Bachelor Thesis

Implementing Delay-Tolerant Routing for a Decentralized Instant Messenger

Fakultät IV - Elektrotechnik und Informatik

Internet Network Architectures

Research Group of Prof. Anja Feldmann, Ph.D.

Felix Ableitner
22. Juli 2016

Prüferin: Anja Feldmann, Ph. D.
Betreuer: Theresa Enghardt, M.Sc.
Mirko Palmer, M.Sc.

Ich versichere an Eides statt, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Felix Ableitner

Zusammenfassung

Zentralisierte Plattformen wie Facebook oder Google werden immer populärer. Auf diesen Plattformen haben Firmen die volle Kontrolle über Nutzerdaten. Dezentralisierte Software funktioniert ohne jede Kontrolle durch Firmen, und kann Kontrolle an die Nutzer zurückgeben. Allerdings ist dezentralisierte Software viel komplizierter zu entwickeln, weil Knoten in einem Peer-to-Peer-Netzwerk kommunizieren müssen. Probleme können nicht einfach mit einem zentralen Server gelöst werden.

In dieser Arbeit nehmen wir einen bestehenden, dezentralisierten Instant Messenger, und erweitern das Protokoll. Vorher wurden Nachrichten auf eine sehr ineffiziente Weise verschickt, und konnten nicht zugestellt werden, wenn der Zielknoten offline ist. Frühere Forschung bietet Protokolle für entweder dezentralisiertes, oder verzögerungstolerantes Routing (Englisch: Delay-Tolerant Routing). Jedoch bietet kein Protokoll beide Features. Aus diesem Grund entwerfen und implementieren wir ein eigenes Protokoll basierend auf "Spray and Wait" und AODVv2. Das Protokoll nutzt Relay-Knoten um Nachrichten zwischenspeichern, bis sie zugestellt werden können. Dazu kommt ein effizienten Routing-Protokoll, das Ressourcennutzung minimiert.

Das Ergebnis ist eine Proof of Concept-Implementierung für einen Instant Messenger, der komplett dezentralisiert ist. Dennoch ist mehr Arbeit nötig, um die Nachrichtenzustellung zu verbessern, wenn der Zielknoten offline ist. Außerdem sollten wichtige Features implementiert werden, etwa die Unterstützung von Mediendateien. Während unser Protokoll speziell für einen Instant Messenger entworfen ist, könnte es auch für andere Szenarien angemessen sein.

Abstract

Centralized platforms like Facebook or Google are becoming more and more popular. On these platforms, companies have complete control over user data. Decentralized software works without any company's control, and can be used to give control back to users. However, developing software without any central server is much more complicated, as nodes have to communicate in a peer-to-peer network, where problems can not simply be solved with a central server.

In this thesis, we take an existing, decentralized instant messenger app, and extend the protocol. Previously, messages were sent in a very inefficient way, and could not be delivered if the destination node was offline. Previous research provides protocols for either decentralized or delay-tolerant routing, but no protocol that has both features. For this reason, we design and implement a custom protocol based on "Spray and Wait" and AODVv2. The protocol uses relay nodes that buffer messages until they can be delivered. In addition, an efficient routing protocol is used to minimize resource usage.

The result is a proof of concept implementation of an instant messenger that is completely decentralized. Still, more work is needed to improve the message delivery in cases where the destination is offline. Also, new features should be implemented, like support for media files. While our protocol is specifically designed for an instant messenger, it may also be appropriate for other scenarios.

Contents

1 Introduction

Today, more and more people use centralized platforms like Facebook, Google, or WhatsApp. This is a problem, because it gives a lot of control to the operators of these platforms. They can easily monitor communications, access user data, and censor their users, all without any oversight. Even if one trusts those platforms, law enforcement agencies may force them to abuse this power against their users (for example with national security letters in the US). In contrast, decentralized software has no single point of failure. Data is stored on the user's own devices, so unauthorized access is much more difficult. Communication is done in a peer-to-peer topology, which makes the software very resilient against failure or censorship on the network level. Additionally, decentralized software can take advantage of unused resources on user devices (CPU, storage, network). On the other hand, many problems that are easily solved in centralized software become very difficult challenges in decentralized software.

One of the most popular examples of decentralized software is the bittorrent protocol [4]. It takes advantage of file sharding to distribute files between many different clients, using their bandwidth for distribution. This means there is no need for an expensive central server, but idle resources on user devices can be used instead. If the system is designed properly, it will scale automatically to support more users. At the same time, it allows the distribution of copyrighted content. To take content down, copyright owners would have to shut down every single node hosting the content, which proves impossible in practice. While copyright infringement is morally questionable, it proves the censorship resistance of decentralized data distribution. In contrast, the centralized Napster server was easily taken down by law enforcement. Another advantage is the security of decentralized networks, where nodes do not have to trust anyone else. For example, the Bitcoin network secures a total investment of around 10 billion euros.

In this thesis, we want to show that it is possible to create a chat service that is completely decentralized. We do this by extending an instant messenger app for Android that we built previously, called Ensichat. Ensichat uses peer-to-peer communication over Bluetooth and Internet for message transmission. It is available under an open source license on Github. An overview over the current features and project structure will be given in a later chapter. There are two main problems in the underlying protocol preventing Ensichat from

becoming a fully featured instant messenger. First, it uses a simple flood routing algorithm, which means every message is sent to every node. This is obviously very inefficient. We will review a selection of existing, decentralized routing algorithms and implement one that is suitable to our use case. Second, messages can not be delivered if the destination node is offline. Instead, the message is dropped silently. Based on previous research, we will develop a solution to this problem that does not rely on any centralized server, and implement it.

The thesis is structured as follows. Section 2 gives an overview over other centralized and decentralized messengers, as well as related scientific work about decentralized routing. Section 3 describes the project structure of Ensichat. In section 4 we give a general, high-level overview of our custom routing protocol. Section 5 describes the details of our implementation. After this, section 6 describes possible attack vectors, and how they can be mitigated. Finally, we draw our conclusions in section 7.

2 Related Work

There are various messengers that are decentralized to some degree. We will have a look at them, and what kind of network topology they use. Additionally, we will describe what advantages and disadvantages their approach to decentralization has. Our goal for this thesis is to create an instant messenger that has end-to-end encryption, and a network structure that does not require any servers. This should make it easier to add new nodes to the network, without any extra setup. Additionally, the protocol should be efficient for usage on mobile networks, and allow media transfers. Other than decentralized messengers, we also look at WhatsApp as an example of a centralized instant messenger.

After this, we will have a look at scientific work in the area of decentralized messengers, as well as delay-tolerant routing. This includes a general overview over the existing work and important concepts, as well as specific routing algorithms. Here, we want to find an algorithm that is applicable to our use case, and which we can implement later.

2.1 Other Messengers

XMPP works similar to email. There are various servers, which connect to each other and exchange messages on behalf of users. Each user is registered to one server, and has an ID of the form *username@example.com*. This approach to identification has the advantage of being familiar to users, who can simply login with a username and password. However, this model of registering at a single server could result in a centralization of users at a single, popular service. In contrast, Ensichat does not rely on server domains for addressing. Instead, each node has an address which is the address of their public key. As another weakness, XMPP only supports end-to-end encryption through the OTR protocol, which is not supported by all clients. The protocol itself is entirely text-based, so it takes up more bandwidth than a binary format. This is especially a concern with mobile devices, and when used with media files, which have to be base64 encoded.

