System Administration Documentation
UniMessenger

Felix Pechtl & Kai Wiest

12.12.2020

Table of Content

1 Introduction

1.1 Motivation

With the releases of more and more messenger-service providers on an ever-growing market it has become a necessity for most people to have more than one app to communicate with different groups of people. Consequently, it can be challenging to keep a clear overview over the conversations someone has with different people.

1.2 Goal of the project

In order to improve the overview over conversations of different messengers, our plan was to implement a common client for as many different messenger-services as possible. This multi-messenger should support the most common features of the native clients.
We also evaluated the APIs of different messengers in regard to their functionality and accessibility for third-party developers. Furthermore, we intended to offer the user the possibility to handle all of his communications in one place, no matter if the chat partner is using 'Wire', 'Telegram' or any other messenger-service.

This documentation contains our analysis and the implemented messenger-services and present our conclusions. Additionally, we describe the functionality and features we implemented in our program, which is publicly available on GitHub[42].

Planned Features

The features that should be implemented in the client are:

- Support for Wire with a command-line based user interface that has a login function, a list of chats and the option to send and receive messages
- The option for the user to send and receive special messages like voice-messages or files
- Encryption of user-credentials and messages on disk
- The 'Ping' function and temporary messages from Wire
- Support for Telegram and Whatsapp

- A graphical user interface and the option to use calls and emotes

We prioritized the first four points on this list over the others. Because we underestimated the required effort to implement these features, we were not able to implement all of them. The last two points on this list, as well as the second point, are not implemented yet, but will be added in a future version of the program.

1.3 Used components

1.3.1 Java 13

We decided to use Java for this project because there is an open source implementation of a Wire client in Java publicly available[13].
Since we wanted to use the Java HTTP client which was added in Java 11, we had to use this version or a newer one. We selected version 13 because we both had it already installed when we started this project.

1.3.2 Messenger Services

Wire

We decided to start implementing the Wire client first, because Wire is a service with high security standards and still completely open source and free to use. It also has a well done documentation of the internal functionality and the API.

1.3.3 Windows 10

Since Windows is installed on the majority of desktop computers[47], we chose to start development for Windows and support operating systems like linux and macOS at a later point. This way we provide a program that most people can use very early in the development process. These people can provide feedback on a wide base for future improvements.

1.4 Disclaimer for the current version of the program

At the time this document was written, the program only runs on Windows 10.
Also certain .dll-files have to be installed manually for the Program to run. This is something that will be automated in later releases.

2 Definition of Terms

2.1 Abstraction

"An abstraction is a general concept or idea, rather than something concrete or tangible." [1]
In case of our project we are abstracting the connection between the user interface and the logic that is specific for every single messenger to be able to reuse large portions of the user interface for different messenger services.

2.2 API endpoint

"An API endpoint is a point at which an API [...] connects with the software program." [29]
The API endpoints we used for this documentation are HTTP requests to the servers of messenger-services

2.3 Asset

"Assets are larger binary entities sent between users, such as pictures." [16]
In this documentation we use the term asset to describe documents of all types (text files, folders, pictures, videos, ...), that are uploaded to a server or transported in messages.

2.4 Bearer Token / Access Token

"API requests are authorised through so-called bearer tokens. For as long as a bearer token is valid, it grants access to the API under the identity of the user whose credentials have been used for the login. The current validity of access tokens is 15 minutes." [4]
We only use the term "bearer token" and avoid the name "access token" in this documentation, to prevent misconceptions with other tokens.

2.5 Client

We define a client as a single device accessing a messengers services. This client belongs to a specific user and is registered persistent or temporarily. A single user can have multiple clients running simultaneously.

2.6 Client ID

"Upon successful client registration the server returns a client ID (Cid) which is unique per user ID" [16]
Client IDs are used in Wire to identify the device a user is connected with.

2.7 Contact

"The people whose names, telephone numbers, addresses, etc. you keep, for example stored on your mobile phone" [6]
In relation to Wire, we use the term "contact" to describe a user that has accepted someones request to communicate and can be added to a conversation.

2.8 Conversation

Conversation in common language means "A talk between two or more people in which [...] information is exchanged" [7]
In this documentation we use the term "conversation" as the container structure in which messages are stored[32]. This structure also contains some meta-information, like the initiator of the chat (or group), the conversation name and ID as well as the conversation type.

