Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Extensions to a Peer-to-Peer Instant Messenger

Bachelor Thesis

Christian Cadruvi

ccadruvi@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zurich

Supervisors:
Philipp Brandes, Tobias Langner
Prof. Dr. Roger Wattenhofer

September 2, 2014

Acknowledgements

I would like to thank my supervisors, Philipp Brandes and Tobias Langner for their assistance and guidance during this bachelor thesis. They were always very helpful and had useful inputs for this thesis.

Abstract

In this thesis we explain our modifications to a peer-to-peer based instant messenger which has previously been developed in several other projects. At the start of this project, the messenger consisted of two different clients that were developed apart from each other, namely an Android client and a client which was developed for the Pidgin user interface for PCs.

The Android client is written in Java with Android specific extensions, where as for the Pidgin client, a plug-in written in C communicates with the user interface of pidgin as well as the core implementation in Java.

A large restructuring of both projects was necessary to make the implementation of new features easier.

The features developed include group chat messaging, this is the extension of a chat, in which only two users communicate, to a larger number of users in the same chat. The synchronization of chat messages is a core feature of this thesis, i.e., to synchronize all chat messages between different devices using a newly developed approach. This requires identifying those chat messages, which are not yet on all devices associated with the same user, while trying to send as few messages as possible. This is useful to be able to carry on a conversation on a different device. As a last feature, we developed file transfer. These features extend the already existing instant messaging clients on both Android and Desktop.

Contents

Chapter 1

Introduction

Instant messaging is a type of online chat, that allows users to send short messages over the Internet. With the arising use of smartphones as well as social networks, that often include some kind of instant messaging system, chatting has become more and more relevant.

There are many different instant messaging applications available, but all of them lack in certain aspects. Chat applications can be categorized into two architecture types: Client-server architectures and decentralized architectures.

A client-server architecture usually stores messages as well as files on servers, which may be undesirable for users, as they do not have control over the data. However, client-server architectures are easier to implement, as the communication always goes through the server, but this also makes it a potential bottleneck.

Decentralised architectures are more difficult to implement, as the storing of information can not be done at one central point. It also has to be considered, that messages can not always be sent over the same path.

Most chat applications therefore use a client-server architecture, which may or may not be encrypted. Encryption of data becomes more and more important, not only for sensitive business data but also for the common user.

Another aspect of instant messaging applications is the platform they run on. Most programs stick to one kind of platform, they run on only one operating system. When a user changes his [1] location, it is difficult to continue a conversation on the way or even at the new place. It is thus desirable to support multiple platforms, where users can use a single account for desktop and mobile platforms. This makes switching between mobile and desktop devices possible and it is easy to carry on a conversation.

The approach we used tries to pick the best from the user's perspective, i.e., a decentralised architecture that uses end-to-end encryption and there exist clients for both desktop and mobile platforms. The instant messaging system we developed uses a peer-to-peer architecture with encryption to prevent access to any messages or files sent. Our approach makes sure that data is stored only on the devices participating in conversations.

[1] In this thesis the male form is used, this applies to both genders similarly.

Chapter 2

Related Work

There are some similar instant messengers available which provide cryptographically secure text messaging and file transfer such as TorChat or RetroShare. These two are available only on desktop.

TorChat is available for Linux and Microsoft Windows and uses Tor hidden services as its underlying network. TorChat does not seem to be updated anymore, the last commit to the project happened two years ago [4]. It is very important to have an up to date messenger for security reasons.

RetroShare tries to solve the same problem, by providing an open source platform for private, secure and decentralised communication [1] on desktop platforms Microsoft Windows, Linux and Mac OS X. It provides instant messaging, serverless e-mail, filesharing and chat rooms. It features end-to-end encryption using OpenSSL and authentication based on GNU Privacy Guard.

Threema [3] and TextSecure [2] are both mobile instant messengers developed for Android, both offer a secure end-to-end encryption. Both of them rely on a client-server architecture but claim that the server operator can not access any data. TextSecure supports this statement by making its code open source.