IRC is a decentralized message protocol that is mostly used for group chats, but also supports private messaging. The protocol only supports text messages. It has a similar network architecture as XMPP, with a strict separation between servers and clients. Clients can only connect to servers, while servers can connect with clients and other servers. This results in a spanning tree, where each server

acts as a central node for the rest of the network it sees. While connections are typically secured with TLS, users always have to trust the servers, because they can read and even change all messages.

WhatsApp is a good example of a centralized instant messenger. It shows what is possible if developers do not have to worry about decentralization, but can put a great focus on usability. This is especially helpful for less experienced users. For example, it uses phone numbers for user identification, so users do not have to exchange any additional ID. Automatic end-to-end encryption was recently enabled for all users. Internally, WhatsApp uses a protocol based on XMPP. However, there is no way to use standard XMPP clients with WhatsApp, and the topology is entirely centralized. Even though end-to-end encryption makes it impossible to read message contents, WhatsApp can access all metadata and analyze it.

Matrix is a relatively new project, that aims to become a standard for decentralized, real-time communication over the Internet [1]. The network architecture is similar to XMPP, with a set of servers that connect to each other, and clients that connect only to servers. Matrix has some advantages over XMPP, like history synchronization and better support for multiple devices. Also, there are plugins which allow "bridging" between Matrix and other networks like IRC. It remains to be seen how successful this project will be compared to the established alternatives.

In short, none of these messengers fit our requirements of being decentralized and delay-tolerant. XMPP and IRC both use a strict separation between clients and servers, and require considerable effort to run these servers. WhatsApp is not decentralized, but has great usability. Matrix has a somewhat wider goal of becoming a standard for all live online communication, while Ensichat is currently focusing on mobile chat only.

2.2 Decentralized Routing

In this section, we will have a look at existing research in the area of delay-tolerant networks and decentralized routing.

Delay-Tolerant Network: A delay-tolerant network is one that allows message delivery even if end-to-end connection may never be present [9, 12]. Instead, data is stored and forwarded by intermediary nodes. Both nodes and links may be unreliable, as nodes move around. Especially mobile nodes usually have lim-

ited resources, like buffer space for messages, processing power, battery power and bandwidth. The effectiveness of a protocol can be measured with some key aspects: delivery latency, delivery ratio and bandwidth usage. A good routing algorithm should maximize probability of message delivery, while minimizing resource usage and delay. As we will see, most delay-tolerant routing algorithms discussed here are intended for mesh networks. The papers we reviewed specifically assume that nodes move around, and take advantage of this for message delivery. We will look for ways to take advantage of Internet connections, too.

Replication and Knowledge: Decentralized routing algorithms can be categorized using two properties, replication and knowledge. Replication means that a message is sent multiple times, usually over different routes. This makes it more likely that at least one copy reaches the destination, but increases resource usage. Pure replication strategies send each message over the entire network, which greatly increases the usage of bandwidth and other resources on all nodes. On the other hand, knowledge is used to send a message across the "best" route over the network. Pure knowledge strategies only send a single message. This minimizes resource usage, but increases the risk that the message is lost, for example if a connection is closed during the transfer. There are various simple algorithms which use only one strategy, while other algorithms take advantage of both knowledge and replication.

Example Algorithms: One of the simplest routing algorithms based on replication is epidemic routing. When two nodes connect to each other, they exchange all undelivered messages they have, so that each node has all messages. This means that every message will eventually be delivered to every single node. While the strategy is sure to deliver all messages with minimal delay, it also has a very high resource usage. This is because every node has to transfer every message, which does not scale well.

An example for a knowledge-based routing strategy is location-based routing. In this strategy, nodes need to know the position of all other nodes. When sending a message, it is always forwarded to the neighbor that is closest to the destination node. Unfortunately, this strategy is not practical for Ensichat, as the destination node has no way to tell other nodes its position, without sending a message itself.

Jones et al present a routing protocol based on epidemic routing [8]. Instead of using epidemic routing to transmit messages, it transfers information about the network structure. Nodes can then use this information to send messages

directly to the target node. However, it means that every node needs to know about the entire network structure. This is not feasible if the network size gets too big, or if the network changes often.

Spray and Wait: This is an advanced routing scheme that combines replication and knowledge [13]. It limits the number of message copies and transmissions without compromising performance, and is highly scalable. The algorithm uses two phases. In the spray phase, messages are forwarded to a number of relays (this number is determined by the estimated network size). In the wait phase, relays wait until they encounter the destination, and deliver the message directly to it. The spray phase uses a binary spreading algorithm, so that each node forwards half of its message copies, and keeps the other half. This goes on until there is only one copy left on each node, and limits the number of transmissions. The protocol is very scalable, simple and efficient, making it a good fit for our use case. It assumes only direct connections between nodes, so we will have to make some changes to support connections over the Internet.

Spray and focus: This algorithm is an improvement over spray and wait by the same authors [14]. It uses the same logic for the spray phase, but changes to the wait phase make it more efficient. Instead of just waiting for the destination node, relays can forward their message copy to a different node, if that node has encountered the destination more recently. This algorithm is more complicated, as every node needs to keep track of when it has encountered any other node. It is also not clear if this will be an improvement in our case, as we can take advantage of Internet nodes for relaying.

Utility Functions: The usage of last encounter time as a metric is an example of a "utility function". Some routing strategies take advantage of one or more utility functions to determine how messages should be forwarded [15]. Utility functions can be destination dependent, like "Age of Last Encounter". Here, messages are forwarded to nodes that have seen the destination node most recently. "History of Past Encounters" is an extension of this concept, that uses more information, like frequency of encounters or average encounter time. "Pattern of Locations Visited" takes advantage of nodes frequenting the same location as the target node. The "Social Networks" function tries to forward a message to friends or family of the destination. Destination independent functions use characteristics of the relaying node itself. Among them are "Amount of Mobility", "Node Resources", or "Trustworthiness".

Relay Nodes: Utility functions like the ones explained are used Spyropoulos

et al for an alternative way to improve spray and wait [16]. This strategy does not just pick the first node it encounters as message relay. Instead, it uses a utility function to choose the best available node as relay. Examples for utility functions presented in the paper include Last-Seen-First (nodes that encountered the destination most recently), Most-Mobile-First, or Most-Social-First. The paper notes that a malicious node could advertise a very high fake utility value, and absorb a large number of messages. For this reason, a utility function that is not advertised by the potential relay itself would be preferable.

Proactive and Reactive: Decentralized routing algorithms can be separated into proactive and reactive [11]. Proactive (or table-driven) protocols always try to keep network information up to date. This means every node has to maintain routing information for every other node. This information is periodically updated as the network topology changes. Proactive protocols are best used when the network is mostly static, as messages can be sent without any delay for route discovery. In contrast, reactive (or on-demand) protocols only try to find a route when they have to send a message to an unknown node. The sending node will ask other nodes for routing information towards the destination node, and send the message when it gets this information. While a route is active, route maintenance is done based on topology changes. If a route is not used, it times out after some delay. Reactive protocols have a potentially slower initial connection, but use less resources if only few messages are sent. In addition to proactive and reactive protocols, there are also hybrid protocols which combine the advantages of reactive and proactive routing. But these protocols are much more complicated to implement, as they need much more additional logic to work. Because our use case has moving nodes, short connections and sparse messages, a reactive protocol would be best suited.

AODVv2: "Ad Hoc On-demand Distance Vector Version 2" is a draft for a decentralized, reactive routing protocol [3]. When a node wants to send a message and does not know a route towards the destination node, it sends a route request, which is flooded over the network. When the destination node receives the route request, it sends back a route reply. Each node along the way stores the next hop towards origin and destination in its routing table. Messages can then be sent along this path. There are also route error messages, which will be sent if a connection to a node is closed, or if a message can not be forwarded. Each node has a sequence number that is incremented with every message, so other nodes can see how recent a message is.

