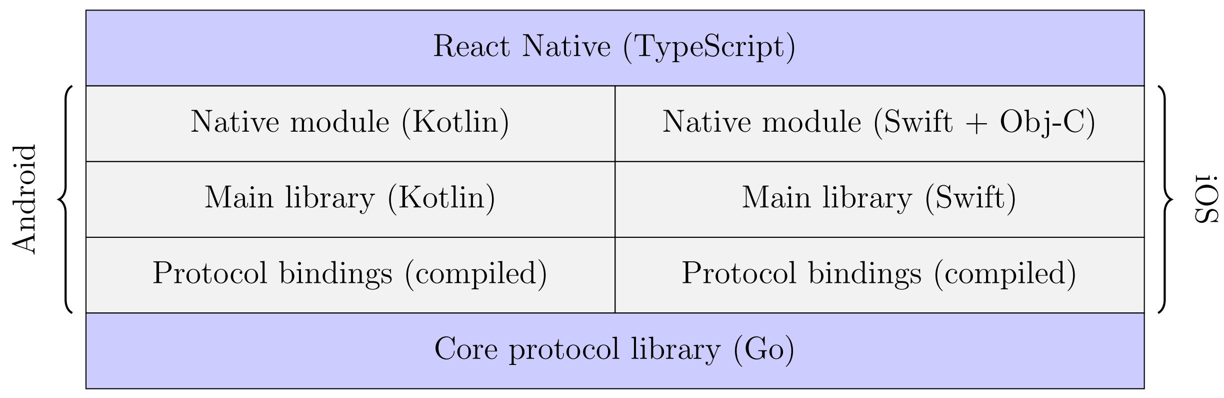

React Native (TypeScript)
Native module (Kotlin)
Native module (Swift + Obj-C)
Main library (Kotlin)
Main library (Swift)
Protocol bindings (compiled)
Protocol bindings (compiled)
Core protocol library (Go)
Android
iOS

Figure 7.2: The structure of modules that make up the application. Modules in blue are cross-platform, whereas gray modules are platform specific.

to send an event to the view, so the view can present it to the user. The view component is further described in section 7.4.

This event based architecture, decouples the interaction between user interface interactions and the effects it causes. It is also beneficial since the protocol is highly event based. For example, an event will be sent when a new session has been established.

Every other component only interacts with the adapter, which then forwards events when necessary to other components. Below that, we have the *Bluetooth*, *Protocol* and *Persistence* components, which manages their respective tasks.

Protocol. This component entails the core Protocol library. It is written in Go and can cross-compile to Android and iOS. This library is also used in the simulator described in chapter 6. This allows us to test it thoroughly with simulations and be confident that the results will also work in practice. How the protocol is integrated in the application is described in section 7.1.1. The protocol itself is described in chapters 4 and 5.

Bluetooth. The Bluetooth component is responsible for handling connections to other devices and is used to send and receive packets to and from other devices. Section 7.2 describes this component in depth.

Persistence. The persistence component is responsible for saving state to storage in a secure way, such that contacts and message history is not lost when the app is restarted. Section 7.3 describes this component in depth.

In practice the application is split up into various modules and libraries as seen on fig. 7.2. At the bottom, we have the core protocol library which is cross compiled for both platforms. Similar, at the top we have the view component made with React Native, which is also cross compiled for both platforms. However, the modules in between are written specifically for both platforms. First, the protocol is compiled to a platform specific library with bindings for the given platform. These bindings are imported by the main library, which is where the implementation for the Adapter, Bluetooth and Persistence components lies. The main library is compiled and included into the native module, which is a binding layer between the main library and the view component. The main library is designed such that it could be used as an SDK for any mobile application wanting to use the protocol.

7.1.1 Protocol

The core protocol library is compiled to two separate libraries, one for iOS and another for Android. This is done using the Go Mobile tool [48], which automatically compiles the library and generates bindings to each platform.

A thin wrapper around the real protocol module has been made in order to make it compatible with the Go Mobile tool. The wrapper mostly converts between primitive types needed for the Go Mobile tool and the Go types used by the protocol. A wrapper for the device interface seen in listing 5.2 has also been created. Again it converts the types to the required primitive types. Some of the methods of the interface can be implemented directly in Go. This includes *Rand()* and *CryptoRand()* which are used to generate random numbers. The *Now()* method is also implemented in Go and it simply returns the current timestamp. This method is relevant for the simulator, where the simulation time is returned instead. Lastly, the *Delay(action, duration)* method of the device interface is implemented to start a Goroutine (a lightweight thread) that sleeps for the given duration before it executes the action closure.

7.2 Bluetooth

In order to get the needed control over the Bluetooth communication, it was necessary to implement the Bluetooth component manually for both platforms in native code. On iOS the Core Bluetooth framework [6] is used, which is Apple's API used to interact with Bluetooth.

On Android we instead chose to use a third-party library by Nordic Semiconductor [36] which makes it easier to work with Bluetooth rather than working directly with the operating system level APIs. We chose to do this since the first-party Bluetooth API of Android is very low-level and requires a lot of extra work which we could offload to the third party library while still being flexible enough to implement the specification.

The Bluetooth and Protocol components from fig. 7.1 interact with each other through the Adapter. For instance, when a connection to another device has been established or a new message has been received, those events will be forwarded to the Adapter, which will forward the event further along to the Protocol component. Likewise, when an event from the Protocol component is emitted, such as when a new packet should be sent. That event is received by the Adapter where it will be forwarded along to the Bluetooth component, which will make sure to send it to the targeted device.

The specification in appendix A.6.1 details how the bluetooth connections are configured and how data is transferred. Essentially, each smartphone acts both as a BLE central and peripheral. The peripheral advertises itself and the central scans and connects to other peripherals. The peripheral starts advertising a *service* with a UUID unique for the protocol. The service contains a *haracteristic* that is used for transferring data when another device connects to it.

7.2.1 Structure of the Bluetooth Component

The structure of the Bluetooth component will be described from the perspective of an iOS device, as the structure is nearly identical for Android.

The *BluetoothManager* class keeps track of everything related to the Bluetooth component. It manages the central and peripheral of this device. The BluetoothManager has the *startAdvertising* and *stopAdvertising* methods that are used to start and stop the entire system. First when *startAdvertising* is called, will the central and peripheral managers be created and started. When this is done, these two managers can start to send and receive Bluetooth events that are handled by the BluetoothManager.

The BluetoothManager holds a set of *connecting peripherals* and a set of *configuring peripherals*. When our central manager scans and finds another peripheral to connect to, an event is sent to the BluetoothManager which immediately attempts to connect to the peripheral. When this is done, the peripheral is added to the set of connecting peripherals. When an event is sent notifying that the peripheral has been connected, it is moved to the configuring peripherals set, until it has been fully configured such that data can be exchanged with it. When it has been fully configured, it is moved to the *ConnectionsTable*.

The connections table stores all active connections to other peripherals and centrals in a map indexed by the device address. It is possible to simultaneously have a connection to the peripheral and central of another device. In this case, there is simply two channels that can facilitate communication to that device. We chose to allow this, since it is not apparent which channel should be prioritized since the connection is flipped for the other party, ie. their peripheral is connected to our central and vice versa.

Because both the peripheral and central can be used for communication, we have made an interface called a *Peer* that specifies the required methods needed to facilitate communication. Both a peripheral and a central connection implements this interface. This allows us to send data to a peer without worrying about whether the data is transmitted by the peripheral or the central.

The *Connection* structure can hold a reference to either a peripheral, a central or both with the same device address. The structure has a *sendPacket(packet)* method that will send the packet using either the peripheral or the central. This generalizes the concept of a connection to another device.

The BluetoothManager exposes a public method *sendPacket(address, packet)*, to the Adapter component, that takes a device address and a byte sequence. It first finds the connection by the device address in the connection table and calls the *sendPacket* method of the found connection.

7.2.2 Background Connections

The app should continue to work even when running in the background, for example when the phone is in your pocket. Smartphones are very strict on what is allowed to run in the background and for how long in order to save battery.

On iOS this is done by adding a background mode capability to the application. This will allow BLE events to be called when the phone is in background mode. The Apple documentation states that it is important to return quickly from these events in order to save on battery, and the process might be killed by the operating system if it takes too long.

On Android a Foreground Service is started. This starts a new process that keeps running even when the phone enters background mode. The Bluetooth component runs inside this service, such that it keeps running when the screen is off.

7.2.3 Bluetooth Interactions Between Platforms

When iOS runs in background mode, it changes the way BLE advertises services. Instead of advertising the services following the BLE specification, it collects all active services into a single advertisement packet by utilizing an "Overflow Area" of the packet in order to optimize the number of advertisement packets to send. Apple automatically manages this for iOS devices, which makes this optimization seamless for iOS to iOS devices. However, when an Android devices scans for other devices, iOS devices running in this background mode will not be detected. We had to develop a custom scanner filter in order for Android devices to detect iOS devices in background mode.

These special advertising packets have been reverse engineered and presented in [55]. Essentially Apple generates a bit-index for each background service as a function of the UUID of the service. These bits are then collected in a bitmap that is encoded in the overflow area of the advertisement packet. However, Apple provides no documentation of how the bitmap works, and thus the mapping from service UUIDs to bit indexes is proprietary and not public. In order for us to find the bit index of our service, we started the service in the background of an iOS device and decoded its advertisement packets to identify the bit index. Knowing this bit, we can decode the special advertisement packets on Android and attempt to connect to it if a match was found.

7.3 Persistence

The Persistence component in fig. 7.1 is responsible for storing all relevant data to disk such that it can be restored whenever the app is restarted. The component is split up into two parts. One that is responsible for persisting all contact secrets, and another that is responsible for persisting the message history of links and groups.

In order to persist contact secrets the native key store framework for each platform is used. For iOS this framework is called Keychain and for Android it is called Keystore. These frameworks act as a secure database meant to store cryptographic keys and passwords. When a new contact is created, its associated secret is stored in this database and can securely be retrieved whenever it is needed. The key stores allow for varying degrees of access control, with the most strict requiring the user to enter a password every time the secret is needed. Since our app needs to have access to the keys when running in the background, we needed to relax the access control to allow accessing keys when in background mode.

Message histories are stored on disk where each link or group is stored in a separate file. The files are stored in private mode, which means that no other app can access these files and they are automatically encrypted whenever the phone is locked. By letting the operating system take care of the security of these files, we both get the best security and the best user experience, as the user does not have to enter a password or similar whenever the app is started.

7.4 View

The View component of the architecture in fig. 7.1 is built using React Native [35]. It is an open-source framework developed by Facebook, used to build cross-platform mobile user interfaces. React Native runs on top of JavaScript, which means that extra work is needed in order for the View to communicate with the Adapter component. This is done through a React Native Module. This is essentially a native library that can be imported into the JavaScript code, and a translation layer between JavaScript and the native code is automatically generated by React Native.

7.4.1 User interface

The user interface consists of a home screen where all contacts and groups are shown. From here, the user can navigate to one of the contacts or groups to enter the chat view. Figure 7.3 shows six screenshots of the application.

Since messages can not be sent reliably as the recipient must be nearby, it is important to give the user as much relevant information as possible regarding the current state of the app. For example, the app highlights contacts that have an active connection and shows them at the top of the list. This helps the user better understand what they can expect when e.g. sending a message to a contact. Another information the app presents is which recipients have received a message. Since the recipients do not always receive the message immediately, this information is very useful to the user.

Home screen

A screenshot of the home screen can be seen on fig. 7.3a. It shows a list of all the contact and group chats the user has. Contact and group chats are differentiated by the shape of their icons. Contacts have a round icon where groups have a rounded rectangle shaped icon.

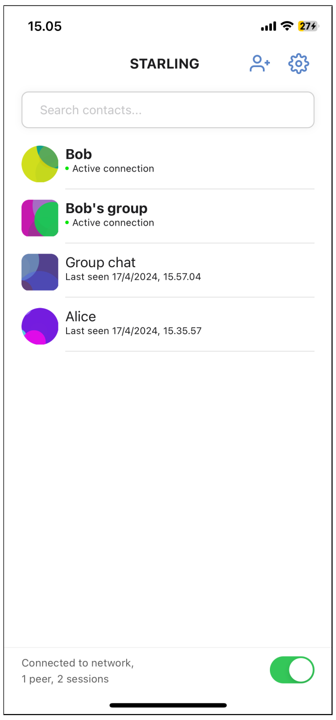
It shows an active connection to another user named Bob. The contact Bob along with a common group is highlighted at the top with an indication that there is an active connection for both of the chats. This indicates that the two devices have established an active session and communication between the devices can take place. For users without an active connection, the time and date of the last time the devices where connected is shown instead. This makes it easier for the user to for example determine how out of sync a conversation might be.

At the bottom of the home screen is the footer. Here the user can connect and disconnect from the network. When connected, the number of active peers and sessions is shown in the footer. The number of active peers refers to the number of active Bluetooth devices which are currently connected to this device.

Chat screen

When the user clicks on one of the contacts or groups, they are presented with a history of the messages in that chat. Figure 7.3b shows a screenshot of a chat with a single contact, whereas fig. 7.3c shows one for a group.

Next to each message is some associated metadata. For instance the timestamp of when the message was created and the name of the sender. It also shows whether the



15.05

STARLING

Search contacts...