2.1 Previous Work

In previous projects, an Android client [6] as well as a desktop client [7] using Pidgin were developed. They were developed independently of each other, but with the intention that they can communicate with each other, thus using the same messages. This means that these two projects were completely different, using a different data model and different message handling. Both clients supported the following basic concepts:

- Adding clients to and removing clients from the contact list.
- Changing a contact and giving him an alias.
- Creating and removing a simple chat with a client in the contact list.
- Sending chat messages to single clients over a chat.

Chapter 3

Concepts

This chapter explains the concepts used in this bachelor thesis.

3.1 Overview of our System

There are two clients available, a PC Client and an Android Client. They are able to communicate with each other and a user can have an account on both clients simultaneously. A peer-to-peer network lies underneath the clients, which handles the communication. For this, TomP2P is used, which is an advanced DHT. A more detailed explanation on peer-to-peer networks can be found in the Bachelor thesis of Theodoros Bourchas [5].

It is intended that all messages eventually arrive. This means if the recipient of a message is currently offline, the message is stored in the peer-to-peer network until it can be delivered. When a client establishes a connection, it requests all messages that have not been delivered while being offline. This is helpful especially with messages that change a chat, as for example an add member message in a group chat (see next chapter).

In the following, the term *resource* is used to describe one device, e.g., one account can be associated with four resources, namely two Android devices and two PC Clients.

3.2 Group chat

A group chat is a chat with at least two users. A message sent to this chat is sent to all participants of the chat. A group chat should support the following functions:

1. Creating a new group. When creating a new group, a message needs to be sent to all other clients in this group, which contains the group chat identifier, the group name and all members.

2. Inviting clients to a group. When someone invites a client to a group, all clients that were previously in this group need to be informed that a new client has joined. The inviter has to send a group open message to the invitee, which contains again the group chat identifier, the group name and all members.

3. Leaving a group. The client leaving the group has to send a message to all group members, such that they do not continue to send the messages in this chat to the client who left.

4. Creating multiple groups with the same clients. A group is identified by a group chat identifier, thus making it possible to create multiple groups with the same members. Every group is also associated with a group name, to distinguish the chats for the users.

5. Sending messages to all clients. A group message can be treated similarly to a normal chat message, it is simply delivered to all participants.

6. Sending files to all clients.

The groups should be visible to all resources of a client, thus making it necessary to store this information globally.

3.3 Synchronization between devices

As it is possible for a user to use multiple clients with the same account, it is useful to synchronize chats and chat messages belonging to the same account, to be able to continue or refer to conversations on a different device. This synchronization includes all attributes of the chats, especially the chat messages.

As this project relies on a peer-to-peer network, data is not stored centrally as with most other chat programs, but rather only on the user's devices. It is necessary to note that each client stores the chats and the belonging chat messages in databases on each device separately. This implies that if a user is not online on any device, the data is not accessible and synchronization can only be handled, when two devices are online simultaneously, as only then information can be accessed and communication between clients is possible.

The attributes of a chat which are synchronized include the chat messages in each chat as well as the unique chat identifier, which is used to associate a chat with a chat message. A protocol handling synchronization of chat messages should be able to identify which chat messages have to be exchanged. The participants of chats are stored in the DHT, thus it is not necessary to synchronize them here.

When synchronizing messages, the number of messages may be very large. Thus it is not a good idea to always send all messages to all devices. Assume a device 1 stores an average of n messages in m chats and device 2 has none of these messages yet. The total number of messages is $m \cdot n$. Then every message needs to be sent from device 1 to device 2 under all circumstances. If device 2 has a part of the messages, e.g., all but 10 of them, it is useful to identify which 10 messages are missing before sending any of them. Here it would obviously be wasteful in terms of data volume to send all $m \cdot n$ messages, when the optimal solution only sends a (small) constant number of messages. If both devices which are synchronizing have the same messages, no chat message needs to be synchronized. This requires a protocol which sends as few messages as possible. The implemented protocol is discussed in the implementation chapter.