AODVv2 seems perfect for our use case. However, it does not support delay-tolerant messaging. For this reason, we will combine AODVv2 with the message relays from spray and wait, to create a routing protocol that has both decentralized and delay-tolerant routing.

3 Ensichat Overview

Ensichat is a decentralized instant messenger. It is written in the Scala programming language and licensed under the GNU General Public License, version 3. The project is available on Github [2]. It is divided into three components. The most important one is the core library, which handles all network connectivity, message serialization and routing. The Android app is the main user facing component, and provides an easy-to-use interface. The server project is a thin wrapper around the core library, which allows users to run Ensichat on their server and support the network.

3.1 Network Topology

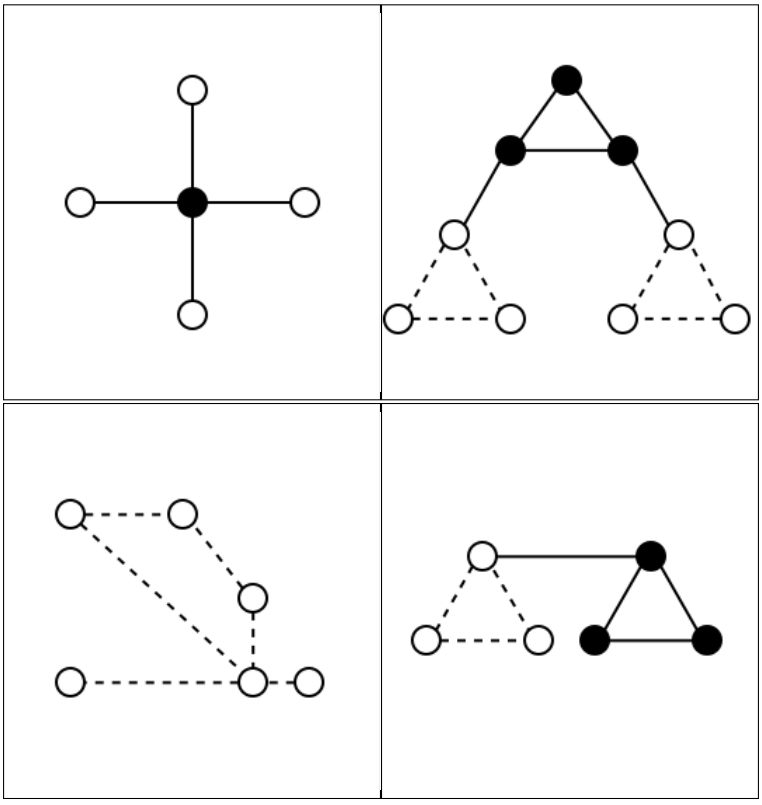

Figure 1: Possible network topologies. Black nodes are servers, white nodes are mobile devices. Filled edges represent Internet connections, while dashed edges are Bluetooth connections.

As mentioned before, Ensichat uses a peer-to-peer topology for communication between nodes. All nodes are equal, can move around, and connect or disconnect at any time. This explicitly includes server nodes. Servers are required because Internet Service Provides block any incoming connections on 4G mobile networks. The same is usually true on public wifi networks. Instead of connecting to other mobile nodes directly, we use servers as intermediaries to forward messages. In contrast to XMPP and IRC, the protocol itself does not differentiate between server nodes and nodes running on a phone. Although Ensichat uses servers, it is still fully decentralized. The servers do not form a central point of failure, because everyone can start a new server, that has the same capabilities as all other servers. This makes the protocol trustless, because there is no need to trust any particular node. Users only need to trust the math underlying the system. Various possible topologies are shown in Figure 1. Note that Ensichat also works without servers, and that no specific network topology is required.

Ensichat supports connections over Internet and Bluetooth. The only difference between Internet and Bluetooth connections is how they are established. Internet connections are opened when a node connects to a known IP address. Bluetooth connections are opened by nodes continuously scanning for other Bluetooth devices, and connecting if they also run Ensichat. Note that Internet connections are only supported between server-server or phone-server, and Bluetooth connections only between phone-phone. This limitation can be removed in future versions of the implementation. Once a connection is opened, the same encoding and protocol is used for all messages. This means that there is no difference between a network consisting only of Internet nodes, or one made only of Bluetooth nodes, or a mixed network, except for transport attributes like range or bandwidth. Every node has an address, which is the SHA-256 hash of its public key. The asymmetric keys are generated with RSA, and have a 4096 bit key size. When two devices connect, they first exchange a *ConnectionInfo* message, which contains each device's public key. This allows for messaging between neighboring nodes without any additional setup. If a node wants to send a message, it needs the public key of the target node.

3.2 Message Sending

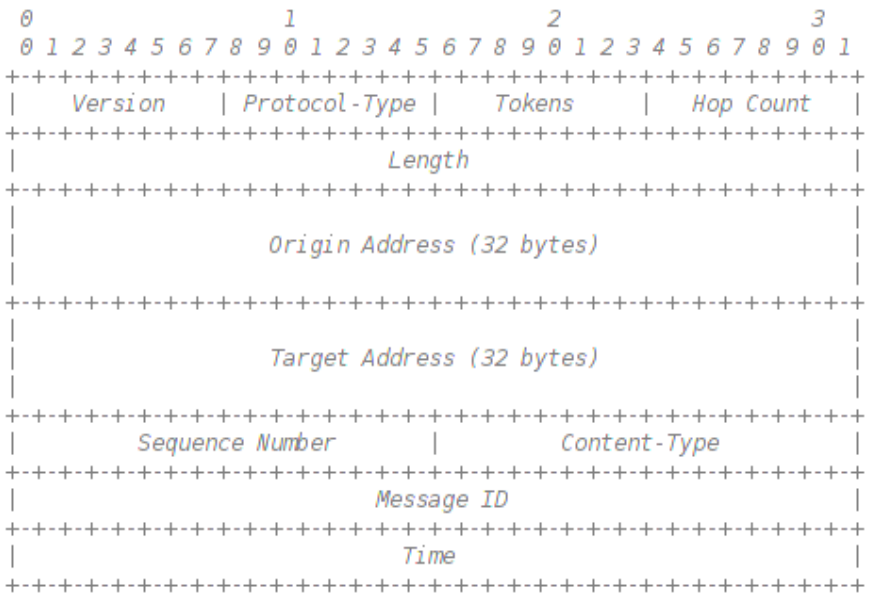

Figure 2: Structure of a message header. Message ID and time are only included for user messages.

When sending a message, the sender computes the target's public key, which is written to the message header, along with other information like sender address, time, and message type (see Figure 2). All user messages are signed and use end-to-end encryption. Some messages are sent automatically by the protocol, like the AODVv2 messages we will implement later. These are only signed, not encrypted, as other nodes also have to read them. Each message also contains a sequence number, which is used to determine if a node has seen a particular message before. If a message has been received before, the new copy is discarded. For routing, Ensichat currently uses a simple flooding algorithm. To send a message, the sender simply forwards it to all neighbors, who also forward it to their neighbors, and so on. If the destination node is not online, the delivery will silently fail.