Bob
● Active connection

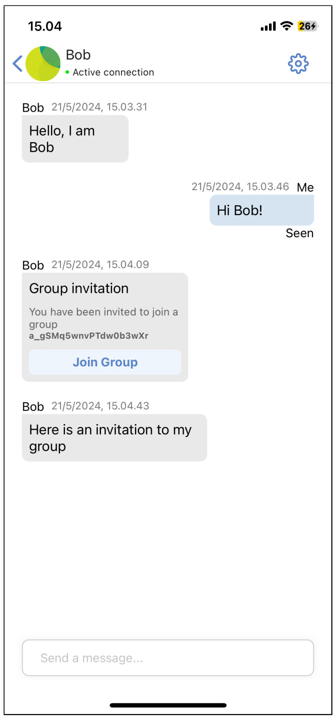
Bob's group
● Active connection

Group chat
Last seen 17/4/2024, 15.57.04

Alice
Last seen 17/4/2024, 15.35.57

Connected to network,
1 peer, 2 sessions

(a) Home screen



15.04

Bob
● Active connection

Bob 21/5/2024, 15.03.31
Hello, I am Bob

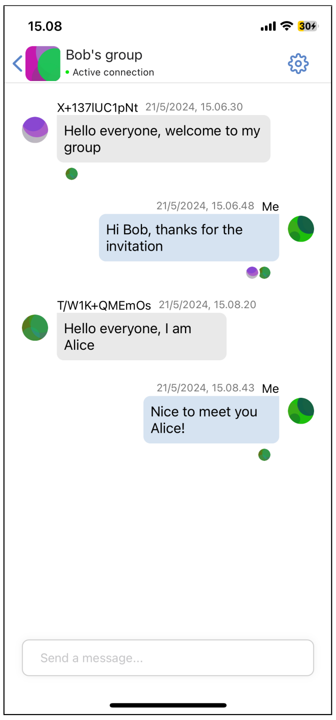
21/5/2024, 15.03.46 Me
Hi Bob!
Seen

Bob 21/5/2024, 15.04.09
Group invitation
You have been invited to join a group
a_gSMq5wnvPTdw0b3wXr
Join Group

Bob 21/5/2024, 15.04.43
Here is an invitation to my group

Send a message...

(b) Contact chat screen



15.08

Bob's group
● Active connection

X+137IUC1pNt 21/5/2024, 15.06.30
Hello everyone, welcome to my group

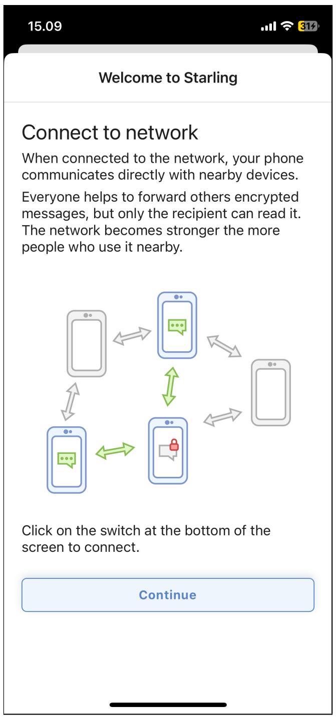
21/5/2024, 15.06.48 Me
Hi Bob, thanks for the invitation

T/W1K+QMEmOs 21/5/2024, 15.08.20
Hello everyone, I am Alice

21/5/2024, 15.08.43 Me
Nice to meet you Alice!

Send a message...

(c) Group chat screen



15.09

Welcome to Starling

Connect to network

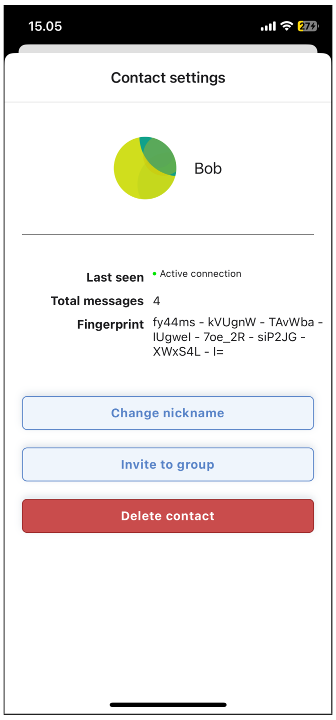
When connected to the network, your phone communicates directly with nearby devices.

Everyone helps to forward others encrypted messages, but only the recipient can read it. The network becomes stronger the more people who use it nearby.

Click on the switch at the bottom of the screen to connect.

Continue

(d) Welcome popover



15.05

Contact settings

Bob

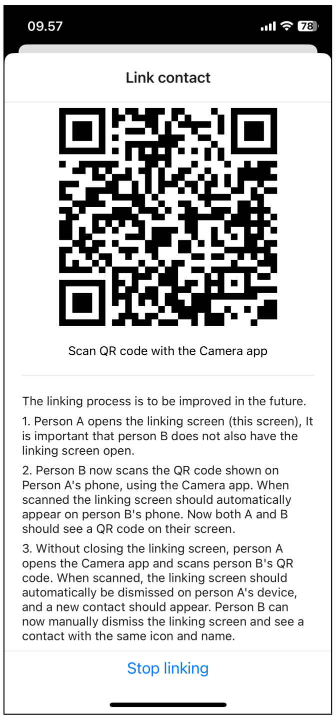
Last seen	● Active connection
Total messages	4
Fingerprint	fy44ms - kVUgnW - TAvWba - lUgwel - 7oe_2R - siP2JG - XWxS4L - I=

Change nickname

Invite to group

Delete contact

(e) Contact settings popover



09.57

Link contact

Scan QR code with the Camera app

The linking process is to be improved in the future.

1. Person A opens the linking screen (this screen), It is important that person B does not also have the linking screen open.

2. Person B now scans the QR code shown on Person A's phone, using the Camera app. When scanned the linking screen should automatically appear on person B's phone. Now both A and B should see a QR code on their screen.

3. Without closing the linking screen, person A opens the Camera app and scans person B's QR code. When scanned, the linking screen should automatically be dismissed on person A's device, and a new contact should appear. Person B can now manually dismiss the linking screen and see a contact with the same icon and name.

Stop linking

(f) Linking popover

Figure 7.3: Six screenshots of the smartphone app running on an iPhone.

message has been received by the contact or not by showing a "Seen" label below the message if it has been acknowledged.

Group chats additionally show an extra icon for each member of the group. This is necessary for the user to differentiate the different members in the group. For groups, each member known to have received a message is instead shown by adding the icons of the members known to have received the message, instead of the "Seen" label.

Users can invite new members to one of their groups by sending an invitation with the group secret to a chat. The third message in fig. 7.3b is an example of such an invitation. When the user clicks the "Join Group" button, they will automatically join the group and connect to every reachable member and attempt to synchronize the entire message history.

Welcome popover

When the user opens the app for the first time, they are presented by a "Welcome to Starling" popover, as seen in fig. 7.3d. This screen shows a list of pages that the user can swipe through and learn what the app does and how it works.

It was important for us to add this guide to the app, since the app works in quite a different way compared to traditional messaging applications that use the Internet. It is meant to quickly give the user a better understanding of what to expect from the app.

It starts by presenting the concept of linking and that users must scan each others QR codes. Next it explains briefly how the network works, and the concept of message forwarding. Lastly, it explains how messages are sent and how groups work.

Link contact popover

The user can press the link icon from the home screen in order to open the link contact popover, as seen in fig. 7.3f. A QR code is presented which can be scanned by another user in order to link and become contacts. The linking process is more complicated than we would have liked, because the built-in camera app is used for scanning the code.

Ideally, we would have liked to have a scanner built into the application, however during development of the app, we could not find any camera component libraries that could do the job well and worked with our tech stack. Instead of spending too much time on implementing this ourselves, we instead focused our time on other aspects of the project. This is an obvious potential improvement for future development.

7.4.2 Implementation

The view is structured into various modules. The view itself made with React. The state is stored in a single global model using the Redux toolkit [1], and provides a single source of truth. Whenever the relevant parts of the model changes, the view is updated to always reflect the state of the model.

The model is for the most part updated as a result of an event from the Adapter component. When the application is started the *registerNativeEvents* function is called, which registers an event listener for each event that can be sent by the Adapter. The event listeners will for the most part convert the event into an update of the global model. For instance, when a session has been established, the contact associated with the session is set to be online. Later when a session broken event is emitted, the contact will be set to offline and the timestamp is stored in order to show the user when the contact was last active.

Synchronization state

The synchronization model, as described in section 5.5.1, consists of a two-dimensional dictionary that maps a public key and a version number to a message, $(PublicKey \times Version) \mapsto Message$. However, in order to present the messages in to the user, the messages in this dictionary must first be converted into a correctly ordered list.

First, every message is extracted into a single flat list with the associated public key and version added to each message in the list. Afterwards the list is sorted by their associated version numbers, which gives the messages a causal ordering. In the case where two messages have the same version number, they will instead be ordered by their timestamp. This attribute is less reliable, since the clock of different phones can be out of sync, but it works great as a tie-breaker as it will result in the same ordering of the messages for everyone.

Identicon

Small icons are automatically generated for contacts and groups. These icons are generated based on a hash of their shared secret. This is done by using the hash as the seed for the pseudo-random number generator used. The icons consist of an SVG image with a circle or rectangular cutout. The background color is chosen randomly by selecting random values for the red, green and blue color channels. Within the cutout, four random circles are generated with a random position, color and transparency.

This gives us visually pleasing icons that the user can quickly identify. Because the same hash is used for a given contact, the corresponding icon will look the same on different devices.

Chapter 8

Evaluation

In this chapter we will evaluate the Starling protocol. This has been done from multiple different perspective with different methodologies. We start by analyzing and evaluating whether the protocol lives up to the security requirements presented in section 3.2. We then discuss the methodology for simulating the protocol in a realistic environment, and present and evaluate the results of these simulations. Furthermore, we also evaluate the protocol by performing a user test and discuss the results of it.

8.1 Security

This section will analyze the defined threats and security requirements in sections 3.1.3 and 3.2, and argue how the protocol ensures that these requirements are met.

8.1.1 Data Leakage and Confidentiality

The protocol ensures confidentiality for all layers above the network layer, i.e. transport and application layer. Transport layer packets can only be contained in a network layer session packet or be piggybacked in a route reply. In both cases, the contents is encrypted and authenticated using AES-GCM, using a shared secret previously obtained from a Diffie-Hellman key exchange in the linking phase. As indicated in section 3.1.2, we assume that adversaries do not have the ability to break AES.

The remaining network layer packets (route requests and route errors) do not carry any user data, so there would not be any data to be leaked.

8.1.2 Intelligence Gathering and Anonymity

Communicating nodes on the link layer are identified by a MAC address. This means that an adversary can record these addresses and re-identify them at a later point. This is generally circumvented by smartphones changing the MAC address of the device at regular intervals. Due to this, the information is only useful for an adversary in a relatively short period of time. As discussed in section 3.1.2, we assume that the attacker generally only has a local view, and thus will have difficulty tracking nodes across different MAC addresses.

We will now argue why each different network packet does not reveal any identifying information, and thus helps provide anonymity.

A route request contains a request ID, TTL, ephemeral key and a contact bitmap. Since the request ID and ephemeral key both are randomly generated uniquely for each route request, there is no way to tie that information to anything other than the route request itself. The contact bitmaps are specifically designed to be anonymous, see section 4.3.5. If the initial TTL is set to a fixed value, this could reveal the distance the route request has traveled so far and whether the sender of the route request is the initiator. This could potentially be mitigated by randomizing the TTL when constructing a route request.

A route reply contains a request ID, session ID and an ephemeral key. The session ID and ephemeral key are both randomly generated uniquely for each route reply and thus reveal no information. The request ID links the reply to the request, but this information is essential in order for the intermediate notes to establish the route. The rest of the packet consists of a nonce, payload and authentication tag which makes up the AES-GCM encryption. Since the session secret is only known by the two contacting parties, this makes it indistinguishable from random bytes for everyone else. Section 4.3.4 describes the anonymity of sessions in more details.

A session packet only contains a session ID and AES-GCM encrypted data. The session ID ties together all session packets for the same session as well as the initial route request, reply packet and potential route errors. This leaks how much traffic is sent through each session and for how long each session lasts. However, it does not reveal anything else about the content of the data.

8.1.3 Impersonation and Authentication

The first link in the chain of trust is the linking process, described in section 4.3.3. The protocol achieves authentication when linking by having each party scan the other parties QR code, and then perform a Diffie-Hellman key exchange with the public keys contained in the QR codes. Using the Diffie-Hellman algorithm we ensure that even if an adversary sees the two QR codes, they will not be able to construct the shared secret. This means that an adversary can only get a contact secret with another user, if they somehow make the user scan their own QR code.

On the network layer both route reply and session packets are fully authenticated by the use of AES-GCM. The upper layer data is encrypted and the plain text metadata such as the session ID is included in the info tag, which means that it will be authenticated together with the encrypted data in the authentication tag. Since the session secret used as the key is only known by the two communicating parties, and since it cannot be computed without the contact secret obtained by linking, the packets are ensured to be authenticated. A more detailed analysis can be seen in section 4.3.4. The same goes for groups, however the contact secret of the group is known by everyone in the group, so messages are only authenticated for the group and not the individual members. This means that members within a group communication can impersonate each other and an extra layer of authentication is needed for each member. An example of this can be seen in the synchronization extension, where this form of authentication between group members is performed.

A route request is not authenticated, but it is designed in a way where it does not need to be. Even if the route request was impersonated, the following route reply would be useless for the impersonator, as it would be encrypted such that only the real identity could read it and reply to it.

8.1.4 Disruption and Availability

The routing part of the protocol has been specifically designed to be resistant against disruption. Proactive routing algorithms (section 2.1) often do not work well in untrusted environments, as nodes trust each other in order to build a map of the routes. For reactive routing, the route can either be stored in the packets themselves, like with DSR described in section 2.2.1. Or it can be stored at the nodes locally, like with AODV described in section 2.2.2. We chose to build the protocol on the latter model, where state is stored at the nodes. That way, trust is distributed out to the individual nodes and no single point can control it easily.

Each route request is flooded throughout the network by every node within range, which can cause a lot of route requests on the network. This makes it an obvious tool for an adversary to disrupt the network by overloading the network with route requests, known as a flooding attack (see section 2.3.4). While this attack is very hard to mitigate against, one means of mitigation is to throttle route requests from a given node by beginning to ignore route requests from the same node if the rate of which they are received becomes too high. We have not explored this in this project, but it would be suitable for future development.