2.9 Cookie / User Token

"In order to obtain new access tokens without having to ask the user for his credentials again, so-called 'user tokens' are issued which are issued in the form of a zuid HTTP cookie. These cookies have a long lifetime (if persistent, typically at least a few months) and their use is strictly limited to the /access endpoint used for token refresh." [4]
To prevent misconceptions with other tokens, we only use the term "cookie" in this documentation.

2.10 Dynamic Link Library

"A DLL (.dll) file contains a library of functions and other information that can be accessed by a Windows program. When a program is launched, links to the necessary .dll files are created." [19]

2.11 Generic message

In this documentation we use the term "generic message" for a data structure that is used by Wire to store the content of messages[13][14][12]. We use this structure as well to send and receive messages[35].

2.12 Interface

"The point at which two different systems, activities, etc. have an influence on each other"[18]
"In its most common form, an interface is a group of related methods with empty bodies."[20]
In this documentation the term "interface" is used to describe abstract classes that can be used by other classes.

2.13 Message

"A short piece of information that you give to a person when you cannot speak to them directly."[25]
In this documentation we also use the term "message" for the structure in which we save messages and handle them[38]. This structure also contains meta information like a timestamp, a sender and a type. A message is not only limited to contain text, but can also contain files.

2.14 Ping

"A ping is a way to get someone's attention with an animation and sound."[26]
In this documentation we use the term "Wire ping" or "ping" for a special message type that is used by Wire to send an alert from one user to another[43].

2.15 Prekey

"Every client initially generates some key material which is stored locally"[16]
"The prekeys are used by other clients to initiate cryptographic sessions with the newly registered client"[16]
"Wire uses the concept of prekeys to use the protocol in an asynchronous environment. It is not necessary for two parties to be online at the same time to initiate an encrypted conversation."[16]

2.16 Repositories

For our analysis and some implemented features we used the following GitHub repositories from the official Wire GitHub repository collection[45]

- wireapp/lithium[13], later also referred to as 'lithium'
- wireapp/helium[12], later also referred to as 'helium'
- wireapp/xenon[14], later also referred to as 'xenon'
- wireapp/cryptobox4j[11], later also referred to as 'cyptobox4j
- wireapp/cryptobox-jni[28], later also referred to as 'cryptobox-jni'
- wireapp/cryptobox-c[31], later also referred to as 'cryptobox-c'

2.17 User ID

"Upon successful registration the client receives a randomly generated user ID[...]" [16]
These IDs are unique in Wire and required to identify a user. A user has only one user ID.

3 Requirements

3.1 Hardware

The only hardware that is required to run the program is a computer that supports Windows 10 and Java.

3.2 Software

Our program currently runs on Windows 10, but we will provide support for operating systems like linux and macOS in the future.

To run our program successfully, a Java installation of at least version 11 is required. It is recommended to use Java 13, because we tested everything on that version.

In addition to the Java installation it is currently required to manually add three .dll-files to the 'bin' folder of the selected Java installation. This will be changed in a future version of the program, so that the files are installed automatically. The .dll-files are required for our de- and encryption to work correctly. The dll's contain the Cryptobox, which is a connection to the Crypto-library Sodium (see Encryption).

4 Technical Implementation

4.1 Overall Project Structure

Our project is split into multiple sections. This is necessary, because of our goal to implement multiple messenger-services. There are two main parts into which we split the project.

First there is the abstract part, which contains the user interfaces and the logic part that handles the user interaction (more information in the abstraction section).

The second part contains the API communication, data storage, decryption and encryption.
The API communication depends on the messenger-service that the user wants to use. Currently we have only implemented the Wire API, but support for other messengers will be added in the future.
The code that handles data storage in our project is highly dependent on the messenger that the data comes from. This is why data storage is a subsection of the Wire implementation in this documentation.
Decryption and encryption are both part of one section that will be explained in the Encryption section.

4.2 Wire

4.2.1 Implemented Features

To provide a better overview of everything we implemented, we split the added features into the following categories. Each category contains the tests and analysis we made, the results we gained from the analysis and how we implemented the feature in our program.

- Login
- Conversations / Chats
- Messages
  - Sending
  - Receiving
  - Message Types
- Data-Storage

4.2.2 Login

Login procedure

There are multiple steps required to log into a Wire account succesfully. We tried to hide most of these steps from the user, since they can be automated once the credentials are provided or if the user was logged in before.