3.3 Core Library

The core library handles all networking and protocol operations, including message transmission over the Internet (TCP/IP). Various classes represent the messages that are sent over the network, and handle their (de-)serialization. For sending, message headers and bodies are both converted to a binary format, which is more space efficient than a text based one. Figure 3 gives an overview that shows how the different projects and classes relate to each other. All routing functionality is implemented in the core package. The exception is the *BluetoothInterface*, which provides connectivity over Bluetooth using Android specific APIs. Both the Android *ChatService* and the server *Main* classes use the *ConnectionHandler* class. While the server project provides only minimal functionality to control the core, the Android project has various classes that display contacts, chats, and other information.

- The core library's central class is called *ConnectionHandler*. It provides a public interface for other projects using the library. There are various functions that allow sending a message to a particular node, or getting information about other nodes. It automatically handles high-level routing functions, like requesting a route to send a message. All other core classes are controlled by the *ConnectionHandler*.
- The *Crypto* class handles all operations that are related to cryptography. This includes key generation, message signing and encryption, as well as signature verification and decryption.
- The *Database* class currently stores message history and the list of contacts. In the future, it can be extended to store more information.
- The *Router* class handles message forwarding to neighbor nodes, making sure they are sent out through the right connection.
- The *TransmissionInterface* and its implementations handle the actual transmission of serialized data. New transport methods can easily be implemented with this interface.

The library is platform independent, and has interfaces that allow it to run on Android as well as desktops and servers. Most of the changes discussed in this thesis will affect the core library.

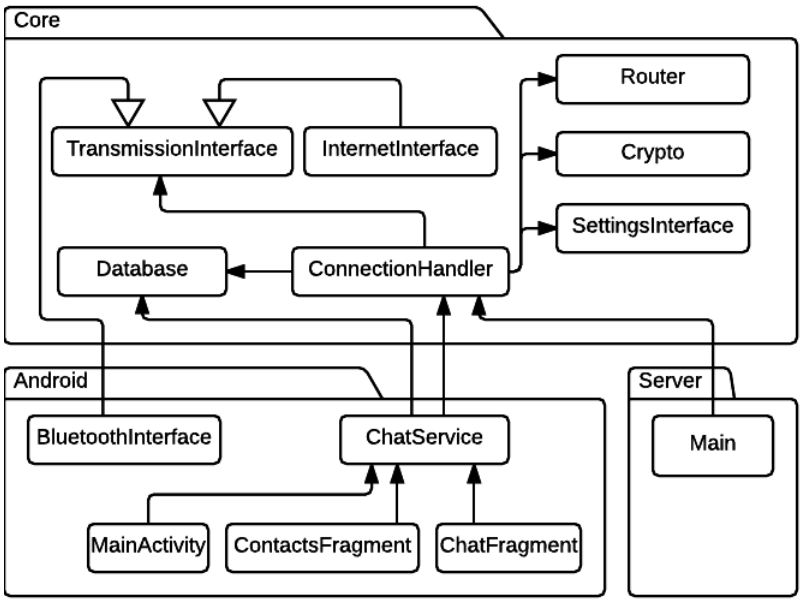

Figure 3: Class diagram of the core, library and Android projects. Only the most important classes are shown in this diagram.

3.4 Server

The server project is a thin wrapper around the core library. As mentioned before, it is required because smartphones do not usually allow incoming connections. This means we can not easily connect directly to another node over the Internet. Instead, we use servers running the Ensichat library to provide connectivity between mobile nodes. It implements the interfaces required by the library, so that it can be run as a standard command line executable. A startup script for systemd-based Linux distributions is also included. The project is intended for enthusiast users who want to help the network, or for users who are especially concerned about their privacy. These users can put the project onto a web server, and add the IP address in the Android app. The app will then always attempt to connect to this server and use it for message delivery. We provide a default server, which is preconfigured in the app. Different servers connect to each other, forming a mesh where messages are forwarded and delivered. In the course of this thesis, we will take further advantage of their availability

and storage space to act as relays for offline messaging (Section 4.2).

3.5 Android App

The Android application is the main user-facing component of Ensichat. Its goal is to provide an easy-to-use messenger application, similar to other instant messengers. After starting the app for the first time, the user can set their name and status, which can be seen by other users. The main screen shows a list of contacts, which is empty at first. The top bar shows information about current connections, including the usernames of connected nodes. Tapping it gives a full list of connected nodes, with their usernames and statuses. These can simply be added as contacts by tapping on them. In the contact list, tapping on a contact opens a chat view. At the moment, only plain text messages are supported. In the future, it should also be possible to send images or other media files. The complexity of decentralized routing is mostly hidden, and only visible in the list of connections, and in a configuration option for server addresses. For every device, an identicon is shown. This identicon is a simple graphic that is deterministically generated based on the device's public key [7]. Users can compare the identicon shown on their device with that on their contacts device, to ensure they have the correct public key, and avoid a man-in-the-middle attack. The app consists of some central classes:

- The most important class is the *ChatService*. It provides a background service that wraps around the library's *ConnectionHandler*, and can be used by other classes.
- The *ContactsFragment* class shows a list of all contacts a user has, and allows to open chats.
- Chats are shown in the *ChatFragment*, where users can also write new messages.
- Other *Fragments* and *Activities* show information like app settings, profile information, and active connections.
- Together with other related classes, the *BluetoothInterface* handles all Bluetooth connectivity, and allows Ensichat to send messages over Bluetooth.

All in all, the Android app provides a wrapper around the core library, that hides all the internals of decentralized routing. This should make it easy to grasp for all users.

4 Approach to Decentralized Routing

Based on our previous research, we develop a routing algorithm that uses both delay-tolerant and decentralized routing. This algorithm will be based on the papers "Spray and Wait" and "Routing in Delay-Tolerant Networks Comprising Heterogeneous Node Populations" for message relays. The forwarding logic is based on the AODVv2 protocol. Our goal is to minimize data transfers, while ensuring successful delivery with minimal delay. While the app is currently used only by a small number of users, we hope that the protocol will easily scale to thousands of users. This is because only few messages have to be sent to discover routes and keep the network running. However, we rely on the assumption that each node sends relatively few messages. If this assumption is not true, scaling may be hindered, as route discovery is relatively expensive for the network.

4.1 Message Relays

Retry Number	Exact Retry Delay	Approx. Retry Delay
1	10^1s	~10s
2	10^2s	~2m
3	10^3s	~17m
4	10^4s	~3h
5	10^5s	~28h
6	10^6s	~12d

Table 1: Retry times for message delivery.

As described by Spyropoulos et al [13], we use relays as intermediary nodes for message delivery, instead of sending the message directly to the target node. The relays immediately try to send the message to the target node using AODVv2 route discovery. If no route is found, the message is stored in a buffer, and we try to send it again after predefined intervals (see Table 1). The use of a message buffer with automatic retries ensures that messages do not get lost if the target node is currently not reachable. It also allows message delivery between two nodes that are never online at the same time. These retry intervals have to achieve a balance between fast message delivery, and low resource usage, because every route discovery floods messages over the entire network. Our assumption here is that nodes are online most of the time. If a node is offline for some hours already, it is unlikely that it will come online very soon. If the first route requests failed, we wait longer for subsequent requests, to avoid high network