Another potential attack using route requests would be to poison request ids. An adversary might, upon receiving a route request, craft a new invalid route request with the same request ID and forward that instead, in the hope that it would be spread throughout the network faster than the original one. Since the protocol will ignore route requests with duplicate request IDs, this could cause the original route request to be ignored. Although this attack is theoretically possible, we do not believe that it would be very efficient in practice. Especially since a node can simply send a new route request.

The route error packet which is used to notify that a route no longer exists, could also be a potential threat for misuse. If an adversary were to send malicious route errors, it could be used to break healthy sessions. This can easily be mitigated, however, by verifying that the route error is sent by an intermediate note on the route, and ignoring the packet otherwise.

Both session packets and route replies are authenticated with the session secret which is only known by the two communicating parties. This means that if an adversary attempts to construct or change such a packet, the authentication would fail, and it would be ignored.

It is impossible for an adversary to perform a black hole attack (section 2.3.1), since route replies are authenticated. If an adversary were to instantly reply with a fake route reply upon receiving a route request, the initiator of the route request would simply ignore the reply, when it fails to authenticate the packet.

However, the protocol does not prevent gray hole attacks (section 2.3.2), where an adversary only allows the route request and route reply to pass through, but drops all packets afterwards. This would allow the session to be established, but not allow data to be sent over it. After a timeout the session will automatically be broken since no acknowledgements has been delivered and the nodes would attempt to find a new path. Unfortunately, the new path could possibly go through the same adversary node and the attack could be repeated. Only when the adversary is no longer on the shortest path between the communicating parties will the disruption end.

A gray hole attack could be amplified by a wormhole attack (section 2.3.3), where route requests are teleported between adversaries using out of band communication in order to get a further reach. This could be used to get a better chance of being on the shortest path between the communicating parties. However, this attack is quite involved to carry out in practice, as multiple adversaries need to work together in order to achieve it.

Although the protocol is susceptible to gray hole attacks, we do not believe that this poses a big threat as the adversary needs to be on the shortest path between the communicating parties. Since nodes are expected to move around, the attack will not be very consistent.

8.2 Simulation

As presented in chapter 6, the simulator allows the nodes to provide custom implementations for movement, drop rates and delays among other things. The implementations used in this evaluation will be presented in this section. After this, we will present the details of the different scenarios that we have simulated.

8.2.1 Transmission

All simulations presented in this chapter use the same transmission behavior, named *LongTailTransmission*. The core idea behind it is to model packet delays that are generally quite short, but sometimes can be very long. The Pareto distribution is used to accomplish this. It was chosen since it is quite simple and thus efficient to calculate, while still providing a realistic estimate when modelling delays [56]. The *LongTailTransmission* also randomly drops 1% of packets, to simulate packet drops.

8.2.2 Movement

The movement of the nodes depend heavily on the scenario we want to simulate. For some scenarios we have used variations of the random waypoint model, while for other we have used movement data gathered from crowd simulations.

The open-source crowd simulation framework Vadere [29] can be used to simulate pedestrian and crowd dynamics. For some of the simulations, Vadere has been used to simulate realistic protest scenarios. It takes a file describing the scenario to be simulated, which includes location of nodes, the goals they are walking towards, obstacles and other relevant information.

The way we generate realistic obstacles is by exporting geographic data from OpenStreetMap [38]. Using this as the foundation, a protest can be simulated by generating pedestrians with a series of common goal locations. Once a simulation has been run, a file containing the movement of each node is automatically generated. The simulator can use this output file as input for a movement model.

8.2.3 Protocol Options

When developing the protocol specification, some behaviors of the protocol was intentionally left vague, such that implementations can choose to implement these in different ways. This is often the case when there is not a clear-cut best way of doing something.

For some parts of the Starling implementation, we have thus had to make decisions which could have an impact on the performance of the protocol. In some instances, we have implemented it multiple ways and allow the user of the protocol to use the implementation they want. The user chooses this by supplying a *ProtocolOptions* structure (see listing 8.1) when constructing the protocol. The protocol options are also used when running the protocol in the simulator. Doing this, we can see how the protocol performs with different behaviors. When describing the options, we will also state what the default values are. The different protocol options will be described now.

EnableSync When true, the sync extension is enabled which means that nodes will attempt to synchronize messages. This is true by default.

DisableAutoRREQOnConnection Normally, when two nodes come within range of each other they will send a route request with a TTL of 1 to each other to discover if they are contacts. If *DisableAutoRREQOnConnection* is true, this behavior is not performed. It is generally false by default.

MaxRREQTTL This integer defines the TTL used when sending route request. If a node receives a route request with a TTL larger than *MaxRREQTTL*, it will also set it to *MaxRREQTTL* before it forwards the request. The default value of it is 10.

RREQBroadcastStrategy This option defines how nodes should forward route requests. *BroadcastAll* means that it should be broadcast to all the neighbors of the node, except for the node it received it from. *BroadcastTwo* means that it should simply forward the route request to two of its neighbors. *BroadcastLogFunc* means that it should forward it to the number of nodes given by the function $\min\{N, \lfloor \log_2(N) + 1 \rfloor\}$, where N is the number of neighbors. The default behavior is *BroadcastAll*.

ForwardRREQsWhenMatching When this option is true, a node forwards a given route request even if it finds a match in the contact bitmap and sends a route reply. This can be reasonable as multiple contacts can be encoded in a single contact bitmap. This option is false by default.

ACKDelay This option specifies how long a node should wait from receiving a data packet until it should reply with an acknowledgement packet. The default value for this option is 1 second.

ACKTimeout This option specifies the delay before a node sees a connection as timed out, after sending a data packet without receiving an acknowledgement packet. For this option, the default value is 3 seconds.

When testing the protocol we have created a few different versions of the protocol option structure to test with. The *lowRREQTTLQOpts* option set, has default values for all things except it only has a TTL of 4. Whereas *RREQBroadcastAllOpts*, *RREQBroadcastLogOpts* and *RREQBroadcastTwoOpts* each have different values for *RREQBroadcastStrategy*. A set of options called *forwardRREQsWhenMatchingOpts* has the default behavior, except it forwards route request as described by *ForwardRREQsWhenMatching*. Lastly *longACKDelayOpts* has an *ACKDelay* of 2 seconds and a *ACKTimeout* of 15 seconds.

Listing 8.1: The *ProtocolOptions* structure.

```go
type ProtocolOptions struct {
  EnableSync                   bool
  DisableAutoRREQOnConnection bool
  MaxRREQTTL                   int
  RREQBroadcastStrategy        BroadcastStrategy
  ForwardRREQsWhenMatching     bool
  ACKDelay                     time.Duration
  ACKTimeout                   time.Duration
}

type BroadcastStrategy int
const (
  BroadcastAll BroadcastStrategy = iota
  BroadcastLogFunc
  BroadcastTwo
)
```

8.2.4 Scenarios

The evaluation is performed by running simulations of different scenarios with varying protocol options. Each scenario is set up by creating, linking and specifying movement for nodes, and then running the simulator for a given amount of time. In general, for all scenarios the simulator has a maximum distance for node reach of 20 meters. We will now go through the scenarios.

Protest

This scenario attempts to emulate a protest in the streets in the city of Odense. A Vadere simulation has been created for providing the movement of the nodes. The simulation uses the layout of the center of the city, and thus set up obstacles such that nodes can only move along streets and through squares. The nodes are spawned in over time in a specific area, and move towards another area in the city. Once they reach this destination, they wait at this area for a while, after which they continue to the next destination. A handful of stationary nodes are also added to the simulation to represent people who are not part of the protest.

The whole crowd simulation is run with more than 1000 nodes. However, we prune this by 90%, such that 119 nodes are represented when simulating the protocol. This is done since we see it as unlikely that every person at a protest will have the application installed, but we still want to get the crowd dynamics of the whole crowd. A screenshot from a visualization of the simulator running the protest scenario can be seen in fig. 8.1.

From the nodes, 100 random pairs are selected. One of the nodes from each pair will attempt to send a message containing “ping”, to which the other node will reply “pong”. The communication can be seen as successful if the ”pong” message is received. Thus, the percentage of the successful communications is an indicator for how well the protocol performed in this scenario. There is a random delay for each node for when they start attempting to send ”ping” messages. This delay is between 20 and 320 seconds. Nodes will attempt to establish sessions by sending route requests, however if they are not successful, they will wait 60 seconds and reattempt to establish a session by sending a new route request.

This scenario does not use the synchronization extension and simply sends the data packets once a session has been established. The simulation is run for 800 seconds.

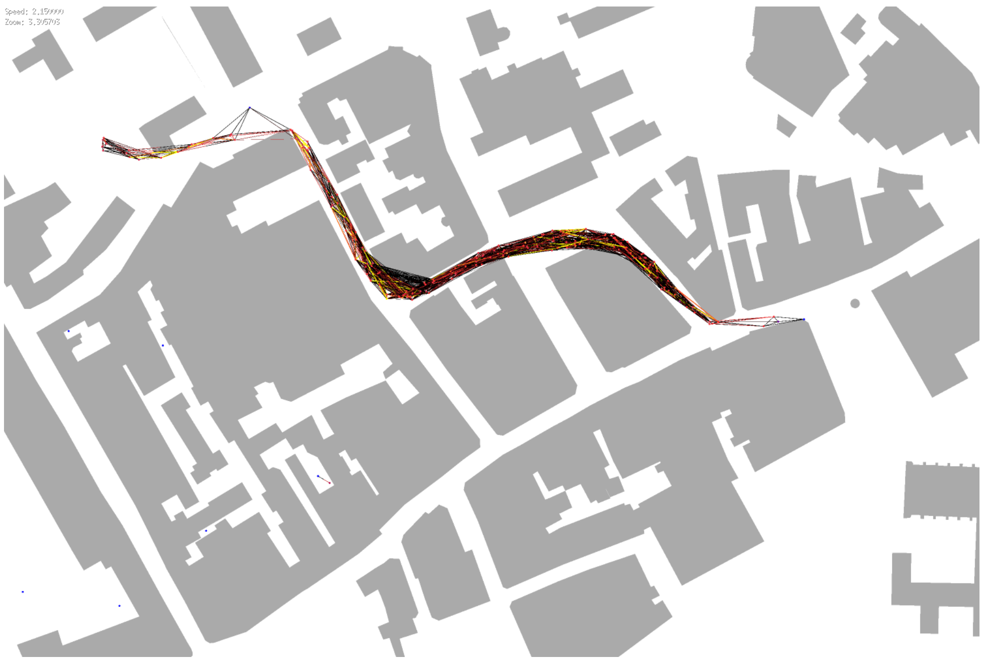


Figure 8.1: A screenshot from a visualization of the protest scenario running in the simulator.

Protest Group

In line with the previous scenario, this one also simulates a protest. It uses the exact same movement of nodes, but includes a different communication pattern for the nodes. The idea behind this scenario is to simulate a large group of people at a protest that have joined a single large group in the protocol. We see this as a realistic scenario where the protest organizers have created a group for coordination, and it has spread throughout the participants of the protest.

This simulation uses the synchronization extension when sending messages. It creates a single large group containing x% of the nodes on the network. Here x will be varied between 10% and 100% for different runs. Each member of the group sends between 1 and 10 messages to the group within the first few minutes of the simulation. The simulation is then run for 800 seconds, and the success is decided by the degree to which nodes have managed to synchronize. This *synchronization degree*, we characterize by the ratio of received messages to the total amount of messages that could have been received (including each nodes own messages). Say that 10 nodes are in a group and each send 5 messages, this means that at the end, the total sum of messages each node has stored would be $10 \cdot 5 = 50$ in the best case. If only 25 messages are received in total, this would give a synchronization degree of 50%.

We also test a variation of this scenario, where instead of having a single group there are instead 10 smaller groups. This could reflect a reality where there are multiple groups within a single protest. The groups in this scenario have between 10 and 20 members, and each node sends between 1 and 10 messages in the groups they are members of.

Random Waypoint Model

This scenario is quite different from the previous. Instead of using crowd simulation, this follows a variation of the more simple random waypoint model. The random waypoint model [17] has been a popular mobility model for evaluating MANETs for many years. While it has been criticized for being too simplistic, the popularity of it means that it can serve as a good way of comparing results with other MANET protocols.

The simulation consists of x nodes moving randomly within a given area. Each node randomly selects a location within the area and moves towards it with a randomly selected speed. Since the simulation contains random movement of the nodes, the simulation is run 10 times. We vary x from 10 to 100 between runs, and regulate the size of the area to keep the node density the same between the runs.

This simulation focuses the efficiency of the protocol for different number of nodes. Before starting the simulation, x random pairs are selected, and these perform a ping-pong similarly to the first protest scenario. The efficiency here is also given by the ratio of successful ping-pong communications. It is important to note that the amount of messages to be sent scales with the number of nodes. Each run of this simulation is run for 600 seconds.

8.3 Simulation Results

While the simulations described in section 8.2 are running, statistics are collected regarding multiple aspects of the protocol, such as types of packets, session information, and much more. These results will be presented and discussed in this section.

8.3.1 Random Waypoint Model

A multiline plot depicting the results can be seen in fig. 8.2. Here, each line represents data for runs with different protocol options. The x-axis represents the number of nodes (and indirectly the size of the area), while the y-axis represents the ratio of successful ping-pong communications. The plot illustrates that generally, the number of successful communications decrease as the number of nodes increases. At 10 nodes, we see that most runs have a 100% success rate. This quickly decrease at 20 and 30 nodes, after which it stabilizes around 10%-30% for the most part. The main outlier here is *RREQBroadcastTwo*. This strategy of only forwarding route request to two other nodes, also sees a decrease in success as the number of nodes increases, however it generally stays within 60%-70%.