One idea to synchronize messages between two devices is to use timestamps for the synchronization, i.e., after each successful synchronization, the current time is saved as the point in time, at which the last synchronization has happened. When a device is being logged in to, only the messages received or sent after this timestamp may be needed to synchronize, but some or all of them may also have been received by both devices, as they were both online when the chat message were sent or received. This reduces the number of messages which may have to be synchronized, but does not help with deciding which exact messages need to be exchanged between the devices.

The approach we use relies on the chat message identifiers, which are used to determine which messages need to be synchronized. It also tries to make use of a timestamped approach, which has the benefit that old messages are more likely to be in an already synchronized state.

As previously mentioned, the timestamp (the time when the message was issued to be sent) is an attribute of a chat message and is thus the same on all systems. When a user logs into a device, the messages of all chats are fetched from the database, grouped by chat and ordered by timestamp. A synchronization timestamp is defined as the current time. The ordering of the chat message's timestamps is used to categorize the messages into time intervals of one hour length. For each interval, the message identifiers of all messages are concatenated and then hashed to obtain a single value per time interval, which can be used to check whether the message identifiers and thus the messages themselves in these time intervals are the same. For the following paragraph, assume that $h(x)$ is a hash function, which hashes x to some value. The symbol $\circ$ is used for contatenation, e.g., "abc" $\circ$ "def" = "abcdef".

A tree we will call Synchronization Tree in the following, such as in Figure 3.1, is built, each node contains a value which represents a checksum, namely the hash value, of the message identifiers in this interval. The leaves of the tree are the hour intervals. All chat message identifiers in one hour interval are concatenated and then the resulting value is hashed. More formally, assume that in one hour

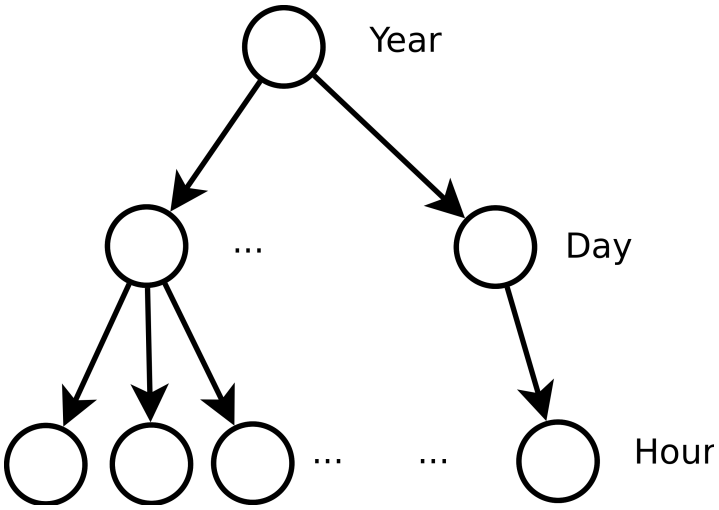


Figure 3.1: Synchronization Tree

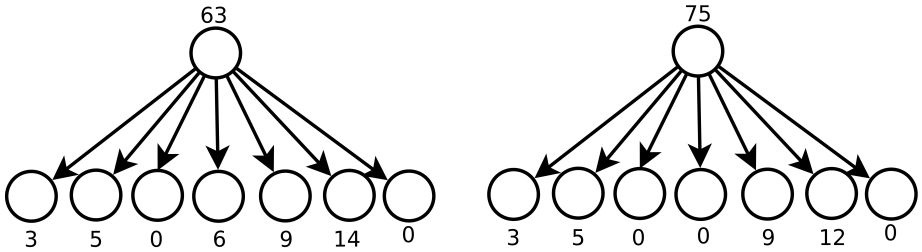


Figure 3.2: Example Scenario

interval the messages $m_1, m_2, ..., m_n$ have been sent. The chat message identifiers are $id_1, id_2, ..., id_n$ respectively. The resulting value is: $h(id_1 \circ id_2 \circ ... \circ id_n)$.