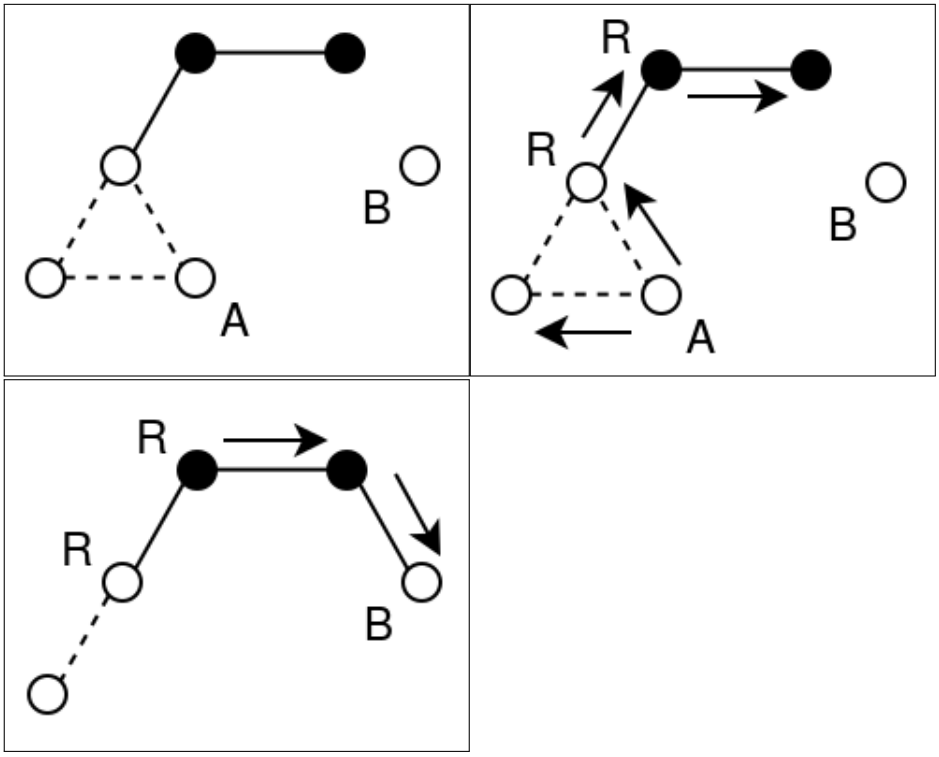

Figure 4: A is sending a message to B. The message is buffered on relays (marked R), and delivered when B connects.

load. However, this poses a limitation as users might receive messages only after a significant delay, even if they were online between route requests. The current retry values are picked arbitrarily and are relatively high. We hope that future research and usage testing will find better retry intervals. The intervals could also be set by each node individually, depending on their available resources.

As an alternative to fixed retry intervals, nodes connecting to an existing mesh could actively ask for messages, by flooding a request for buffered message over the network. Relays could use this message as a trigger to start the route discovery, and deliver the buffered message. Such an event based approach means messages would be delivered almost immediately after the nodes connect, whereas the current approach may take multiple hours of waiting in the worst case. At the same time, fewer attempts at route discovery would be necessary, reducing the overall resource usage. Unfortunately, we did not have time to implement this improvement, but hope it can be done in the future.

Messages are removed from the relay's buffer on any of the following conditions:

- A route to the target was found, and the message was sent to it
- We tried and failed to discover a route for six times
- Ensichat is stopped or restarted on the relay (because messages are only stored in RAM)

After sending the message to relays, the sender also tries to discover a route and send the message to the target. If this does not work, it uses the same retry intervals. In contrast to relays, it keeps retrying the message delivery until it receives a confirmation message from the destination node. The message itself is stored in the persistent database, so it can also be accessed and resent after a restart.

For relaying, we define a number of forwarding tokens. Each forwarding token represents one copy of the message, without the bandwidth usage of transferring the message itself multiple times. For the time being, we use a hard coded value of 3 forwarding tokens for every message. If the network size increases in the future, this value should also be increased according to the formulas by Spyropoulos et al [15].

The first relay is chosen by the sender of a message among his direct neighbors. For this, we use a simple utility function based on total connection time to a node. For every node, we store the total cumulative time we were connected to it. This means whenever we connect to a node n, we start a timer t_n, and stop it when we disconnect. When we want to send a message, we check this connection time for every neighboring node, and pick the one with the highest value for t_n. The intuition behind this is that nodes with longer connection time are better connected in general. This is especially true for Internet servers, which are connected to many nodes, and should be picked as relays first. At the same time, we do not have to rely on data that nodes report themselves, keeping the network trustless. If a node is chosen as a relay, we forward $\lfloor \frac{c}{2} \rfloor$ of our tokens to it, where c is the total number of tokens we have. This process is repeated by the sender and relays until every node has only one token per message left. Figure 4 shows how a message is transmitted over relays to the destination node.

Due to the number of independent relays, it is very likely that every message will be delivered soon. However, delivery can fail if none of the relays can reach the destination. This can happen for any of the following reasons:

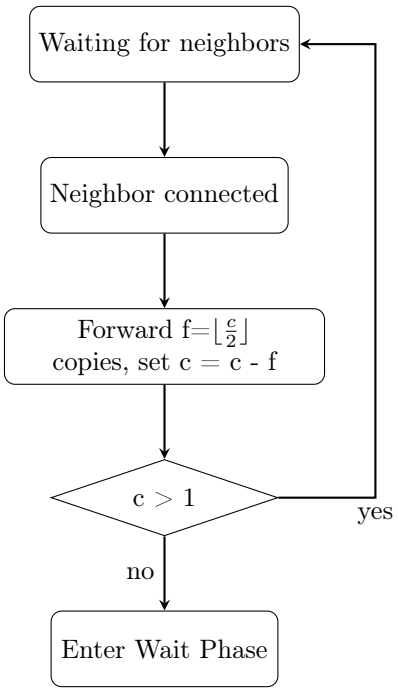

Figure 5: Flowchart of a node in the spray phase for a given message.

- Relays are taken offline for any reason
- The buffer gets too full, so some messages are dropped
- None of the relays can find a route to the destination node, and the message expires

However, even if all relays fail, the message can still be delivered by the original sender. As discussed earlier, the sender always keeps his own messages in the buffer, until they are confirmed delivered. Once the message arrives at the destination node, it will send a confirmation message back to the original sender (but not to the relays). The same routing algorithm is used to send the confirmation message. A flow chart for the relaying logic can be found in Figure 5. Figure 6 shows how messages are forwarded to relays, which spread it to other relays and do a route discovery with AODVv2. After a node finds a route to the destination, it is delivered, and the destination node sends back a confirmation.