Initially the very positive result for *RREQBroadcastTwo* might seem counter-intuitive. Since nodes forward fewer route requests, one can imagine that the routes that are found will generally be less direct, and thus that routes will be less stable and more prone to disconnections. The fact that sending fewer route requests leads to better results, indicates that the network is too flooded with route requests to function effectively. This aligns with the fact that *RREQBroadcastLog* performs better than *RREQBroadcastAll*, since it also sends fewer route request. Also, *longACKDelay* does quite well, which we attribute to it sending fewer route request because sessions do not time out as quickly.

This is further corroborated by considering table 8.1. This table shows the percentages of the different network layer packet types, for runs with different protocol options. Here, we look at the runs with 100 nodes. It is clear that a very large part of the packets sent on the network are route requests. For almost all the runs, it is above 96%, while only *RREQBroadcastTwo* reaches 93%. This indicates that one of the limiting factors of the protocol is that it generally is too liberal with its use of route requests. The rate of meta packets to actual data packets is very high.

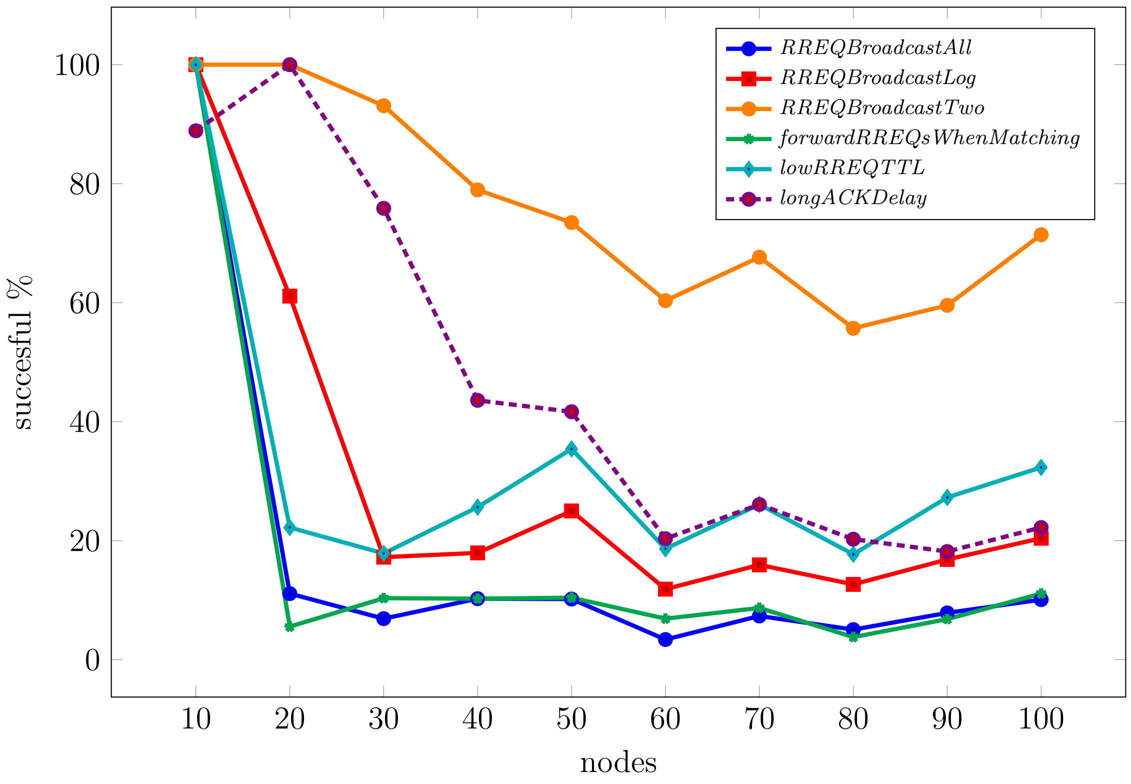


Figure 8.2: Results for the random waypoint model simulations. The x-axis represents number of nodes in the simulation and the y-axis represents the percentage of the communications which are deemed successful. Each line represents different protocol options.

While these results are interesting, they are generated using the random waypoint model, and thus they are not necessarily representative of the use of the protocol in a protest. The random waypoint model is quite generous in the fact that nodes are moving randomly and independently of each other, which leads to many of the nodes, at some point in time, coming within relatively close range of each other.

8.3.2 Protest

We will now present and discuss the results of running the protest scenario. The main results for this scenario can be seen in table 8.2.

We clearly see a substantial decrease in success rate for communications in this scenario compared to the scenario using the random waypoint model. Here, the success rate is below 10% for all protocol options. We again see that *RREQBroadcastTwo* performs best, but it is a lot worse than the previous scenario. However, this is to be expected. The protest scenario takes place on a larger area, and the crowd dynamics of the protest means that nodes have a tendency to stay somewhat close to the neighbors that are nearby as the protest moves. There are of course exceptions, but generally nodes do not independently move throughout the crowd. This means that when selecting random pairs of nodes, there is a decent chance that the two nodes are simply far enough away from each other during the entire simulation, that achieving a session is very unlikely. This is of course a property of the scenario, and essentially limits how well the protocol can operate.

In the table, we see that *longACKDelay* does well when it comes to the percentage of data packets that are successfully acknowledged. This is likely because it is more forgiving

Protocol options	**% RREQ**	**% RREP**	**% SESS**	**% RERR**
RREQBroadcastAll	97.82	0.87	0.30	1.01
RREQBroadcastLog	96.60	1.37	0.49	1.54
RREQBroadcastTwo	92.96	2.55	1.82	2.66
forwardRREQsWhenMatching	97.80	0.87	0.32	1.01
lowRREQTTL	97.34	0.94	0.69	1.03
longACKDelay	98.24	0.79	0.27	0.70

Table 8.1: Percentage of network layer packet types for different protocol options.

Protocol options	**Success rate (%)**	**Acknowledged (%)**	**Drop (%)**
RREQBroadcastAll	6.06	15.62	17.66
RREQBroadcastLog	5.05	14.24	16.62
RREQBroadcastTwo	9.00	40.43	10.15
forwardRREQsWhenMatching	6.06	19.16	17.77
lowRREQTTL	3.03	11.45	17.88
longACKDelay	8.00	54.19	17.63

Table 8.2: Different statistics for running the protest scenario for each of the protocol options. The *success rate* represents the successful ping-pong communications. The *acknowledged* field represents the percentage of received messages that are successfully acknowledged at the other party. *Drop* refers to the percentage of packets that are dropped.

when it comes to sessions. If an acknowledgement packet is delayed, it has a higher chance of arriving before the session times out. We also see that *RREQBroadcastTwo* performs better than the rest in this regard. This is probably again because there are fewer route requests on the network and that the protocol is thus more efficient.

An interesting statistic is the drop rate. While the transmission model only drops 1% of messages randomly, the percentage of dropped packets is above 15% for most runs. Packets are mainly dropped by the transmission model or if the buffer is full. This strongly indicates that a large portion of the network packets are simply dropped due to full buffers. Here it is also notable that *RREQBroadcastTwo* again stands out from the other protocol options, since fewer route requests are filling up buffers.

8.3.3 Protest Group

The scenarios presented so far have only considered point to point communication, and has not considered groups. This scenario has a single large group which some percentage of the protesters are members of. The results of running this scenario for different proportions of membership, can be seen in fig. 8.3. It is clear from the figure that all protocol options have roughly the same behavior as membership increases. This indicates that the performance of group synchronization does not depend much on the different variations of the underlying layers, but more on how the synchronization extension is implemented.

The protocol performs best for lower membership percentages. This is likely because the number of messages to be sent on the network is much lower. For 10% membership, the synchronization degree lies roughly within 70%-80% for most protocol options. This

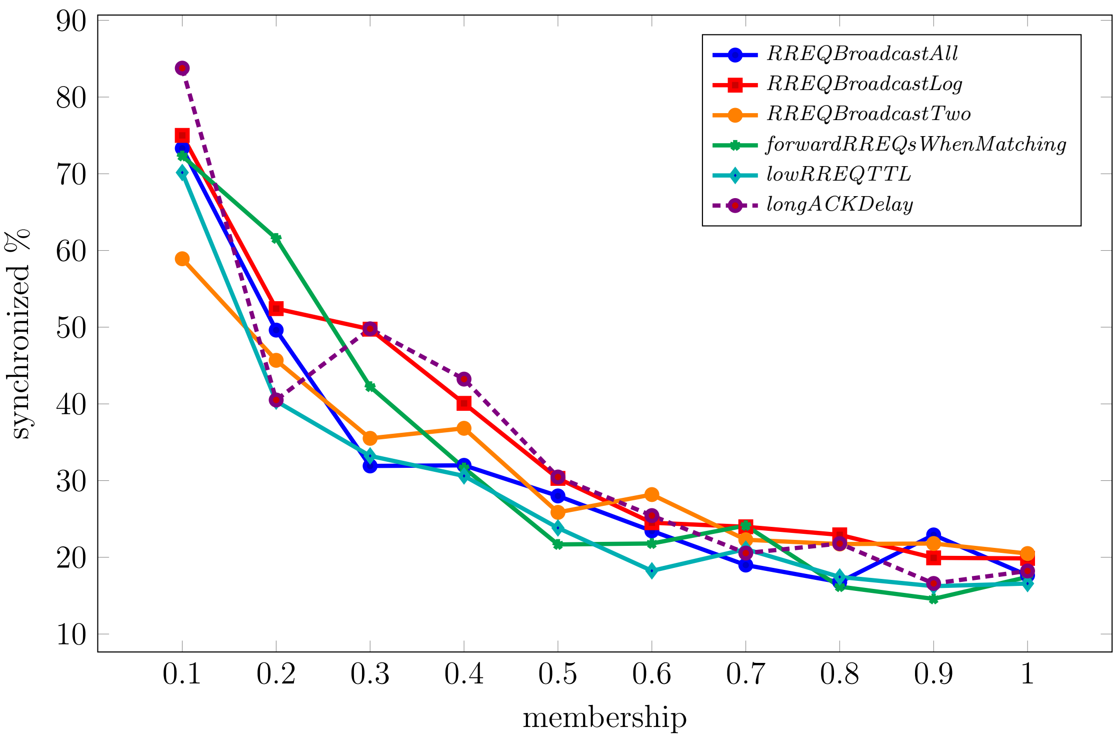


Figure 8.3: Results for the protest group scenario. The x-axis represents the proportion of nodes that are in members of the group. Whereas the y-axis represents the degree to which the nodes have synchronized. Each line represents different protocol options.

decreases to around 20% when every node is a member of the group. However, this is to be expected since the traffic on the network is increased significantly, while the number of nodes on the network stays the same.

One interesting point about this scenario is that packet drops are significantly decreased compared to the previous protest scenario. Where most runs in the previous scenario had drop rates above 15%, the average drop rate over all of the runs of this scenario is 2.45%. This indicates that the network is less flooded by route request and this results in fewer drops due to full buffers. The run with 100% membership that uses the *RREQBroadcastTwo* option, manages to achieve that 49% of the packets on the network are session packets, while only 30% are route requests.

We will now consider the variation of this scenario, where there is not a single big groups but instead 10 smaller groups. This could be seen as a quite realistic scenario where multiple different groups of friend participate in the protest and each have their own group. The results of running this scenario can be seen in table 8.3. Here, the result that really stands out is that *RREQBroadcastTwo* manages to achieve a degree of synchronization of 72.1%. This essentially means that if a node sends a message in a group, it can expect for 72.1% of the members of the group to have received the message within 10-15 minutes.

8.3.4 Discussion

We see that generally the problem of communication within a protest setting is quite difficult. The results we got from the scenario using the random waypoint model were much better than the ones for the first protest scenario. To us this indicates that the

Protocol options	**Synchronization (%)**
RREQBroadcastAll	11.33
RREQBroadcastLog	39.65
RREQBroadcastTwo	72.1
forwardRREQsWhenMatching	13.4
lowRREQTTL	17.5
longACKDelay	52.16

Table 8.3: The results of running the protest multi group scenario with different protocol options.

	Question	**Result**
Q1	Understandable welcome screen	4.4
Q3	Technical problems when linking	1 Yes, 9 No
Q4	Intuitive to link contact	3
Q6	Could send message	9 Yes, 0 No
Q7	Could get response	9 Yes, 0 No
Q8	Message conversation intuitive	4.7
Q10	Able to join group	9 Yes, 0 No
Q11	Easy to follow conversation	4.5
Q12	Create and join groups confusing	3 Yes, 6 No
Q14	Connection lost	9 Yes, 0 No
Q15	Messages delivered on return	4.5

Table 8.4: Aggregated user test results.

movement involved in the scenario is extremely important when evaluating MANETs. The results of point-to-point communication in the protest scenario were generally not very positive. We do attribute part of this to the fact that the setting simply is very difficult to route through, but we also see that the number of route requests being sent on the network in this scenario severely limits how effective the network is.

One category of scenarios where this is much less pronounced is when it comes to groups and synchronization. In these cases, the protocol holds up very well even in a protest setting. The groups allow nodes to gossip messages on behalf of others, and generally means that sessions across long distances are not as necessary. We see this as an accomplishment and an indication that the protocol could be useful in a real life setting. We also see it as an area of opportunity where more refinement could improve the protocol even more.

8.4 User Test

In order to evaluate the smartphone app in practice, we ran a user test with 11 participants. The purpose of the test was to evaluate the usability of the app from the perspective of a first time user. Each participant was given a test sheet starting with a set of tasks that were meant to be done individually without external help. A copy of the test sheet can be found in appendix B. After each task, the user was asked to answer a set of questions used to evaluate how easy and intuitive the task was, and whether they encountered problems in performing it. Some questions simply had yes and no answers, while others used a five-level Likert scale where 5 is positive and 1 is negative.

The first task asked the participants to read through the welcome screen (see relevant screenshot in fig. 7.3d), and evaluate its understandability. The overall response was positive (Q1 in table 8.4). Participants generally thought that the welcome screen had too much text to go through before being able to use the app. One participant mentioned to introduce concepts ongoing throughout using the app, rather than presenting everything at once.