Hour intervals are grouped together to form a new node, which represents an interval of 24 hours (afterwards called day interval) and has the corresponding hour intervals as children. The hash value of the day interval is calculated by summing up all hash values of its children and hashing this value again. Assume $h(0), h(1), ..., h(23)$ are the hash values of the hour intervals, then the corresponding day interval is computed as: $h(h(0) + h(1) + h(2) + ... + h(23))$.

365 of such day intervals are again grouped together as before to form a year interval.

An example scenario is described in Figure 3.2. Assume that the left tree is the synchronization tree built with the messages from the database of device 1 and the second tree is built from the data of device 2. In this scenario, we synchronize over a day, which is 7 hours long. The rightmost child node represents the time interval from now until one hour past, the second from the right from one hour past to two hours past etc. The root node is the day interval. We can

observe that the messages over the whole day are not yet synchronized, as the values of the roots are different. This means the comparison algorithm can go down one level and check where the differences are. On the hour level it can be observed that the intervals from one hour to two hours past and from three hours to 4 hours past are different. If the two values are different and one is zero, the device with the zero value at this place does not have any chat message in this interval yet. If both are different from zero and do not have the same value, no statement can be made on which messages are synchronized and which not. Thus all messages in the corresponding intervals are sent to the other device.

Note that in the actual implementation, the trees are much larger than in this example scenario.

3.4 File Transfer

File transfer is a feature which is often seen in chat programs, especially for images. It offers a simple transfer between devices without the need to upload the file to some server and download it again on the other device.

Often the bandwidth on mobile devices is limited, as well as the data volume. Thus it is not desirable to be able to send files of any size. Another issue is security; the sending of certain files is often prohibited, e.g., executable files. If executable files such as Android apk files (apk is the package file format used in Android operating systems to install application software) were allowed to be transferred, the recipient would likely try to open said apk file and may possibly install a malicious program.

Chapter 4

Implementation

4.1 Group Chat

4.1.1 Android Client

The group chat in the Android Client is implemented straightforward. In the listview displaying open chats, there is a menu entry to create a new group. Only clients in the contact list can be selected to create the group with. When they have been selected, a group name must be entered before actually creating the group. A screenshot of this open group activity can be seen in Figure 4.1. The client that created the group stores the group chat identifier together with the contacts in the contact list. For each contact, all group chat identifiers and the group names are stored with it, such that the group is synchronized with potential other resources. An example of such a store message can be found in the Appendix A.1.

A group open message containing the group chat identifier, the group name and all participants is sent to all selected members of the group. The recipients of the group open messages add a chat with these attributes. If a group member was not yet a contact, this member is added to the contact list. A system message is inserted into the chat which contains information about the members of this chat. All members of the chat now have the option to write chat messages, to add another group member from their respective contact list or to leave the chat.

Examples of the messages sent can be found in the Appendix A.2, A.3 and A.4.

4.1.2 Pidgin Client

The architecture for the PC Client is a bit more involved. A Pidgin plug-in developed in C communicates with the Java client, which in turn communicates with the peer-to-peer network. The Java client manages the contact list, group list, the sending of messages etc., where as the plug-in is responsible for forward-