For a user to be able to use his account properly, we need to send a login request to the server, containing the users credentials. After a successful login we then register the client at the server. Once this step is successful as well, the user can use our client properly.

In the background we then need to verify the validity of the client every 15 minutes to update the bearer token and to avoid asking the user for a new login (see Updater).

Tests / Analysis and Results

By analyzing the login process, we wanted to find out which URL we need to use to send HTTP requests to Wire.
After that we had to find out what the server requires in the body of a valid login request and what the response contains.
The last test we had to make for the login procedure was to find out how to use the bearer tokens and cookies we got from the login requests.

We tested on the Wire API List of Operations[23] website and also analyzed packets we sent from the web-client[46] with the help of Wireshark.
Both of these tests were unsuccessful, because the website uses another URL than the real Wire client, and Wireshark found too many packets to analyze.

The other two options we had were to log the packets that the Wire web-client sent with the help of the Chrome developer tool[5] and to send custom HTTP requests with our client.
Logging the packets in chrome provided us with the correct URL that we could use in our custom HTTP requests in Java.

For the tests in Java we started by finding out how the body of the HTTP request had to look to be a valid login request.

After we completed this step successfully (figure 1), we moved on to analyzing what we could do with the information we got returned from the server, which were a bearer token and a cookie.

```
{
    "email": "[Email]",
    "password": "[Password]"
}
```

Figure 1: Login request body json structure

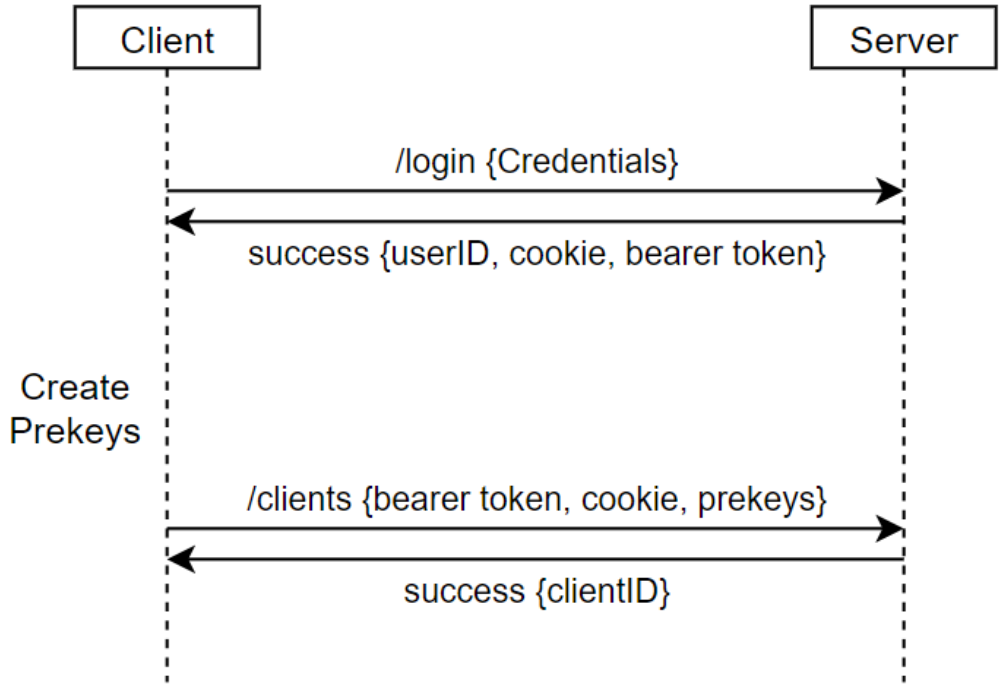

Figure 2: Client registration

The bearer token was useful for us at this point to register the client at the server to be able to receive messages.

The cookie we got from the login response could only be used to send a request to the `/access` endpoint. This returns a new bearer token that is valid for another 15 minutes.

We also read in the Wire Documentation that there are persistent and non-persistent cookies. The persistent ones are valid for 54 days and the others are only valid for one day[4].

Implementation

To implement the login function into our client, we had to ask the user for his credentials. This function is handled by the user-interface.

With the login information of the user, we then send a HTTP request to the `/login` endpoint and check if the response code we get is 200.

If it is, we store the bearer token and cookie we receive from the response and try to find out if the client is already registered.

The client needs to be registered in order to receive and send messages. This is because an unregistered client has no client ID, which is required by the Cryptobox.