Next, the participants were asked to link with each other and become contacts. We knew beforehand that this process was more complicated that we would have liked, and some participants raised the same concerns that we had (Q3, Q4).

Afterwards, we asked the participants to send messages back and forth between their newly linked contact. Everyone was able to send and receive messages (Q6, Q7) and participants generally found the task very intuitive (Q8). We find this result extremely positive as it confirms that one of the core functionality of the app works well.

We then moved focus to groups, where we asked the participants to create or join a group with each other and test how well group communication worked. All participants were able to join small groups and have a conversation (Q10). Participants mostly raised issues regarding the user experience with the process of creating groups and inviting members. One participant mentioned that synchronization in larger groups seemed to stop working such that new messages would never be delivered.

We later saw from the logs that an error regarding synchronization did occasionally occur. We therefore believe that the problem was caused by a bug in the implementation rather than a design flaw. Unfortunately we did not have enough time to fully investigate the problem.

Lastly, we asked the participants to physically move away from each other such that the connection would be dropped and come back, to test that the connection would automatically be reestablished, and new messages be synchronized. This process generally worked very well for the participants (Q14, Q15).

8.4.1 Coordinated Tasks

After the individual tasks, two coordinated tasks involving everyone was performed. In the first task, all participants were asked to stand in a line with a few meters between each. The two people at each end of the line then attempted to send a message to each other. If successful, the distance between people in the line was increased. The aim of the task was to evaluate how viable multi hop communication is in the app.

At the start with small distances between each participant, communication worked as expected. As the line spread out, the communication continued to work for some time. We ended up with a line consisting of 6 people, where the distance between the two ends was roughly 150 meters. After this however, when participants tried to spread out further, the communication became unreliable. Because coordination of the participants was hard when the line became long together with the complexity of the real world testing environment, it was hard to determine the cause of the unreliability.

Communication was in general stable and reliable, but we sometimes encountered problems with the Bluetooth communication. Sometimes the direct Bluetooth connection between two devices would only work in one direction and sometimes the devices could not communicate at all even though the devices seemed to be connected with each other. We expect this to be the cause of the unreliable results of the test, and we think this could be fixed by improving the code responsible for maintaining the Bluetooth connections.

Lastly, in the last test, all participants joined one large group. The goal of this test was to evaluate the performance of a larger group in practice. Synchronization in the group seemed to work well in the beginning, but when the message history reach a certain size, it seemed to cause synchronization problems similar to those mentioned earlier.

Chapter 9

Final Remarks

We have used the last year to research, design and implement the Starling protocol along with a simulator and a smartphone application. All the work has been collected and published online[1]on GitHub under the permissive open-source Apache-2.0 license. We hope that this work will prove useful for future advancement in the field of ad-hoc smartphone networks. In this chapter we present the potential future work for the project, followed by a conclusion.

9.1 Future Work

9.1.1 Application

This section highlights some potential improvements to the smartphone application.

One of the immediate improvements to the application would be to simplify the linking process. As of now, an external app is needed in order to scan the linking QR codes. To simplify this process, a built-in QR scanner could be added, which would make the process much more intuitive and decrease the risk of user error.

The app still contains a few bugs and minor issues. For instance, the Bluetooth Low Energy (BLE) connections are not always reliable, and from looking at the debug logs it is clear that this likely is caused by bugs in the implementation. Fixing and improving this reliability issue, would generally make the app much more efficient.

We also saw an issue when synchronizing larger groups, where in some cases synchronization would stop working altogether. Late in the process, we found a suspicious error mentioning a problem with the authenticity of the signatures.

Another improvement that we did not get to implement is validation of the version numbers in the delta of a synchronization push packet. Without this, it would be possible for an adversary who is member of a group to corrupt the synchronization state of a recipient in the group, This attack could be performed by sending messages with very high version numbers, essentially making it impossible for the recipient to later receive old messages.

Another practical detail we did not manage to implement in time, is that we ideally should be persistently storing the request IDs from the route requests we reply to, such that we avoid responding to them in the future. This is to avoid an attack where an adversary stores route requests and some time later resends them. If a node were to not

[1]The project can be found at `https://github.com/starling-protocol`

store request IDs and thus reply to the adversary, the adversary can use this to check if the node is currently on the network.

Since the application is designed with security as the main focus, it is vital that the user understands the security implications of their actions performed in the app. As an example, they need to understand that when adding new members to a group they are trusting that those members will not add other new untrusted members to a potentially private group containing sensitive messages. They also need to be informed that while the app is installed and connected, it will help forward messages of other people. This is relevant since an authoritative regime could potentially view this as a criminal offense. Because of this, the application could be improved to further inform the users about the implications of using the different functionalities of the app.

One of the main difficulties in distributing the app to the people who need it, is that we cannot assume they have uncensored internet access or internet access at all. It is thus relatively simple for an authority to stop the distribution of the app through the Internet. A solution to this could be to allow people who already have the app, to distribute it to nearby devices which do not have it. The optimal way of doing this would be to give the user the option of doing it in the app. However, we are unsure if this is actually feasible. Another option would be, e.g. on Android to bundle the APK with the app and provide a guide for the user of how to send it using a feature like Androids “Nearby Share” or over a mobile hot-spot. A general problem with distributing the app in this fashion is that it is very difficult for the user to verify the integrity of the application. An adversary might share a tampered and malicious version of the app without the recipient having any way to verify its validity.

9.1.2 Improvements to Security

Security has been an integral part in the design of the Starling protocol. We will now mention some improvements to the protocol that could increase the level of security.

We believe that the current method used to establish secure sessions has some limitations, such as the lack of forward secrecy and key-compromise impersonation, that could be improved by using another scheme. This is further described in section 4.3.4, where we also propose that the double ratchet algorithm could potentially be used.

A major improvement regarding security would be to create a way for users to share their group identities with their contacts. Although groups are authenticated, the protocol currently provides no way to authenticate the individual members within it. The only way to verify the identity of a group member is to manually inspect their public key. It would be nice if the protocol provided a way to automate this process, such as by sending a signed message to the corresponding linked contact. Group identities would of course still be kept anonymous by default.

One aspect of the contact bitmap that we would like to improve in the future, would be to provide a proof that our implementation for generating a family of hash functions for the contact bitmap, as described in section 5.2.2, is secure. As described, our implementation uses an efficient method where only a single hash is computed per contact. This improves the time for encoding and decoding contacts in contact bitmaps. We believe that it is secure, however we have not supplied a formal proof of this.

Currently, there is no notion of throttling route requests in the protocol. Route request throttling could be a valuable addition, since it would make flooding attacks less efficient. The difficulty is of course to separate the spurious route requests from the genuine ones.

You could assume that nodes sending too many route requests are attempting to flood the network, and thus simply ignore route requests from neighbors sending too many. However, there is of course no guarantee that these route requests actually originate from this node, or if the node is simply located at a choke point in the network where many route requests are passing through them.

An aspect of the security of the application that could be improved in the future relates to the physical security of the user. Here, functionality such as messages that are automatically deleted after a given time, would allow a user to deny communication even if an adversary has physical access to their device, if enough time has passed. Another mitigation could be for the user to uninstall the app or delete sensitive messages, when they see the threat coming. An efficient way of doing this, could be to integrate with panic button apps, such as [21].

Both the Secure Enclave [7] for iOS on Apple and Keystore [4] for Android exposes APIs that allows for key generation using the security hardware of the device. This means that cryptographic keys will be stored securely in hardware, such that the original private keys can never be extracted from the device and all encryption will have to go through the security chips. By storing the shared secrets and private keys in hardware, we would ensure that the identity of a person is strictly tied to the device, and that a message can only be decrypted by the device of the intended receiver.

9.1.3 Improvements to Efficiency

The results of evaluating the performance of the protocol were presented in section 8.3. Through this, we saw that the protocol handled some scenarios very well, while it was less efficient in others. We will now present some of our thoughts regarding how we could improve these inefficiencies.

One optimization that we think would affect the protocol positively when using the synchronization extension, would be to split up synchronization packets. Currently, when pushing deltas, a node will always include all the messages that the receiver is missing. This is potentially problematic for large groups. Here, there could be a very long message history that the receiver has not seen yet, and this results in one very large synchronization packet containing everything. This packet likely has a lower chance of actually arriving at the receiver. A better solution could be for the sender to split this delta into multiple packets. We of course would still need to follow to the principle that deltas must not arrive out of order, however the transport layer would automatically handle this.

One of major problems discovered through the evaluation was that route requests take up a very large part of the network traffic. Currently, nodes forward route requests to their neighbors one at a time. If a node has many neighbors, this of course means that it will spend a considerable amount of time forwarding the same route request to many different nodes. A major optimization to this would be to allow nodes to actually broadcast the route requests. In this case, each route request would maximally be forwarded once by each node, and nodes in high density areas could even choose to only forward some percentage of route requests to further decrease the number of route requests on the network. The big problem with this is that while the BLE protocol technically supports broadcasting, contemporary smartphones usually do not allow unrestricted access to this in the APIs. There might be some ways around this restriction such as abusing the advertisement functionality of the API, however even then the size of BLE advertisement packet are quite limited in size. In the future, devices might have more options regarding broadcasting packets, and the protocol could benefit greatly from this.

Another interesting aspect of forwarding route requests that we have not explored, is to use some heuristics for determining which neighbors to forward them too. It could be interesting to consider the effects of forwarding based on the Received Signal Strength Indicator (RSSI) of a neighbor. Using RSSI one can obtain an approximation of the distance to the neighbor. Here, prioritizing nodes that have a high RSSI and thus are seen as close, might provide more stable routes since they are less likely to lose connection immediately. On the other hand, if you prioritize nodes that are far away you might be able to extend the reach and get more efficient routes. Thus, testing these different strategies could reveal interesting dynamics and thus potentially improve the protocol.

9.2 Conclusion

In this thesis we have designed and implemented the Starling protocol, which specifies how common smartphones can be used to form a Mobile Ad-hoc Network (MANET). In doing so, we have focused this protocol for use in protests and during civil unrest, where access to digital communication has been intentionally restricted by a government or state actor. Furthermore, we have developed a smartphone application that utilizes the protocol, using BLE for the underlying link layer, to demonstrate its use in a practical real world scenario.

The application has been evaluated in a user test of 11 participants showing great potential for the application. Although the application did not work perfectly in all areas, all the shortcomings seem to be related to the practical implementation and could have been resolved with more time and effort.

The protocol has been designed using a layered structure, wherein each layer provides certain properties. Here, the network layer allows user to establish sessions and send secure messages across routes with multiple hops. The transport layer guarantees correct ordering of packets and provides reliable transportation. The synchronization extension facilitates group communication and message history.

In this thesis we have also implemented a simulator specifically designed for simulating and evaluating MANETs. This was used to perform an evaluation of the Starling protocol by simulating its use in large scale protests. In order to do this, the simulator handles realistic movement and attempts to simulate the necessary properties for the link layer, such as packet delays, drops as well as network congestion.

We conclude that the extreme requirements on both the network structure and security properties of the Starling protocol, means that we have to strike a balance between anonymity and efficiency. Our evaluation shows that the protocol generally has some trouble with point-to-point communication in a setting like a large protest. The protocol does however shine when used for group communication, as it allows members of groups to exchange messages transitively.

Glossary

acknowledgement packet A transport layer packet used to acknowledge that a set of data packets has been received. 30, 40, *see* transport layer

AES-GCM Symmetric encryption using AES with GCM as the mode of operation. 22, 23, 33, 73, 74

BLE Bluetooth Low Energy is a variant of Bluetooth made primarily for IoT devices. It is the technology the application uses for communication. 1, 2, 5, 12, 17, 51, 53, 57, 63, 65–67, 87, 90

contact bitmap A contact bitmap in a route request. 19, 24, *see* route request

contact linking The process of linking two devices for them to become contacts and communicate. 21, 22

contact secret a shared secret between two linked users or between members of a group, used to facilitate encryption and authentication. 21, 22, 24, 44, 67, *see* contact linking

data packet A transport layer packet used to carry data. 29, 40, *see* transport layer

Ed25519 Standard for digital signatures using elliptic curves. 33, 38

ephemeral key A public / private key-pair used in session establishment to derive the ephemeral secret. 19, 22

ephemeral secret A shared secret unique to a session, used for providing confidentiality and authentication for the give session. 22

epidemic routing A routing protocol uses epidemic routing if messages are shared among nodes using gossiping. 9

KDF key derivation function. 22

MANET Mobile Ad-hoc Network. 1, 2, 5, 9, 51, 52, 58, 59, 80, 90

network layer The network layer handles routing and session establishment in the network. 18, 19, 37, 73

proactive routing A routing algorithm that maintains a routing table of all routes. 5

reactive routing A routing algorithm that only generates routes when they are needed. 5, 18

request ID The identifier used to identify a route request. 19, *see* route request

route error A packet used to indicate that a route has broken. 7, 19, *see* network layer

route reply A packet sent back as a response to a route request. 6, 19, 73, *see* network layer

route request A packet flooded across the network in order to find a recipient. 6, 19, *see* network layer

routing table A local table keeping track of received route requests. 19, *see* network layer

sequence ID A sequence ID of a transport layer data packet, indicates the index of the packet in the sequence of packets send over the session. 40, *see* data packet

session A short-lived end-to-end connection between two nodes where encrypted data can be transmitted securely. 17, 19, 22, 39

session ID The identifier used to identify a session. 19, 38

session packet A packet used to transmit data securely over a session. 19, 73, *see* network layer

session secret A shared secret between two parties in a session, used to facilitate encryption and authentication of the session. 22, 38

session table A local table keeping track of active and past sessions. 38

transport layer The transport layer provides reliability of a session. 29, 39

TTL The time to live of a route request, indicates the number of hops the packet can be forwarded before being dropped. 19, *see* route request

X25519 ECDH Standard for performing Diffie-Hellman key exchange using elliptic curves. 21

References

[1] Dan Abramov and the Redux contributors. *Redux*. URL: `https://redux.js.org`.