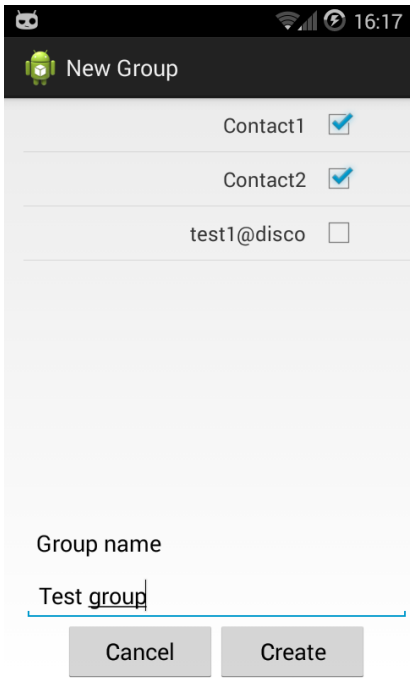


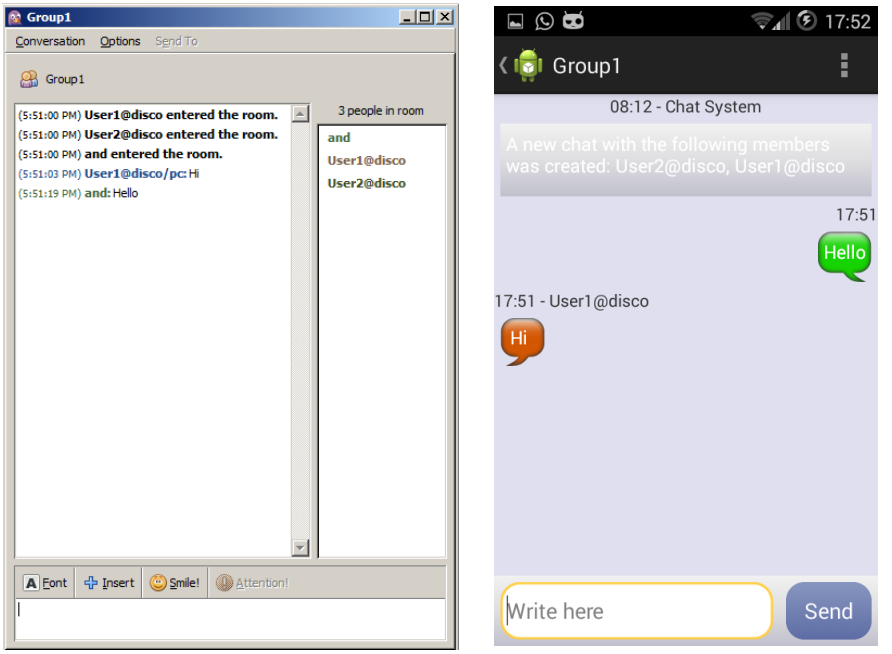
Figure 4.1: Screenshot of the OpenGroupActivity

ing the information it receives from the Java part to the user interface. The Java part can send the following messages to Pidgin:

- Open group message: An array containing the sender of the message, the members of the group, the group name and the chat identifier.
- Add member message: An array containing the name of the new member, the chat identifier and the group name.
- Group message: An array containing the sender of the message, the chat identifier, the message itself and the timestamp.

Upon the reception of these messages above, the plug-in handles them appropriately, by creating a new group for the first message, adding a member to a group for the second message and writing a message into the correct chat window for the third message. In Pidgin, there is a menu entry to create a new group. As opposed to the Android Client, first the group has to be created and then members have to be added afterwards. The plug-in sends the information needed to the Java part, which then handles the forwarding to the according clients.

It is worth noting that Pidgin has chat identifiers as well as the Java client, they are synchronized such that the Java client can tell Pidgin to insert a chat message to the chat with this identifier. This synchronization of chat identifiers



(a) Group conversation in Pidgin (b) Group conversation in Android

Figure 4.2: A group chat in both developed clients.

happens at two places, namely when the user creates a new group in Pidgin, the chat identifier is created by pidgin and forwarded to the Java client, when receiving a group open message from someone else, the chat identifier there is forwarded to Pidgin to be used there as well. An example conversation between two Pidgin clients (User1@disco and User2@disco) and one Android client (and) can be seen in Figure 4.2a and 4.2b respectively.

4.2 Synchronization between Devices

A chat message consists of a timestamp, i.e., when the message was sent, a chat identifier, which associates the chat message with a chat, a chat message identifier, which uniquely identifies the chat message and of course the contents of the message itself. The chat message identifier uses Java's `UUID.randomUUID()`, which creates a globally unique identifier [8].