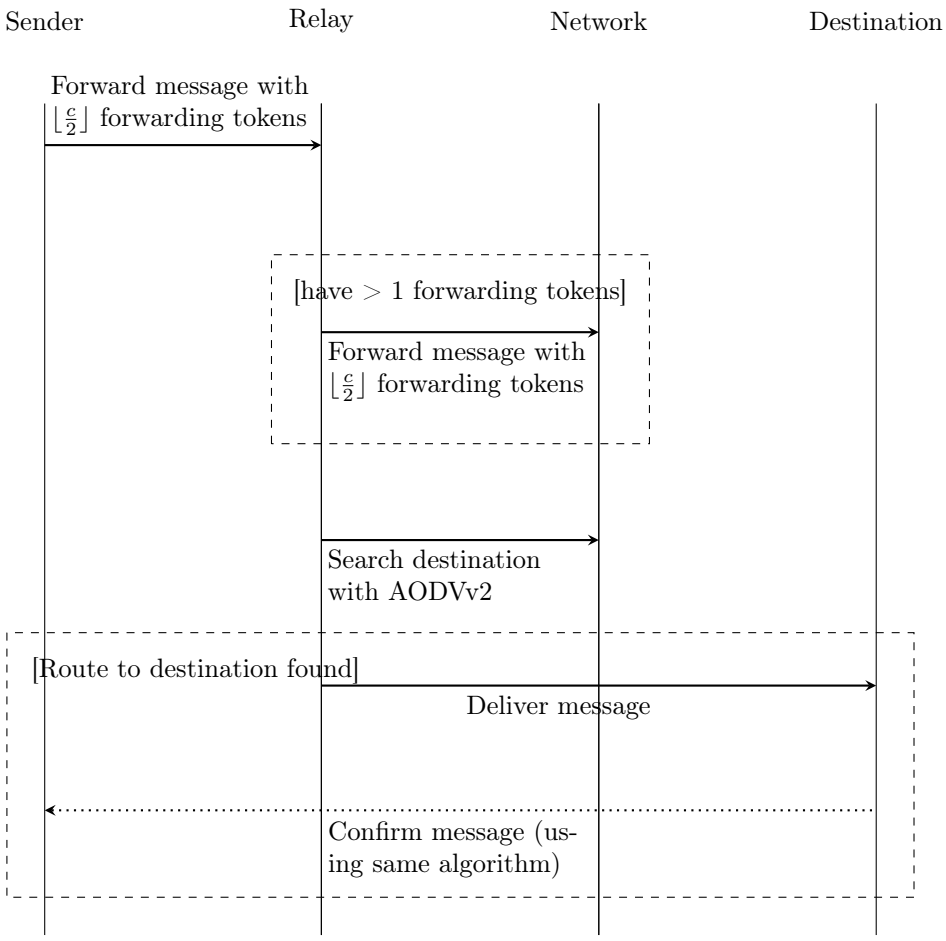

Figure 6: Message transmission between devices. Note that the sender performs the same discovery actions as the relay. Additional relays are not shown.

4.2 Routing

Our routing algorithm is inspired by AODVv2, but does not actually implement the standard [3]. AODVv2 uses three types of messages for its routing. Route Requests (*RREQ*) are sent to ask for a route to a specific destination address. They are flooded over the entire network, until they reach the destination. When a node receives an *RREQ* message addressed to it, it sends back a Route Reply (*RREP*). Both *RREQ* and *RREP* contain the sender's sequence number, so other nodes can easily tell whether a message is recent or not. Additionally, both message types contain a metric value, which in our case is the number of

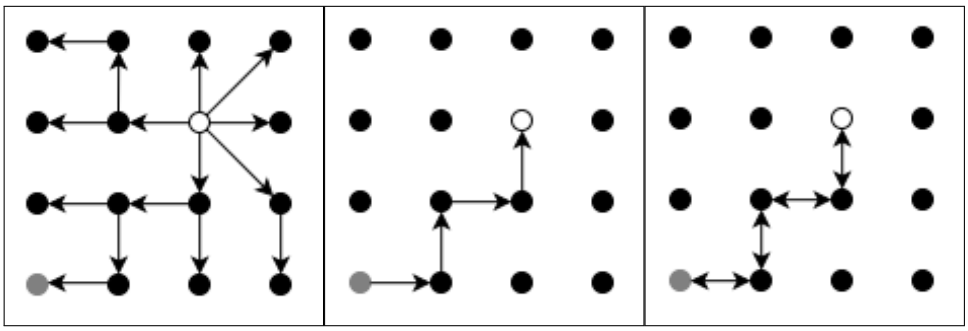

Figure 7: Visualization of the AODVv2 route discovery process.

hops between the nodes.

When an intermediary node receives an *RREQ* message, it stores the neighbor it first received the message from as next hop towards the sender. After that, it forwards the message to its neighbors. The *RREP* will then come back through the route formed by intermediary nodes. Each intermediary node then stores the neighbor it first received the *RREP* from as previous hop toward the *RREQ* destination node. After the *RREQ* and *RREP* have been delivered, a route between the nodes has been established, and messages can be sent along it. Figure 7 shows a graphical representation of the route discovery process. First, the white node sends a route request, which is flooded over the entire network. When the destination node (grey) receives the message, it sends back a route reply. With the routing information on the intermediary nodes, a route between white and grey is formed.

The third type of messages are Route Errors (*RERR*). These are sent to inform that a route is no longer available, in a number of cases:

- An *RREP* message can not be delivered, because the next hop towards the *RREQ* sender was disconnected.
- A connection to a neighbor was closed.
- An *RERR* message was received from a neighbor and is forwarded.

RERR messages are forwarded along all routes that used the broken link. Every node that knows about the route will then mark it as invalid, and will request a new route when needed.

ADOVv2 itself has many features which do not make sense for our use case. The protocol uses IP addressing, which we have to replace with our addressing

based on 32 byte node addresses. Because this is a major change, we also do not use the message format described in the AODVv2 protocol. This means that our custom protocol is not compatible with any other AODVv2 implementation. AODVv2 also supports "router clients". These are devices with their own address, but which do not support the protocol itself. Instead, they rely on other nodes to perform actual message transfers for them. We completely leave out any functionality related to router clients, as all Ensichat nodes implement the protocol. Before enabling any connection, AODVv2 checks that it is bidirectional. This is always true in our case, both for Internet and Bluetooth connections. For this reason, we leave out all functionality related to these checks. Additionally, AODVv2 has support for different "metric types", to determine the distance between nodes. For simplicity, we just use the hop count, and leave out support for different metric types.

5 Implementation

In this section, we describe our changes to Ensichat, and what we had to do to make our combination of decentralized and delay-tolerant routing work. We also give an overview over the tests we implemented.

5.1 Delay-Tolerant Routing

To implement our new routing algorithm, we need a variety of new classes. The easiest ones are the classes *RouteRequest, RouteReply* and *RouteError.* They simply hold the message parameters, and provide methods for serialization to a byte array, and deserialization back into an object. These classes represent the messages which are used by the AODVv2 protocol for route discovery.

Sequence Numbers: AODVv2 requires every node to have a sequence number. This number is a counter that is incremented and sent with every new message. It has a maximum value of 65535 (the highest unsigned integer that fits into 2 bytes). After reaching this value, it starts again at 0. The sequence number is included in *RREQ, RREP* and *RERR* messages to differentiate between them. Any two message with the same sender and sequence number are treated as identical. This is used to ignore a message if it has been received already, and to determine which message from a specific node is more recent. Additionally, text messages sent by Ensichat have a "Message ID". This is a 4 byte counter that is used to differentiate between user messages. A node can never send two different messages with the same message ID. Contrary to the sequence number, it is never reset, so it can also be used to refer to a specific message.

Message Sending: To send a message, we first have to request a route to the destination node. As described before, this is done with a route request, which is sent by the *ConnectionHandler* class whenever we try to send a message and do not have a valid route towards the destination. When other nodes receive the *RREQ* message, they forward it to their neighbors, and keep track of the previous node where they first received the *RREQ* from. The address of this node is stored as the next hop towards the sending node in the *LocalRoutesInfo* class. Through this, a path is formed from destination to sender, by the nodes between them. The destination node then sends back a route reply, which is automatically passed along this path. Again, nodes store the previous node where they received the *RREP* from as next hop towards the destination. After the *RREP* arrives at the sender, a bidirectional path is formed between both

nodes, and normal messages can be sent. Figure 8 shows which classes are involved in sending a message, and the different code path taken if we have to perform a route discovery first.

Route Management: The routes themselves are stored as entries in the *LocalRoutesInfo* class. Every entry contains the destination address, last known sequence number of the destination, next hop towards the destination, and the number of hops needed. Routes are in one of three states: *Active*, *Idle* or *Invalid*. Routes are marked as active if they have been used in the last 5 seconds, after this they are marked as idle. Routes are marked as invalid if a *RouteError* has been reported for them. Invalid routes are ignored. Public methods provided by *LocalRoutesInfo* are *addRoute()*, which adds a new route based on an *RREQ* or *RREP* message, *getRoute()*, which returns a valid route for a given address (or the *None* value), and *connectionClosed()* which marks an existing route as invalid.