[2] Martin R Albrecht, Raphael Eikenberg, and Kenneth G Paterson. "Breaking Bridgefy, again: Adopting libsignal is not enough". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 269–286.

[3] Martin R Albrecht et al. "Mesh messaging in large-scale protests: Breaking Bridgefy". In: *Cryptographers' Track at the RSA Conference*. Springer. 2021, pp. 375–398.

[4] Android Open Source Project. *Hardware-backed Keystore*. Apr. 29, 2024. URL: `https://source.android.com/docs/security/features/keystore`.

[5] Richard Ankney. "The unified model". In: *contribution to X9F1 Oct* (1991).

[6] Apple Inc. *Core Bluetooth*. URL: `https://developer.apple.com/documentation/corebluetooth` (visited on 04/22/2024).

[7] Apple Inc. *Protecting keys with the Secure Enclave*. URL: `https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave` (visited on 05/29/2024).

[8] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer. 2006, pp. 207–228.

[9] Daniel J Bernstein et al. "High-speed high-security signatures". In: *Journal of cryptographic engineering* 2.2 (2012), pp. 77–89.

[10] *Berty Technogogies*. URL: `https://berty.tech` (visited on 12/18/2023).

[11] Simon Blake-Wilson and Alfred Menezes. "Authenticated Diffe-Hellman key agreement protocols". In: *International Workshop on Selected Areas in Cryptography*. Springer. 1998, pp. 339–361.

[12] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[13] Core Specification Working Group Bluetooth SIG. *Bluetooth Core Specification*. 2023.

[14] Azzedine Boukerche. "A Taxonomy of Routing Protocols for Mobile Ad Hoc Networks". In: *Algorithms And Protocols For Wireless Mobile Ad Hoc Networks*. Wiley Series on Parallel and Distributed Computing. Wiley Ieee Press, 2009. Chap. 5. ISBN: 9780470383582.

[15] Azzedine Boukerche. *Algorithms And Protocols For Wireless Mobile Ad Hoc Networks*. Wiley Series on Parallel and Distributed Computing. Wiley Ieee Press, 2009. ISBN: 9780470383582.

[16] *Bridgefy Messaging App.* URL: https://bridgefy.me (visited on 12/18/2023).

[17] Josh Broch et al. “A performance comparison of multi-hop wireless ad hoc network routing protocols”. In: *MobiCom ’98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking.* ISBN: 978-1-58113-035-5.

[18] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[19] Facebook, Inc. *Messenger Secret Conversations: Technical Whitepaper.* May 18, 2017. URL: https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[20] Google. *The Go programming language.* URL: https://go.dev/.

[21] Guardian Project. *Ripple: respond when panicking.* URL: https://guardianproject.info/apps/info.guardianproject.ripple.

[22] Hajime Hoshi. *Ebitengine.* URL: https://ebitengine.org/.

[23] Kirsten Hildrum et al. “Distributed object location in a dynamic network”. In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures.* 2002, pp. 41–52.

[24] Niels de Hoog and Elena Morresi. “Mapping Iran’s unrest: how Mahsa Amini’s death led to nationwide protests”. In: *The Guardian* (Oct. 31, 2022). URL: https://www.theguardian.com/world/ng-interactive/2022/oct/31/mapping-irans-unrest-how-mahsa-aminis-death-led-to-nationwide-protests (visited on 11/08/2023).

[25] *Information technology – Open Systems Interconnection (OSI) – Basic Reference Model: The Basic Model.* Standard ISO/IEC 7498-1:1994. International Organization for Standardization, Nov. 1994.

[26] David B Johnson, David A Maltz, Josh Broch, et al. “DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks”. In: *Ad hoc networking* 5.1 (2001), pp. 139–172.

[27] David B. Johnson and David A. Maltz. “Dynamic Source Routing in Ad Hoc Wireless Networks”. In: *Mobile Computing.* Ed. by Tomasz Imielinski and Henry F. Korth. Boston, MA: Springer US, 1996, pp. 153–181. ISBN: 978-0-585-29603-6. DOI: 10.1007/978-0-585-29603-6_5. URL: https://doi.org/10.1007/978-0-585-29603-6_5.

[28] Muhammad Kashif Nazir, Rameez U. Rehman, and Atif Nazir. “A Novel Review on Security and Routing Protocols in MANET”. In: *Communications and Network, 08 (04).* Springer. 2016, pp. 205–218.

[29] Benedikt Kleinmeier et al. “Vadere: An Open-Source Simulation Framework to Promote Interdisciplinary Understanding”. In: *Collective Dynamics* 4 (Jan. 1, 2019). DOI: 10.17815/CD.2019.21. published.

[30] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication.* RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://www.rfc-editor.org/info/rfc2104.

[31] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme.* Cryptology ePrint Archive, Paper 2010/264. https://eprint.iacr.org/2010/264. 2010. URL: https://eprint.iacr.org/2010/264.

[32] Ajay D. Kshemkalyani and Mukesh Singhal. "Causal Order (CO)". In: *Distributed Computing: Principles, Algorithms, and Systems.* 1st ed. USA: Cambridge University Press, 2008. Chap. 6.5, pp. 206–215. ISBN: 0521876346.

[33] G Vijaya Kumar, Y Vasudeva Reddyr, and M Nagendra. "Current research work on routing protocols for MANET: a literature survey". In: *international Journal on computer Science and Engineering* 2.3 (2010), pp. 706–713.

[34] David McGrew and John Viega. "The Galois/counter mode of operation (GCM)". In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.

[35] Meta Platforms, Inc. *React Native: Learn once, write anywhere.* URL: https://reactnative.dev.

[36] Nordic Semiconductor. *Android-BLE-Library.* URL: https://github.com/NordicSemiconductor/Android-BLE-Library. (Visited on 04/22/2024).

[37] Laure Nouraout. "FireChat app drives offline communication during Umbrella Revolution". In: *International Journalists' Network* (Oct. 30, 2018). URL: https://ijnet.org/en/story/firechat-app-drives-offline-communication-during-umbrella-revolution (visited on 12/18/2023).

[38] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org.* https://www.openstreetmap.org. 2017.

[39] Charles Perkins and Elizabeth Belding. "Ad-hoc On-Demand Distance Vector Routing". In: vol. 25. Jan. 1999, pp. 90–100. DOI: 10.1109/MCSA.1999.749281.

[40] Charles E. Perkins and Pravin Bhagwat. "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers". In: *Proceedings of the Conference on Communications Architectures, Protocols and Applications.* SIGCOMM '94. London, United Kingdom: Association for Computing Machinery, 1994, pp. 234–244. ISBN: 0897916824. DOI: 10.1145/190314.190336. URL: https://doi.org/10.1145/190314.190336.

[41] Trevor Perrin and Moxie Marlinspike. "The double ratchet algorithm". In: *GitHub wiki* 112 (2016).

[42] Andrea Peterson. "Ukraine's 1984 moment: Government using cellphones to track protesters". In: *The Washington Post* (Jan. 21, 2014). URL: https://www.washingtonpost.com/news/the-switch/wp/2014/01/21/ukraines-1984-moment-government-using-cellphones-to-track-protesters/ (visited on 11/08/2022).

[43] J. Postel. *Transmission Control Protocol.* RFC 793. Internet Engineering Task Force, Sept. 1981, p. 85. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[44] Michel Raynal, André Schiper, and Sam Toueg. "The causal ordering abstraction and a simple way to implement it". In: *Information processing letters* 39.6 (1991), pp. 343–350.

[45] George Riley and Thomas Henderson. "The ns-3 Network Simulator". In: Jan. 2010, pp. 15–34. ISBN: 978-3-642-12330-6. DOI: 10.1007/978-3-642-12331-3_2.

[46] *Secure messaging, anywhere - Briar.* URL: https://briarproject.org (visited on 12/18/2023).

[47] Karthikeyan Sundaresan et al. "ATP: a reliable transport protocol for ad-hoc networks". In: *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*. 2003, pp. 64–75.

[48] The Go authors. *Go Mobile*. URL: `https://go.dev/wiki/Mobile`.

[49] Jacopo Tosi et al. "Performance Evaluation of Bluetooth Low Energy: A Systematic Review". In: *Sensors* 17.12 (2017). ISSN: 1424-8220. DOI: `10.3390/s17122898`. URL: `https://www.mdpi.com/1424-8220/17/12/2898`.

[50] Amin Vahdat, David Becker, et al. *Epidemic routing for partially connected ad hoc networks*. 2000.

[51] Robbert Van Renesse et al. "Efficient reconciliation and flow control for anti-entropy protocols". In: *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. 2008, pp. 1–7.

[52] Jane Wakefield. "Hong Kong protesters using Bluetooth Bridgefy app". In: *BBC* (Sept. 3, 2021). URL: `https://www.bbc.com/news/technology-49565587` (visited on 11/08/2023).

[53] *WhatsApp Encryption Overview: Technical Whitepaper*. Sept. 27, 2023. URL: `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`.

[54] J. Yoon, M. Liu, and B. Noble. "Random waypoint considered harmful". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Vol. 2. 2003, 1312–1321 vol.2. DOI: `10.1109/INFCOM.2003.1208967`.

[55] David G. Young. *Hacking The Overflow Area*. May 7, 2020. URL: `http://www.davidgyoungtech.com/2020/05/07/hacking-the-overflow-area`.

[56] Wei Zhang and Jingsha He. "Modeling End-to-End Delay Using Pareto Distribution". In: *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. 2007, pp. 21–21. DOI: `10.1109/ICIMP.2007.26`.

Appendix A

Specification

The Starling protocol defines an open standard suitable for anonymous ad-hoc routing in an unstructured peer-to-peer network. It is optimized for use in low-bandwidth environments, and it has been specifically designed for use with smartphones using Bluetooth Low Energy for the link layer.

A.1 Conformance

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

A.2 Definitions

- **User** A user on the network.
- **Device** The device running the protocol from the point of view of the user.
- **Peer** A peer is a point-to-point connection to the device.
- **Node** A host running the protocol on the network.
- **Endpoint Node** A node at either end of a (potentially multi-hop) communication.
- **Intermediate Node** A node that is helping facilitate the communication between two other endpoint nodes.
- **Link** Users can link their devices and become linked in order to communicate.
- **Link secret** A secret shared between two contacts used to encrypt their communication.
- **Group** A group is used to facilitate multicast communication between all members.
- **Group secret** A secret used for encryption in a group. Any user holding the group secret is a member of the corresponding group.
- **Session secret** A secret that is used for encrypting a single session.
- **Contact** a group or a link.
- **Contact secret** The corresponding link or group secret.

A.3 Overview

This specification details a protocol stack for communication in Mobile Ad-hoc Networks (MANETs).

The protocol enables users to link, as a way to communicate with other users. When two users want to link up with each other they establish a common contact secret, as detailed in Linking. After linking, the two users are able to send data to each other. The process of sending data contains multiple steps, which will be demonstrated by the following example.

User `A` wants to send a message to user `B`. Initially `A` needs to find a route to `B` such that the message can be delivered. This is handled by the network layer as described in Network Layer. What essentially happens is that `A` sends a packet containing a route request to its neighbors, which then continue spreading the request on the network. If `B` is within the network, it will receive the route request and reply to it with a route reply. Once `A` receives the reply, a route has been found and a session can be established which includes generating a session secret. Now the session has been established, `A` can send the data it wanted to send using a session packet. Before `A` sends the data it will encrypt it using the session secret to ensure confidentiality.

While the network layer provides the basis for routing messages, reliability and message ordering is provided by the Transport Layer. It attaches a sequence number to the messages in the data packets it sends, using the session packets from the network layer. Thus, when `A` sends a message to `B`, `B` can identify if it is missing some earlier messages. `B` also replies with an acknowledgement packet once it has received a message, which enables `A` to confirm the delivery. If no acknowledgement is received, `A` will assume that `B` did not receive the message, and resend it.

A.4 Binary Encoding

Packets are encoded following the structure described in the associated table. Bytes are concatenated in the order they appear in the table from top to bottom. Unsigned integers (uint) are encoded using big endian.

A.5 Cryptographic Primitives

The protocol uses the following cryptographic primitives.

- `AES-GCM`: Is used for all symmetric encryption. Specified in RFC 5288.
- `X25519 ECDH`: Is used for key agreement when linking. Specified in RFC 7748, section 6.1.

- `Ed25519`: Is used for public key authentication. Specified in RFC 8032.
- `HKDF`: Is used for key derivation. Specified in RFC 5869.
- `HMAC`: Is used to find indices when encoding contact bitmaps. Specified in RFC 2104

A.6 Architecture

The protocol is split up into a number of layers, each building on top of the previous, similar to the ISO model.

A.6.1 Link Layer

Assumptions for the link layer:

- Bidirectional communication
- Can be unreliable
- Point to point (multicast is not required)
- Does not need to have: authentication, integrity, confidentiality.

The link layer can potentially be implemented on top of any communication technology, as long as it facilitates bidirectional, point to point communication.

Bluetooth Low Energy

The implementation MAY choose to use Bluetooth Low Energy to facilitate as the link layer.

Bluetooth Low Energy introduces two kinds of actors, a central and a peripheral. The peripheral advertises itself and the central can scan and connect to it. When a connection has been established they can communicate with each other through the use of notifications.

When the device is connected to the network, it MUST run both a central and a peripheral, in order to both connect to other devices and to let other devices connect to it.

The peripheral MUST advertise a service with a characteristic with the following settings:

Service UUID: `e3c9f0b1-a11c-eb0b-9a03-d67aa080abbc`

Characteristic UUID: `68012794-b043-476c-a002-650191132eef`

Characteristic properties:

- Permit notifications
- Write without response
- Readable and writable permissions

The central SHOULD scan for other peripherals with the same service and characteristic UUIDs and connect to it when a match is found.