Recall the Synchronization Tree explained in Section 3.3. The following paragraph explains Figure 4.3 and describes what happens with the Synchronization Tree. The tree is serialized and sent to the other resource together with the synchronization timestamp (see Appendix A.5). The other resource can then build its own tree with the same synchronization timestamp and traverse through the trees to find the hour intervals which do not match. If the root of both trees

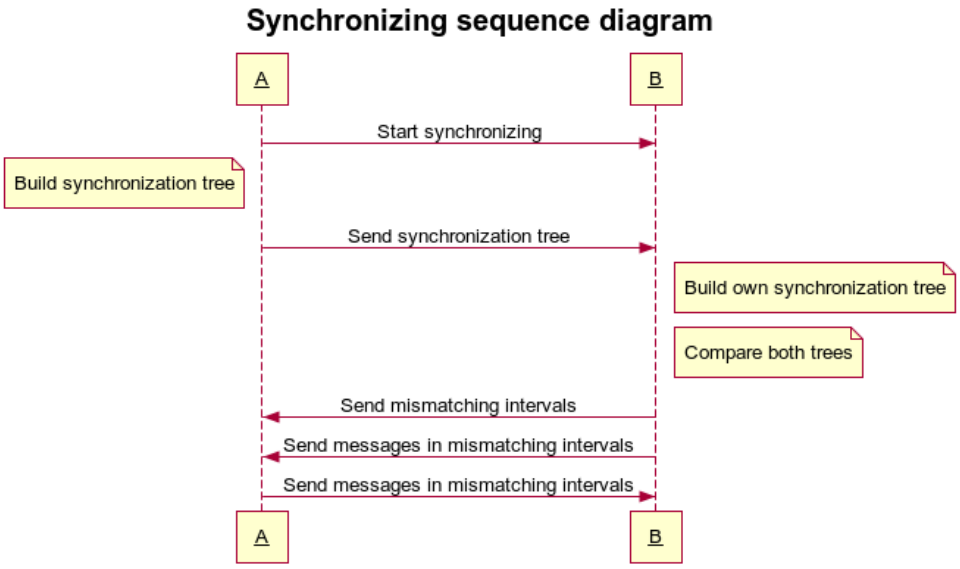


Figure 4.3: Sequence diagram of the synchronization protocol

are equal, then the chat messages are synchronized completely. If they are not equal, the hash value of at least one day interval must be unequal as well, thus narrowing down the time interval of unsynchronized messages. This can be done again to obtain hour intervals. These hour intervals can then be sent back to the resource which initiated the synchronization (see Appendix A.7), together with the messages in its own database which are in the corresponding intervals that are not yet synchronized (see Appendix A.8). Upon the reception of the intervals, the chat messages in these intervals are sent to the other resource. Note that all messages in a corresponding hour interval are sent, regardless of whether they are synchronized or not. All messages received that are not present yet are stored into the database.

All messages used in the synchronization protocol are sent to all resources, which makes it necessary for all resources that participate in the synchronization to ignore the messages sent by themselves.

A more finegrained model would allow to find smaller time intervals that are not synchronized, but this increases the height of the tree and thus the size of the message which contains the tree. It also increases the time to build, serialize, deserialize and compare trees to find mismatching intervals.

4.3 File Transfer

File transfer has only been implemented in the Android Client. A menu button in the ChatControllerActivity leads to a file explore activity, which allows the user to select a file he wants to send to the chat he is currently in. When the file was selected, two checks are executed, first whether the file type is allowed to be sent by this chat program (limited on the file types JPG, PNG, TXT and PDF) and second whether the file size is not too large (limited to 4MB). If the selected file passes both checks, there is a distinction between image files (JPG) and other files. In case it is a JPG file, it is compressed using Android's Bitmap facilities. The byte representation of the resulting file is then converted into a hexadecimal string representation to avoid that some content of the file interferes with the xml wrapper (such as an end of line character in the file). This hexadecimal representation is then sent as the message body to the other chat participants together with the file name. An example of such a message can be found in the Appendix A.9. Note that the conversion of the file to a hexadecimal representation may take some time and is thus executed in a background thread.

The client receiving a file has to decode it first, i.e., converting the hexadecimal representation back to a byte representation. Again this is done in a background thread to not block the user interface. When this is done, the file is saved to the device and a notification is shown to the user that a file has been transferred.