To find out if the client is already registered, we request a list of all clients for the user. This list contains the clients with their IDs, but to identify if one of these clients is our client, we need to check if a cookie is sent for one of these clients. The client that has a cookie is the one we are registered as.

As soon as we know that a returned client contains a cookie, we can store its ID as ours. If we go through the list of returned clients and find no cookie, we know that our client is not registered.

To register a new client we send a HTTP POST request to the `/clients` endpoint. This request contains information like the cookie of our client and prekeys we generated.

The response we get contains the client ID, which we need for sending and receiving messages (figure 2).

4.2.3 Conversations

Display Conversation

Every time a new client is initialised or a message from a new chat comes in, the client needs to display the chat to the user. For that, an updater thread periodically checks for new conversations. If a new conversation is on the server that is not known to the client, it is added to the clients conversation list.

Tests / Analysis and Results

We needed to acquire a list of the open conversations the user had at a given time. By looking at the List of Operations from the Wire API[23], we

found the `/conversations` endpoint that returns up to 500 conversations.
By printing the return values, we found out that the conversations are returned as json-Strings, together with some meta information.
In case a user has more than 500 conversations, the `/conversations` request can contain a parameter called "start" to indicate that only conversations with an ID bigger than the passed value are returned.

Implementation

The conversations are loaded from the Wire servers by using the `/conversations` endpoint. The response returns a json-String that we parse using a json-parser[3] library. If the server has more than 500 conversations to return for the given client, it sets a so called "hasMore" key in the json-object.
If this key is set, the client requests another 500 conversations, starting at the last ID the prior request has returned.
The conversations are converted into a json-array and then every element of that array is converted into a WireConversation object[32]. These WireConversation objects can be accessed by the UI for presentation to the user through the IData interface[34].

The process of repetitively looking for new conversations is done by the updater thread.

Conversation Names

The conversation json structure that is returned by the `/conversations` request contains a name field. Unfortunately the content of this field does not always contain the real name of the conversation. On some conversations it is the correct name of the chat partner, sometimes it is the name of the user who requested the conversation and sometimes it is null.
In our tests this field always had the name of the user who initiated the conversation or the group name.

Since this name is correct for groups, we only needed to get the right names for one to one conversations. We achieved this by sending a GET request to the `/users` endpoint with the user ID acquired from the only other person in the chat, who must be the chat partner in a one to one conversation.

The `/users` endpoint requires authentication via a bearer token and can be called with multiple IDs as parameters if it is required by the implementation, but we only call it with a single ID whenever we have a conversation with an unknown name.

4.2.4 Messages

Sending and receiving messages is the most basic task a messenger client has to be able to perform. In this section we want to give insight into how we send and receive messages from and to Wire.

Sending messages

Sending messages works by posting a json structure to the `/conversations/{id}/otr/messages` endpoint. This structure contains the sender, some meta information and a set of recipients.
The recipients contain the user ID of the given recipient and the separately ecrypted message content for every single client of the user.

Tests / Analysis and Results

In the Wire List of Operations[23] we found two POST requests that take generic messages as parameter. One of them appeared to be broadcasting a message to all contacts that match certain criteria, but we did not investigate this request further, since it was not what we were looking for.

The other endpoint, `conversations/cnv/otr/messages`, seemed to be the one we needed. It takes a client ID as sender and a set of recipients as parameters and needs authentication via a bearer token.
After looking at some more captured http requests from the official web client[46], it appeared that the recipients are a set of client IDs with a cipher text added to them which seems to be the message content encrypted for the specific client of the user.

Implementation[40]

The message has to be encrypted for all devices of a recipient separately to provide proper end to end encryption. Prekeys for all devices of a user are acquired via the `/users/uid/prekeys` endpoint. This process also needs to happen for all other devices of the sender, because the message should appear in the chats of these devices too.
The encrypted messages are packed into the recipients structure (figure 3) and sent in a single HTTP request.