When the central connects to a peripheral it MUST request to enable notifications for the connection. It is also RECOMMENDED that a higher MTU is negotiated.

When notifications has been enabled, the central can send data to the peripheral by writing to its characteristic.

The peripheral can send data to the central by updating the value of its central only for that central. Since notifications has been enabled, this will let the central know of the data immediately.

A.6.2 Packet Layer

When using the link layer, packets must fit in blocks of some maximum size (MTU) defined by the link layer. To handle this, the packet layer splits up the data to be transmitted into multiple packet layer packets.

Table A.1: Format of a packet layer packet.

Name	Type	Description
Non-empty	1 bit	Indicates whether this packet contains data
Continuation	1 bit	Indicates whether this packet is a continuation packet
Length	14 bit	The length (in bytes) of the `Data` field
Data	`Length` bytes	The data to be transmitted

The `Non-empty` field is 1 if the packet contains data and 0 otherwise. The `Continuation` field indicates whether the data contained in this packet is the end of the data, or if the data continues in another packet (1 indicates continuation, and 0 indicates end of data). The `Length` field specifies the length of the data contained in this packet, in bytes. The `Data` field is the actual data to be transmitted.

A.6.3 Network Layer

The network layer is responsible for routing and providing nodes the ability to send packets to each other in a confidential and authenticated manner.

Request and Session Tables

Table A.2: An entry in the Request table.

Name	Type	Description
Request ID	8 byte	The ID of the route request
Source	MAC	The MAC address of the source neighbor

Table A.3: An entry in the Session table.

Name	Type	Description
Session ID	8 byte	The ID of the session
Request ID	8 byte	The ID of the initial route request
Source	MAC	The MAC address of the source neighbor
Target	MAC	The MAC address of the target neighbor

The request and session tables are used to maintain the local state required to properly route packets. For the session table, intermediate nodes will have both the `Source` and `Target` field, whereas the nodes at each end of the routes will leave one of these fields empty.

`RREQ`: Route Request

Implementations MUST handle this packet.

Table A.4: Format of a route request packet.

Name	Type	Description
Packet type	1 byte	The value `0x01`
Request ID	8 byte	A random ID used to identify the request
TTL	16 bit uint	The max number of hops before the packet is dropped
Ephemeral key	32 bytes	ECDH Public key used for session key derivation
Contact bitmap	256 bytes	A bitmap encoding the targeted contacts

A node constructs and sends a `RREQ` when it wants to communicate with one or multiple contacts. The node should store the newly generated `Ephemeral key` along with the `Request ID`. The contact(s) are encoded in the `Contact bitmap` as seen in creating a contact bitmap.

When a `RREQ` is received it MUST be discarded if the `Request ID` already is contained in the request table. If the `Request ID` is not in the request table, its `Source` and `Request ID` SHOULD be added to the `Request table`. Following this, the contact bitmap SHOULD be matched against each contact of the device (see matching contact bitmaps). If no contact matches, the `RREQ` SHOULD be forwarded by decrementing its `TTL` field by 1, and if the `TTL` is non-zero sending the modified packet to its neighbors. If instead one or more contacts match, an entry SHOULD be created in the session table for each contact, wherein the `Target` field is left empty (this indicates that the node is an endpoint). An appropriate `RREP` SHOULD also be sent for each matched contact (see `RREP`), to the node the `RREQ` was received from.

The `TTL` SHOULD be capped to some maximum value, in order to limit the reach of route requests.

`RREP`: Route Reply

Implementations MUST handle this packet

Table A.5: Format of a route reply packet.

Name	Type	Description
Packet type	1 byte	The value `0x02`
Request ID	8 byte	The ID used to identify the route request
Session ID	8 byte	A random ID used to identify the new session
Ephemeral key	32 bytes	ECDH Public key used for session key derivation
Nonce	12 bytes	Random nonce used as IV for AES-GCM
Payload size	32 bit uint	The size in bytes of piggybacked data, 0 specifies no data
Payload	`Payload size` bytes	Encrypted application data to be piggybacked
Tag	16 bytes	Authentication tag produced by AES-GCM

A `RREP` SHOULD be issued when the contact bitmap of a `RREQ` finds a match. The `Request ID` of the `RREQ` MUST be copied to the `RREP` and a new `Session ID` is generated randomly. An `Ephemeral key` is generated and attached to the route reply which is used when computing the session secret.

The `RREP` MAY include a `Payload` with Application layer data if relevant. If no `Payload` data is contained, the `Payload size` must be 0. The `RREP` MUST use `AES-GCM` to encrypt the `Payload` (if present) and to sign the packet. Everything up to the `Nonce`, is used as the `info` argument when sealing.

If a `RREP` with a `Request ID` not in the request table is received, the packet MUST be ignored. Otherwise, when a `RREP` packet is received, if the `Source` field in the session table is not nil, the node is seen as an intermediate node in relation to the session, else the node is seen as an endpoint. If the node is an intermediate node, a new entry is added to the session table copying the `Session ID`, `Request ID` and `Target` address from the `RREP`, and setting the `Source` field to the value of the `Source` field from the corresponding route table entry identified by the `Request ID`. The node then forwards the `RREP` to the `Source` node. If the node is an endpoint it similarly creates an entry in the session table except it leaves the `Source` field empty, and can now consider the session established.

The `Contact ID` matching the route reply should be stored in the session table. The `session secret` should be computed and also stored in the session table.

`SESS`: Session Packet

Implementations MUST handle this packet. See session communication for more.

Table A.6: Format of a session packet.

Name	Type	Description
Packet type	1 byte	The value `0x03`
Session ID	8 byte	The ID of the session
Nonce	12 bytes	Random nonce used as IV for AES-GCM
Size	32 bit uint	The size of the succeeding data block
Data	`Size` bytes	Data encrypted by AES-GCM
Tag	16 bytes	Authentication tag produced by AES-GCM

A `SESS` packet is sent when a node wants to send data over a session. The `SESS` packet MUST use `AES-GCM` to encrypt the `Data` field and to sign the packet using the `Session secret`. Everything up to the `Nonce`, is used as the `info` argument when sealing.

If a `SESS` packet is received and the sender is neither the `Source` or the `Target`, the packet MUST be ignored. If the `Session ID` of the `SESS` packet is not contained in the session table, the packet MUST be ignored. Otherwise, if the receiving node is an intermediate node the packet SHOULD be forwarded to the `Target` node in the session table entry for this session. In the case that the receiving node is an endpoint, is should decrypt the `Data` field using the `Session secret` associated with the `Session ID`, and deliver the resulting data to an upper layer.

`RERR`: Route Error

Implementations SHOULD handle this packet.

Table A.7: Format of a route error packet.

Name	Type	Description
Packet type	1 byte	The value `0x04`
Session ID	8 byte	The ID of the session

When a valid `RERR` is received containing a `Session ID` known to this device from the correct neighbor, the session SHOULD be removed from the local session table such that future packets on this session will be ignored. If the receiving node is an intermediate node, a `RERR` SHOULD also be forwarded to the relevant `Source` or `Target` node from the session table entry.

A new `RREQ` MAY be broadcast in an attempt to establish a new session.

Session Secret

The session secret combines the `contact secret` between two linked devices with the `ephemeral secret`.

The ephemeral secret is computed as the ECDH shared secret of the ephemeral keys exchanged in the route request and route reply.

The `session secret` can now be computed to be the first 32 byte key material of $HKDF(contact\ secret\ ||\ ephemeral\ secret,\ nil,\ nil)$. Where $||$ is the concatenation operator.

A.6.4 Transport Layer

The transport layer is responsible for sending data reliably over the sessions maintained by the network layer. It ensures FIFO ordering of packages and adds reliability of packet delivery within a session.

`DATA`: Data Packet

Implementations MUST handle this packet. See session communication for more.

Table A.8: Format of a data packet.

Name	Type	Description
Packet type	1 byte	The value `0x01`

Name	Type	Description
Sequence number	4 bytes	The sequence number of the packet
Data	Remaining bytes	Data from the application layer

If a `DATA` packet is received with a `Sequence number` that it has seen before, the packet SHOULD be ignored. If a `DATA` packet is received with a `Sequence number` greater than one more than the last delivered `DATA` packet, it should be buffered until all earlier packets have been delivered first. If the `Sequence number` is exactly one greater than the last delivered packet, this packet can be delivered immediately. Delivering a packet can cause other packets that are currently buffered to also be delivered, as long as their `Sequence number` is exactly one larger than the last one delivered.

When sending a `DATA` packet, the `Sequence number` MUST take the value one larger than the previously sent `DATA` packet by that device. The first `DATA` packet to be sent on a session by a device MUST have a `Sequence number` of 1.

`ACK`: Acknowledge Packet

Implementations MUST handle this packet. See session communication for more.

Table A.9: Format of an acknowledgement packet.

Name	Type	Description
Packet type	1 byte	The value `0x02`
Latest seq. num.	4 bytes	The `Sequence number` of the packets
Count	32 bit uint	The number of missing packets
Missing seq. nums.	Count * 4 bytes	A list of missing seq. nums.

When receiving an `ACK` packet, the node SHOULD attempt to resend the packets with the associated sequence numbers listed in `Missing seq. nums.`. All the packets up to `Latest seq. num.` not listed in `Missing seq. nums.` can be reported as delivered.

An `ACK` packet MUST be sent back after at most 1 second from receiving a `DATA` packet. However, a single `ACK` packet can be used to acknowledge many `DATA` packet potentially received within that second. The `Latest seq. num.` represents the highest sequence number this node has seen. The list of `Missing seq. nums.` contains all the sequence numbers less than `Latest seq. num.` that this node has not seen yet.

Session Timeouts

If an `ACK` packet has not been received after `t_SESS` seconds from sending a `DATA` packet, the underlying session SHOULD be considered to be timed out. The variable `t_SESS` is left as a choice of the implementation but MUST be at least one second. It is RECOMMENDED that is at least the expected round trip time + one second.

A.6.5 Application Layer

The application layer consists of all the encrypted data sent across a session using transport layer DATA packets.

The first byte is reserved as an indication of how to interpret the rest of the packet.

Byte	Description
0x01	Application specific packet
0x02	Synchronization extension

An application specific packet `0x01` indicates that the format of the packet lies outside the scope of this specification. And it is thus up to the implementation to identify and handle the packet properly.

Packet types of `0x02` are handled by the synchronization extension.

Values above 0x02 are open to be used for further extensions of the protocol.

A.7 Linking

The purpose when linking devices is to establish a common contact secret between the two users. This is done in an out-of-band session using X25519 ECDH.

1. Alice generates an ephemeral ECDH key-pair a_p / a_s. She shares her public key (a_p) with Bob over the out-of-band session.
2. Bob generates his own ephemeral ECDH key-pair b_p / b_s. He then shares his public key (b_p) with Alice over the out-of-band session.
3. Both Alice and Bob compute the shared secret using ECDH.

The implementation MUST ensure the following properties of the out-of-band session hold.

- Authentication
- Integrity

A.7.1 Using QR codes for Linking

The implementation MAY choose to use QR codes to share the public keys in the linking process. By REQUIRING the users to physically scan a QR code on the others' device, authentication and integrity is obtained.

1. Alice starts a new short-lived session by generating and encoding her public key in a QR code and shows it on her device.
2. Bob starts his own short-lived session by scanning her public key with his device.
3. Bob then generates his key-pair and performs ECDH to get the shared contact secret.
4. Bob now encodes his generated public key in a QR code and shows it on his device.
5. Alice scans his QR code with her device and performs ECDH to also obtain the shared contact secret. She then terminates her session and forgets her original key-pair.
6. When Alice has scanned the QR code from Bob's device. Bob SHOULD perform an action to terminate his session. Eg. by pressing a "Done" button on his device.

Since it is possible for Bob to obtain the link secret before Alice has the chance to do the same, it is RECOMMENDED to make Bob aware of this until he has properly received and verified a `SESS` packet from Alice.

A.7.2 Group Linking

Users can create new groups like such. The device creates a new random 256 bit group secret s. The secret s can then be shared securely with pre-linked contacts. Now s can be encoded in `RREQs` and multiple devices can reply targeting s.

A.8 Contact Bitmaps

The contact bitmap field in a route request is used to encode multiple contacts that are the intended receivers into a single route request.

The contact bitmap uses the `Request ID` from the `RREQ` packet as the seed.

A contact bitmap is 256 bytes (2048 bits) long.

A.8.1 Creating a Contact Bitmap (Encoding)

A contact bitmap MAY be constructed in the following way.

```
seed = rreq.requestID
bitmap = 256 empty bytes

for c in contacts: # Outer loop
  hash = SHA256_HMAC(seed, c.secret)

  contact_bits = []
  for j in range(12):
    index = ((hash[j*2] & 0x07) << 8) | hash[j*2+1]
    while index in contact_bits:
      index = index + 1
    contact_bits.append(index)

  for j in range(12):
    # convert 2 bytes of hash to bitmap index
    index = contact_bits[j]

    # check for conflicts
    if index already set in bitmap and the value is different from (j % 2):
      continue Outer loop

  for j in range(12):
    index = contact_bits[j]

    # set bit at index in bitmap to j % 2
    set_bit(bitmap, index, j % 2)

set remaining bits in bitmap to be cryptographically random
```

A.8.2 Matching a Contact Against a Bitmap (Decoding)

A list of matched contacts MAY be computed in the following way.