4.4 Architectural Changes

As, previous to this project, the Android and PC Client have been developed apart from each other, there are little similarities, the only equal part being the messages sent over the peer-to-peer network. This means especially that even the model classes, e.g., a class that defines a chat message, were not the same. A big refactoring was necessary to ease the implementation of the above explained features. This included changing most model classes in one project and adapting the code for these new classes.

The Android client was divided into two main projects, a chat protocol and an Android specific part. These two projects were at first not clearly structured, some parts that are not Android specific such as the message handling was integrated in the Android specific part instead of the chat protocol. A significant amount of time had to be invested into separating these two projects better.

At first, there was no database involved in the PC Client. However a database is of great importance when synchronizing chat messages, as the goal there is to synchronize the messages in the respective databases. Together with the refactoring of the model classes, a database was implemented for the PC Client,

which behaves similarly to the database in the Android project.

To check the functional behaviour of the added features, unit tests were developed for Android as well as the Java part of the PC Client.

Chapter 5

Conclusion

This thesis developed some core features for the already existing instant messaging applications. These features include a group chat, the synchronization of chat messages between multiple devices of the same user and file transfer on Android devices. One important aspect developed is the synchronization of chat messages, namely the combination of a timestamped approach combined with a tree like structure that helps to identify which messages need to be synchronized.

It would have been advantageous if the two projects for the Android and PC Client had been developed closer together. The reworking of these two projects, such that feature development would take less time to develop, did take a lot of time. Important aspects as parsing and handling of messages is still not unified.

There were some difficulties with the implementation of the C plug-in for Pidgin, as the Pidgin code is documented rather poorly. It is not clear what some important functions of the library do without delving deep into the source code.

5.1 Future Work

To simplify further development, the two clients have to be tied together more. This includes especially the message handling and parsing. When introducing new message types, the parsers for both projects have to be adapted separately as well as the handling of the parsed message. This further merging would also allow to write fewer tests to check the functionality of the code. Additionally, adding a wider range of clients, e.g., for other operating systems, would benefit from a separated message handling.

An important point that remains to be doing is further testing of both the network part and the clients themselves.

In parallel to this project, the cryptography of the clients is being developed, which ensures the end-to-end encryption of all messages sent as well as the authentication of clients.

Bibliography

[1] Retroshare. http://retroshare.sourceforge.net/. Accessed: 2014-08-28.

[2] Textsecure. https://github.com/WhisperSystems/TextSecure/. Accessed: 2014-08-29.

[3] Threema. https://threema.ch/de/. Accessed: 2014-08-29.

[4] Torchat2. https://github.com/prof7bit/TorChat. Accessed: 2014-08-28.

[5] Theodoros Bourchas. Distributed HashTables for P2P-Messenger. Bachelor's thesis, ETH Zurich, 2014.

[6] Pascal Fischli. Encrypted Peer-to-Peer Based Instant Messenger for Android. Bachelor's thesis, ETH Zurich, 2014.

[7] Hildur Ólafsdóttir. Peer-To-Peer Based Instant Messenger. Bachelor's thesis, ETH Zurich, January 2014.

[8] Hirondelle Systems. Generating unique ids. http://www.javapractices.com/topic/TopicAction.do?Id=56, August 2014. Accessed: 2014-08-28.

Appendix A

Appendix Chapter

In this section, the messages sent through the network which are relevant for this thesis are listed.

Store contact list: This stores a contact list with all group chats the contacts are in. The contact list below can be interpreted as User1 having two contacts, Contact1 and Contact2 are in a group with User1 called Groupname1 and Contact1 is also in a group with User1 called Groupname2.

Listing A.1: Store contact list

```
"<?xml version='1.0'?>
<contactList from='User1' to='User1' id='490308006' type='push'
    timestamp='1386045542'>
   <item jid='Contact1' name='Contact1' friends='no' action='set'>
      <groupId>2004772946</groupId>
      <groupName>Groupname1</groupName>
      <groupId>2004777943</groupId>
      <groupName>Groupname2</groupName>
   </item>
   <item jid='Contact2' name='Contact2' friends='no' action='set'>
      <groupId>2004772946</groupId>
      <groupName>Groupname1</groupName>
   </item>
</contactList>";
```

Group open message: This message creates a group with the members User1, User2, Contact1 and Contact2 called Test group.