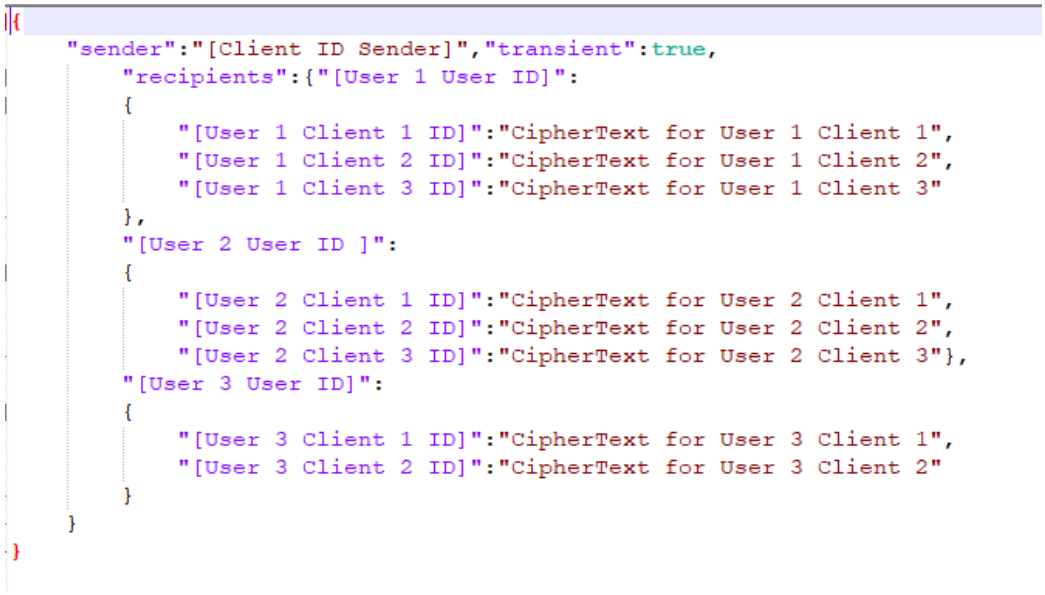

Figure 3: Recipients json structure

Receiving messages

Messages are received by getting a set of notifications from the server. These notifications contain, among other things, the new messages a user has received. Each client has to decipher the encrypted content of the message and add it to the proper chat. We request notifications periodically in our updater thread.

Tests / Analysis and Results

Our first attempt was to try extracting messages from the conversations we received by requests we sent to the `/conversations` endpoint. This proofed to be a dead end because the conversations do not have any messages stored in them.

Then we discovered the `/notifications` endpoint, which returns a list of notifications. These notifications can notify the user about a lot of things, but most of them had the type `conversation.otr-message-add` (figure 4) and contained a cipher text. After some try and error we got our Cryptobox (see Encryption) to decrypt the cipher text and return a readable String.
We also learned that trying to decrypt a message twice results in a Cryptobox error.

Figure 4: New message notification json structure

Implementation

The notifications are received by a message receiver class[39], which is periodically called by the updater thread to check for new messages.
The message receiver then passes all messages that are new to the Cryptobox (see Encryption) where they are decoded. If the message is decoded without errors and it is verified that this client is the proper recipient of the message, the message is passed to a message sorter[41], which chooses the appropriate action for the kind of message (text, ping, temporary, file).

The timestamp that marks the last received message is set to the timestamp from the last message after the messages are passed to the Cryptobox. This prevents messages from being decrypted twice to avoid errors.

Message Types

Text Messages

The messages a client receives are contained in a generic message structure. This structure can be one of multiple types. After detecting which type a message has, each message type is handled differently.

Text messages are meant to be stored on disk and displayed to the user. To achieve this, we first have to store a message we received in the conversation it belongs to. The structure in which we store messages[38] on disk is a bit simpler than the generic message structure.

Our message structure only stores the text of the message, the time it was

sent and the sender. With this information the message can later be viewed correctly.

If the user wants to see the messages of a conversation, a method in the conversation class[32] is called that returns all messages of that conversation. Since the messages have the text, time, and sender stored with them, this information can easily be displayed to the user.

To send a text message, we ask the user for the chat that he would like to send the message to and the text he wants to send. With this information we send the message to other users, as explained in this section, and additionally store the message within our message structure and add it to the desired conversation. This way the message gets displayed alongside all received messages when the messages of a conversation are displayed to the user.

Wire Ping

The main difference between text messages and pings is the generic message type that they are stored in. The generic message type of a Ping is marked as a "Knock" message, which makes it easy to identify.

Pings are not stored on disk, because they should be used to alert a user. That is why there is no point in displaying a ping in the message history of a chat. Instead a received ping creates a notification in the user interface as soon as it is received.

To send a ping we ask the user only in which conversation he would like to send it and then create the message and send it like we explained in this section.