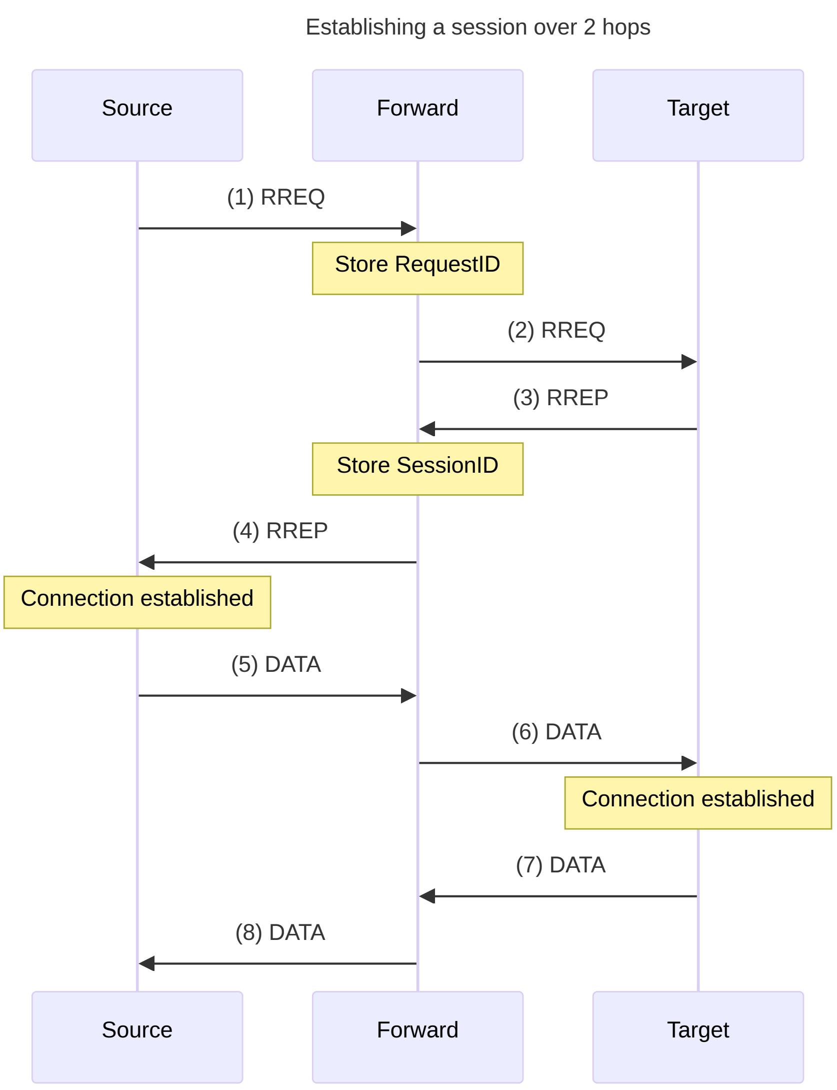
```
seed = rreq.requestID
bitmap = rreq.bitmap

matched_contacts = []
```

```
for c in contacts:
  hash = SHA256_HMAC(seed, c.secret)

  contact_bits = []
  for j in range(12):
    index = ((hash[j*2] & 0x07) << 8) | hash[j*2+1]
    while index in contact_bits:
      index = index + 1
    contact_bits.append(index)

  contact_match = True
  for j in range(12):
    # convert 2 bytes of hash to bitmap index
    index = contact_bits[j]
    value = j % 2

    if bitmap[index] != value:
      contact_match = False
      break

  if contact_match:
    matched_contacts.append(c)
```

A.9 Sessions

A.9.1 Session Establishment

Before two contacts can communicate, a session must first be established.



1. The forward node updates its routing table by storing the `RequestID` along with the `DeviceID` of the node it received the `RREQ` from.

2. `RREQ` has reached the target. The target will check the contact bitmap to see if they should reply to the request. If the bitmap matches, they will generate a new `SessionID` and respond with a `RREP`.

3. `RREP` is received by a forwarding node. It updates the entry in the routing table with the new `SessionID` and the node it received it from.

4. The source gets the `RREP`, and can verify that it was created by the target. It now sees the connection as established, and can start sending `SESS` to the target with the correct `SessionID`.

5. Once the `SESS` packets arrive with the `SessionID`, the intermediary nodes can be sure that the `RREP` they saw before was valid (and not from a black hole attack). It forwards the `SESS` according to its routing table.

6. `SESS` packet reaches the target, and thus it also sees the connection as established.

A.9.2 Session Communication

After a session has been established, the two contacts can begin to send data back and forth. This is done using `DATA` packets. A message is stored in the `Data` attribute of the `DATA` packet along with the next `Sequence number`.

Each `DATA` packet sent over a session includes an incrementing `Sequence number`. The first packet has a `Sequence number` of 0, and it is incremented by 1 for each new packet.

Upon receiving a `DATA` packet, the receiver MUST respond with an `ACK` packet within 1 second. If multiple `DATA` packets are received within this time span, a combined `ACK` can be sent. The `ACK` packet contains the latest `Sequence number` the receiver has seen, as well as a list of `Sequence numbers` smaller than the latest that the receiver has not seen.

A.10 Extension: Message Synchronization

Implementations MAY handle this extension as a way to synchronize messages between contacts.

A.10.1 `SYNC-PULL`: Synchronization Pull Packet

Table A.11: Encoding of a `SYNC-PULL` packet.

Name	Type	Description
Packet type	1 byte	The value `0x01`
Sender public key	32 bytes	The public key of the sender (us)
Sender version	4 byte	The latest version number of the sender
Digest count	32 bit uint	The number of encoded digests
Digests	`Digest count` * 36 bytes	List of digests encoded as described below
Sender signature	64 bytes	Packet signature signed by the sender

Table A.12: Encoding of a digest within a `SYNC-PULL` packet.

Name	Type	Description
Public Key	32 bytes	The public key of the node
Version	32 bit uint	The latest version number of the node

A.10.2 `SYNC-PUSH`: Synchronization Push Packet

Table A.13: Encoding of a `SYNC-PUSH` packet.

Name	Type	Description
Packet type	1 byte	The value `0x02`
Sender public key	32 bytes	The public key of the sender (us)
Receiver public key	32 bytes	The public key of the receiver (them)
Delta count	32 bit uint	The number of encoded deltas
Deltas	`Delta count` encoded deltas	List of deltas encoded as described below
Sender signature	64 bytes	Packet signature signed by the sender

Table A.14: Encoding of a delta within a `SYNC-PUSH` packet.

Name	Type	Description
Public key	32 bytes	The public key of the node
Version	32 bit uint	The latest version number of the node
Content length	32 bit uint	The length of the included content
Content	Content length bytes	The contents of the message
Has group invitation	1 byte	0x00 not attached, 0x01 attached
Group invitation	32 bytes (or 0 bytes)	The shared secret for the group
Signature	64 bytes	Message signature

A.10.3 Maintaining Local State

The synchronization layer maintains a local state of messages such that when two contacts synchronize they will converge towards a common state.

For each contact, a set of messages is maintained where each message holds:

- **Public key** a unique public key used to identify and sign all messages sent by the same device. Each device generates its own Ed25519 public/private key pair.
- **Version** the version number associated with the message
- **Content** arbitrary binary data representing the contents message.
- **Group attachment** an optional shared secret for another group
- **Signature** the signature of the message when encoded in a Delta.

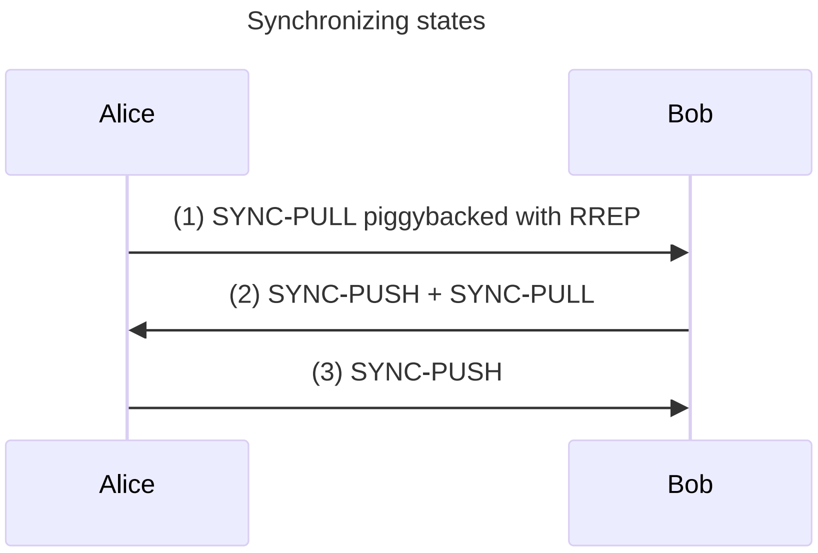
A.10.4 Adding a Message

When sending a new message it is first added to the local state before it can be synchronized with the contact. The version number of the message is set to be one more than the largest version number in the entire local state for the given contact.

The message can optionally include another group secret in order to invite this contact to the corresponding group.

A public key of the user is included in the message and will be used to identify the message sender. Lastly the entire message is signed with the corresponding private key as described in the encoding of a delta section.

A.10.5 Synchronizing States


Synchronizing states
Alice
Bob
(1) SYNC-PULL piggybacked with RREP
(2) SYNC-PUSH + SYNC-PULL
(3) SYNC-PUSH
Alice
Bob

1. When a session is established between two synchronizing contacts, the receiver of the `RREQ` uses the payload attribute of the `RREP` to encode a list of digests in the form of a `SYNC-PULL` packet.

2. When the `SYNC-PULL` packet is received, the attached digest can be used to compute a sequence of deltas containing the messages that the other device has not received yet. The computed deltas are encoded in a `SYNC-PUSH` packet and sent back. Simultaneously a second `SYNC-PULL` packet is created and sent along to perform synchronization both ways.

3. The node now merges the deltas of the `SYNC-PUSH` packet into its local state. When the second `SYNC-PULL` packet arrives, a corresponding `SYNC-PUSH` packet is created and sent. When this packet is received and handled the synchronization process is complete.

A.10.6 Digests

A digest is a set of all known `Public keys` in the local state together with the largest version number seen by any message from that respective node. A digest can be encoded in a `SYNC` packet and is used to derive the deltas.

A.10.7 Deltas

A delta encodes a single message that the other node does not have yet. When synchronizing a sequence of deltas with all messages that the other node does not have yet is sent in order to reach identical local states.

An ordered sequence of deltas can be computed using a digest received from another node by a `SYNC` packet. Now all messages with a version number larger than the version number associated with the `Public keys` in the digest will be included in the delta sequence.

It is important that deltas are ordered such that for any two deltas with the same `Public keys`, the one with the largest `Version` comes after the other one. This ensures that if only a subset of the full delta sequence is received, the messages are still in order and deltas generated from future digests will include all missing messages.

A.10.8 Incremental updates

If two nodes have synchronized such that they are both up to date and still maintain the same session the synchronization occurred on, they can use incremental updates to keep synchronized. This is relevant if one node either sends a new message, or receives a new message from a third node in the same group. Here, instead of starting the entire synchronization process over between the two original nodes, the node can simply send a single `SYNC-PUSH` to the other node.

A.10.9 Authentication

When a node receives a `SYNC-PULL` packet, the signature of the packet MUST be verified using the senders public key. Similar each delta encoded in a `SYNC-PUSH` packet MUST be also authenticated by verifying its signature using the public key included within.

Appendix B

User Test Sheet

This evaluation is a part of a Master's thesis by Troels Andersen and Viktor Kløvedal. A part of the thesis is to develop a mobile app that implements our own protocol.

The goal of this test sheet is to guide you through the app and the various common use cases we want it to provide, and get feedback along the way. All feedback will only be used anonymously.

Installing the app

Currently the app is only available on IOS. Scan the following QR code, and follow the instructions given.

`https://testflight.apple.com/join/KI4zyDpW`

Individual Tasks

The following tasks are meant for you to go through on your own.

Task 0: Read the welcome screen

Go through the welcome screen to get familiar with the functionality of the app. If you accidentally close the welcome screen, you can reopen it under Settings.

Q1: The welcome screen was easily understandable.

☐ Strongly agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly disagree

Q2: Additional comments. ____

____

Task 1: Get a contact

- Find a person and link devices. You can potentially do this for a couple of people to get more contacts to talk with.

Q3: Were there technical problems with linking your devices? ☐ Yes ☐ No

Q4: It was intuitive to perform the task.

☐ Strongly agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly disagree

Q5: Additional comments. ____

____

Task 2: Have a conversation with a new contact

- Connect to the network if not done already.
- Select a contact and send messages to them and see them respond.

Q6: I was able to send a message. ☐ Yes ☐ No

Q7: I was able to get a response. ☐ Yes ☐ No

Q8: It was intuitive to perform the task.

☐ Strongly agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly disagree

Q9: Additional comments. ____

____

Task 3: Create or join a group

- Either create a new group or join another's group.
- Invite multiple contacts to the group
- Send messages to the group

Q10: Were you able to join or invite someone to the group? ☐ Yes ☐ No

Q11: The flow of the conversation was easy to follow.

☐ Strongly agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly disagree

Q12: Was anything confusing or not intuitive when performing this task?
☐ Yes ☐ No

Q13: Additional comments. ____

____

Task 4: Leave the network and return

- Move away from other people (e.g. by going into another room or further down the hall)
- While not close to other people, attempt to send messages to some of your contacts and groups.
- Move back so that you are within range of other people.
- Send a message to a couple of contacts.

Q14: Did you lose connections with your contacts? ☐ Yes ☐ No

Q15: The messages you sent were reliably delivered to people once you returned.

☐ Strongly agree ☐ Agree ☐ Neutral ☐ Disagree ☐ Strongly disagree

Q16: Additional comments. ____

____

Task 5: Have fun!

- Play around with the app and try things.

Q17: Do you have any general comments on the app? ______________________________

____

Coordinated Tasks

When you have completed all of the previous tasks, meet back at where we started. The following tasks will be done in a coordinated fashion with everyone.

Stand in a line

Everyone stands in a line. The two ends link and try to establish a connection. Everyone else can try to communicate to their existing contacts.

When we are ready, everyone moves further apart, we repeat this until the connections stop working.

Everyone joins one group

We create one group and share it with each other until everyone has joined. Then everyone tries to send messages in the group to see how well the network holds up.