Rate Limiting: The class *RouteMessageInfo* keeps track of *RREQ* and *RREP* messages that have been sent by the local node. This is to make sure we do not spam the network with too many unnecessary protocol messages. Before we sent any *RREQ* or *RREP* message, we use the method *isMessageRedundant()* to check if we should actually send it. We only send a new message if there has not been a message to this node in the last 300 seconds, and we have not recently received route information from that node.

Relay Nodes: Whenever the user sends a new message, we also send this message to relay nodes. The number of relays a message is forwarded to is limited by forwarding tokens, which represent copies of the message. Currently, every new message has the default of 3 tokens. Relays are picked with a utility function based on the total time we were connected to a node. The longer we were connected, the more likely we will choose it as a relay. The connection time for each node is stored in the *Database*, and updated whenever we disconnect from a node. To pick a relay, we first take a list of all our neighbors, and select the total connection time for each node from the *Database*. After this, we sort the nodes by longest connection time. We then forward $\lfloor \frac{tokens}{2} \rfloor$ to the first node in the sorted list. This process is repeated with subsequent nodes in the list, until we only have 1 token left. Relays in turn use the same algorithm to forward their extra tokens to other relays. Just like other nodes, relays also use AODVv2 to attempt route discovery, and deliver messages to their destination.

Message Buffering: Both the original sender and the relays attempt to find

a route and deliver the message as soon as they first receive it. While the route discovery is active, the message is stored in the *MessageBuffer*. This class keeps track of all undelivered messages. Whenever a route to a new node is discovered, the *ConnectionHandler* checks if we have any messages for that node. If we do, the messages are removed from the buffer and sent over the route to their destination. If no route can be found, the message is kept in the buffer. We retry the route discovery at the intervals detailed in Table 1. If a relay can not find a route after the 6th try (or about 12 days), the message is deleted from the *MessageBuffer*. Relays also lose their buffered messages if Ensichat is restarted, as the buffer is only stored in RAM at the moment. However, the original sender keeps the message in its persistent database and will keep retrying even after the 6th failed attempt at route discovery.

Message Confirmation: When the destination node receives a message, it sends back a *MessageReceived* confirmation, containing the message ID of the received message. The *MessageReceived* confirmation is sent with the same algorithm, which means it will also use relays and AODVv2 routing. Only after receiving this confirmation will the sender stop retrying the message delivery. This ensures that every message will eventually reach the destination, but might cause some additional network traffic.

5.2 Unit Tests

Most existing and new classes in the Ensichat library have unit tests. These make sure that individual classes work correctly. For *RREQ*, *RREP* and *RERR*, we added simple tests to verify that an existing message can be serialized, then deserialized, and still has the same values as before. For the *Router* class, we test that it sends messages to the correct neighbor, and that messages where we do not have a valid route are handled correctly. For the *LocalRoutesInfo* class, we check that the correct route is returned for a given destination, and that timeouts are respected. The tests for *RouteMessageInfo* pass various *RREQ* and *RREP* messages, and check that they are correctly allowed or disallowed, based on properties like hop count or sequence number.

5.3 Integration Tests

To test our new implementation, we also wrote an integration test suite. It is a simple wrapper around the Ensichat core library, which starts a number of

nodes on the local device, on different ports. These nodes are then connected to each other, forming a virtual mesh.

The tests include a simple scenario called *testNeighborSending()*, which checks that messages sent to a direct neighbor are delivered correctly, within a short timeout. The *testMeshMessageSending*() function creates 8 virtual nodes, and sends various messages between all of them, making sure that each message is delivered correctly. Another function, *testRouteChange*(), creates a mesh, then forces a route discovery by sending messages. After the messages have been delivered, a single node is shut down, forcing route errors to be sent. Subsequent messages have to use a different route to reach their destination. Simple relaying is tested by *testIndirectRelay*(), which starts a few nodes, sends a message, and shuts down the sending node. Only then is the destination node connected to the mesh. This assures that relays actually deliver the message to the destination node. Various other tests also exist, which test specific details of the protocol and implementation.

After cloning the git repository, the integration tests can be run with the terminal command `./gradlew integration:run`. Ensichat uses 4096 bit public key pairs, and generating them can considerably slow down tests. If this is the case, the key length can be temporarily changed in the file `core/src/main/scala/com/nutomic/ensichat/core/Crypto.scala` (variable `PublicKeySize`). Additionally, the tests seem to fail randomly at some times. As a workaround, we advise to run only one of the test functions in `integration/src/main/scala/com/nutomic/ensichat/integration/Main.scala` at a time. These problems seem to be caused by bugs in the test code itself, which could not be fixed yet.

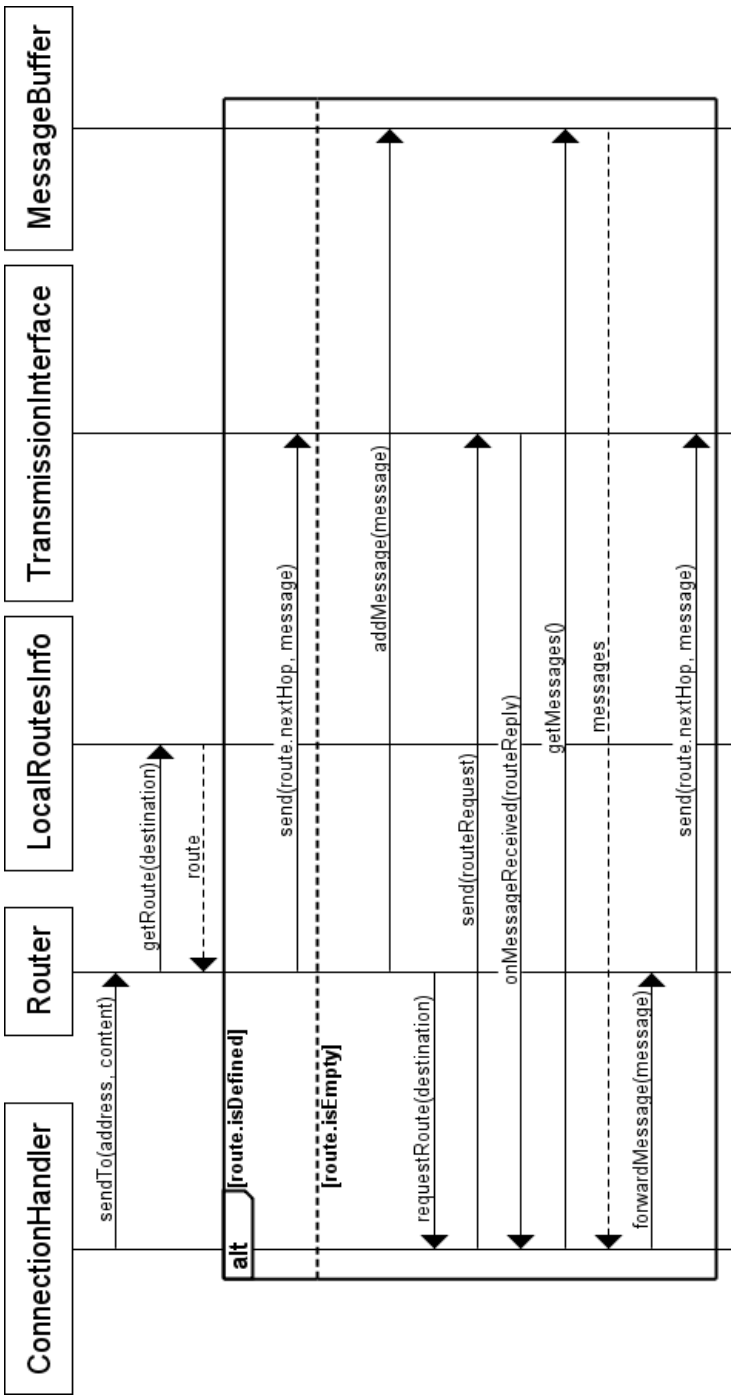

Figure 8: Sequence diagram showing how a message is sent.