Listing A.2: Group open message

```
<?xml version='1.0'?>
<message from='User1' to='User2' id='490308006' m-id='2004772946'
    timestamp='1396854816191' type='openGroup' groupName='Test
    group'>
   <member>Contact1</member>
   <member>Contact2</member>
</message>
```

Group leave message: This message sent by User2 tells User1 that he has left the group associated with id and will not acknowledge messages with this id anymore.

Listing A.3: Group leave message

```
<?xml version='1.0'?>
<message from='User2' to='User1' id='490308006' m-id='f94294c9-e7bb
    -421e-82c2-1f3e1a51aa17' timestamp='1396856923238' type='
    leaveGroup'>
</message>
```

Add member to group message: User1 has added a contact called ccc@disco to the chat associated with id and notifies User2 of this change to the group.

Listing A.4: Add member

```
<?xml version='1.0'?>
<message from='User1' to='User2' id='490308006' m-id='0f907729-5cfc
    -4444-9d8e-008db572bf1e' timestamp='1396856092778' type='
    addMember' name='ccc@disco'>
</message>
```

Send tree message: Note that an actual message is longer than this shortened version, as the body contains 1 integer for the year interval, 365 for the day intervals and 8760 for the hour intervals. This makes the message quite large, the body itself is over 9KB.

Listing A.5: Send tree message

```
<?xml version='1.0'?>
<message from='User1' to='User1' chatId='490308006' m-id='8e44cce3-
    f180-43c2-b9c3-a1014e6877df' timestamp='1407586772065' type='
    synchronization' name='user2' groupName='Chat with user2'>
   <body>
      2761106 1267303981 0 0 -1695429263 -389741105 9854161 0 0 0 0
           0 0 -2128380539 1718415768 834654156 0 -396266863 0 0 0
          0 0 856440964 0 -1837210594 -1264542875 0 0 0 0 0 0 0 0 0
           706044922 -432377592 0 0 0 0 -1538210205 0 0 0 0 0 0 ...
   </body>
</message>
```

Start synchronizing message: This message is used to tell other resources, that they should initiate the synchronization of chat messages.

Listing A.6: Start synchronizing message

```
<?xml version='1.0'?>
<message from='User1' to='User1' chatId='all' m-id='a87c11f9-57a0-4
    bca-9b17-3023463180dd' timestamp='1407588124988' type='
    startSynchronizing' >
</message>
```

Send intervals message: This message sends all intervals, which are not synchronized to other resources.

Listing A.7: Send intervals message

```
<?xml version='1.0'?><message from='client1' to='client1' chatId
    ='490308006' m-id='79726def-0b2b-427e-a7fa-853566c5d5e1'
    timestamp='1407589507596' type='sendIntervals' >
   <intervals>1407585907545|1407589507545</intervals>
   <intervals>1407582307545|1407585907545</intervals>
   <intervals>1407575107545|1407578707545</intervals>
</message>
```

Send old message: This message contains a message, which other resources may not have in their respective databases.

Listing A.8: Send old message

```
<?xml version='1.0'?>
<message from='User1' to='User1' chatId='905016504' m-id='9132975d-
    b4b9-4607-9f1b-8d02c84c686b' timestamp='1407588620344' type='
    oldMessage' >
   <body>sender='User1' receiver='User2' timestamp='1407588620315'
       messageId='256a2d63-30ed-4d63-8d0f-97f8f191062e' message='
       Message content' chatId='905016504'
   </body>
</message>
```

File transfer: User1 sends a file called testFile.txt with the content "content of this file" to User2.

Listing A.9: File transfer

```
<?xml version='1.0'?>
<message from='User1' to='User2' chatId='490308006' m-id='c10168cb-
    b285-4623-b397-312980ad4cec' timestamp='1409132808102' type='
    file' fileName='testFile.txt' >
   <body>636F6E74656E74206F662074686973206696C65
   </body>
</message>
```