6 Security Considerations

Security is a very important aspect of any application that works with sensitive data, such as private messages. In this section, we will give an overview over possible attacks that could be used against Ensichat users and their data. Some of them are problems in the current implementation, and can be completely mitigated in a future version (like signing of header data and perfect forward secrecy). Others are inherent problems of decentralized software, and the risk can only be reduced to some degree (sybil attack).

6.1 Header Data is not Signed

At the moment, header fields in each message are not signed by the sender. These fields contain various information like sender, receiver, sequence number, and other fields. An attacker could send packages with a fake sender and/or receiver address, or change header fields of an existing message. Fixing this attack vector is relatively trivial, as message signing is already in place.

6.2 Perfect Forward Secrecy

Ensichat does currently not implement perfect forward secrecy. This means an attacker can collect encrypted messages, and aquire the receiver's private key at a later time to decrypt all of them [10]. If perfect forward secrecy is implemented, every message is encrypted with a separate public key. Because of this, the compromise of a single message does not compromise any other messages. It would be desirable to implement this as an additional measure to protect user privacy.

6.3 Man in the Middle Attack

The metadata of messages is not encrypted, as it has to be read by intermediary nodes. This means that any node which sees a message can see all of its metadata (most importantly sender, receiver and time). This data could be analyzed and correlated to build a profile of individual users. Note that an attacker could only see a subset of all messages, assuming he does not control a majority of the nodes. As a countermeasure, Ensichat could use onion routing, similar to TOR [5]. Messages would be packed into a container message and sent to a

random node. The node would unpack the message, and forward it to the actual destination node. A man in the middle would be unable to detect which nodes communicate with each other.

As another attack vector, the attacker could give his own public key to users, and forward messages between them. The users would think they are sending messages directly to each other, but are instead sending them to the attacker. This is why users are encouraged to compare their public keys in the Android app.

6.4 Sybil Attack

As in any decentralized network, the possibility of a sybil attack exists [6]. For this, an attacker spawns many nodes, giving him a large influence on the network. In Ensichat, the attacker could use these nodes to delay or inhibit message delivery by not forwarding messages. He can also silently drop messages that he should relay, breaking offline messaging. Or he could spam relays with lots of messages, causing them to drop other messages. This threat is somewhat reduced, because nodes always pick as relay the nodes with the longest overall connection time, so the attacker needs to run his nodes for some time first. Additionally, it may be possible to disrupt the route discovery of AODVv2. As an additional countermeasure, there could be a proof of work required to start a node, or an explicit reputation score for each node.

7 Conclusion

In this thesis, we took an existing decentralized instant messenger, and greatly improved the efficiency and usability of its routing algorithm. We showed that it is possible to design and implement a decentralized, delay-tolerant instant messenger. This means that messages can be sent over the network with minimal resource usage. Messages can also be delivered if the sender and destination node are never online at the same time. All this functionality works without a central server, and is completely trustless.

Existing protocols did not fit our requirements, so we designed and implemented a custom routing protocol based on AODVv2 and Spray and Wait [3, 13]. Because we assume sparse messages, route discovery is only performed when a message should be sent, saving resources. All messages are sent via relay nodes. These relays buffer the message until the destination node is online, and then deliver the message.

While we have already implemented the protocol together with some integration tests, it is still a proof of concept, with much room for improvement. Before the application can be considered stable, there have to be tests with actual users. Such testing will show how practical the protocol is, and how well it scales. Of particular interest is the retry interval, which determines when relays retry the delivery of a message. The retry interval has to strike a balance between fast message delivery and low network resource usage. Only practical testing with many users can determine ideal values for this. Alternatively, an event-based approach could be researched and implemented, as described in section 4.1.

Additional future work should be done to fix known security problems, and detect any additional vulnerabilities which may exist. At the same time, more features should be implemented, like sending media files, or supporting group chats. These changes should be implemented carefully, to keep the protocol efficient and avoid unnecessary resource usage.

The entire project is available under an open source license on Github [2]. Interested users can simply install the Android application from Google Play. There is no setup required to use the app, besides simply adding the contacts.

8 References

[1] Matrix faq. Retrieved on 29. 06. 2016 from `https://matrix.org/docs/guides/faq.html`.

[2] Felix Ableitner. Ensichat on github, 2014. Retrieved on 29. 06. 2016 from `https://github.com/Nutomic/ensichat`.

[3] S. Ratliff C. Perkins, Futurewei. Ad hoc on-demand distance vector version 2 (aodvv2) routing, April 2016. Retrieved on 29. 06. 2016 from `https://datatracker.ietf.org/doc/draft-ietf-manet-aodvv2/16/`.

[4] Bram Cohen. The bittorrent protocol specification, 2008. Retrieved on 29. 06. 2016 from `http://www.bittorrent.org/beps/bep_0003.html`.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[6] John (JD) Douceur. The sybil attack. January 2002.

[7] Phil Haack. Identicons as graphical digital fingerprints, 2007. Retrieved on 29. 06. 2016 from `http://haacked.com/archive/2007/01/22/Identicons_as_Visual_Fingerprints.aspx/`.

[8] Evan PC Jones, Lily Li, Jakub K Schmidtke, and Paul AS Ward. Practical routing in delay-tolerant networks. *IEEE Transactions on Mobile Computing*, 6(8):943–959, 2007.

[9] Evan PC Jones and Paul AS Ward. Routing strategies for delay-tolerant networks. *Submitted to ACM Computer Communication Review (CCR)*, 2006.

[10] Hugo Krawczyk. Perfect forward secrecy. In *Encyclopedia of Cryptography and Security*, pages 921–922. Springer, 2011.

[11] Shima Mohseni, Rosilah Hassan, Ahmed Patel, and Rozilawati Razali. Comparative review study of reactive and proactive routing protocols in manets. In *4th IEEE International Conference on Digital Ecosystems and Technologies*, pages 304–309. IEEE, 2010.

[12] Jian Shen, Sangman Moh, and Ilyong Chung. Routing protocols in delay tolerant networks: A comparative survey. In *The 23rd International*

Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC 2008), pages 6–9, 2008.

[13] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259. ACM, 2005.

[14] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Spray and focus: Efficient mobility-assisted routing for heterogeneous and correlated mobility. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 79–85. IEEE, 2007.

[15] Thrasyvoulos Spyropoulos, Rao Naveed Bin Rais, Thierry Turletti, Katia Obraczka, and Athanasios Vasilakos. Routing for disruption tolerant networks: taxonomy and design. *Wireless networks*, 16(8):2349–2370, 2010.

[16] Thrasyvoulos Spyropoulos, Thierry Turletti, and Katia Obraczka. Routing in delay-tolerant networks comprising heterogeneous node populations. *IEEE Transactions on Mobile Computing*, 8(8):1132–1147, 2009.