Temporary Messages

Wire offers the option for users to send temporary messages. These messages are only available to chat participants for a certain amount of time. Our implementation of temporary messages currently only includes text messages and not file messages or anything similar.

Sending temporary messages is nearly as easy for us as sending a normal text message. The only difference is that we have to ask the user for a time and then create a generic message that has the type ephemeral. This message can then be sent like any other message.

Since the user needs to see what he sent and received as temporary messages, we had to add them to the stored message list as well. But to ensure that

they disappear after the set amount of time, we added a new variable in our message structure[38] that contains the time when the message expires.

When we receive a temporary message, we handle it like any other text message, with the exception that we add the expiration time to the message structure we store.

When a user wants to display messages of a chat and the chat contains temporary messages, the method that returns the messages that are to be displayed first checks if a temporary message is expired. If this is the case, the temporary message gets deleted from the message list and is therefor permanently removed.

File Messages

When sending a file with Wire, the client sends 2 separate HTTP requests. The first one is to upload the file to the wire server as an asset. The second one is to inform the chat partner that there is a file to be downloaded for him.

The location of the uploaded asset is returned by the first request in which the file was uploaded. Then all clients can download the asset file from the Wire server and display it in the chat.

This is how we assume this should work because we never managed to successfully send a message that contained a file. At the current state of our project the asset file can be uploaded, bot not downloaded by any other client. This will be fixed in a future release.

4.2.5 Data Storage

Our UniMessenger stores data in 3 ways, each one for a specific Purpose.

The first way how data is stored on the Disc, is in a Java encrypted file. This method is used to save the authentication cookie for persistent logins (see Login).

The file that stores this data is created by using a `CipherOutputStream`, which is essentially a `ByteOutputStream` that takes care of the encryption by using a passed `Cipher` object which contains the encryption details.
At the beginning of the file is a so called "Initialization vector" in plain text. "Using an IV is mandatory when using CBC[9] mode, in order to randomize the encrypted output. The IV however is not considered a secret, so it's okay to write it at the beginning of the file."[15]

After the IV is written, the `CipherOutputStream` stores the encrypted information.

The second way of storing data is through writing an object to a file via an `ObjectOutputStream`.
This is used to store the active chats and the messages contained in them. We use a `ConversationHandler` singleton object[33] which has a list of conversations that consist of some information on the chat partner and the messages. This `ConversationHandler` object is written to the disc and loaded by static functions of the `ConversationHandler` class.

The last way how data is stored, is by internal workings of the Cryptobox which we do not control directly. The box is storing a set of pre-generated prekeys, open cryptosessions and its local identity. The directory the Cryptobox stores data in is passed when the Cryptobox is initialised (see Encryption).

4.3 Updater Thread[44]

We decide to disconnect the tasks that have to wait for server-responses from the user-interface to make the client feel more responsive.
We implemented the task of repetitively checking for new notifications and conversations in a new thread.

This thread is started before the user-interface thread, so that the user has no delay when waiting for messages.
The updater thread maintains a list with running services, which contains only messenger-services that the user has logged in to. The services in this list need to be updated by the updater.

The updater threat updates the conversation list every 10 seconds and checks for incoming messages every 2 seconds. That is because new chats rarely appear and new messages come in much more often.
It also saves a lot of bandwidth, because conversations can not be updated individually since the request always returns all open conversations[23]. The request for notifications has the option to only return notification since a certain timestamp (see Messages section).

The updater thread is also responsible to validate and refresh the access to the service. This means that for the Wire service a HTTP request is send to the `/access` endpoint to refresh the bearer token every 14 minutes (1 minute before a bearer token becomes invalid). This ensures that the client never uses an invalid bearer token.

4.4 HTTP

We implemented HTTP requests[17] with the HttpClient from Java. This class provides all the functions we need to send HTTP requests.

To send a HTTP request with the HttpClient, it is necessary to create an instance of the client first. This instance can then be used to send a HTTP request, which needs to be created using the HttpRequest class from Java.

This class is abstract and to get a HttpRequest object from it, we need to use a static method and provide the type of request, which can be GET, PUT or POST. We also need to provide the URL, the headers and for PUT and POST requests a body[21].

To simplify the creation of a HTTP request in our code, we added the classes "HTTP" and "RequestBuilder"[36].

The HTTP class is only used to get the type of a request and then uses the RequestBuilder to create a matching HTTP request. This request is then used by an instance of the HttpClient in the HTTP class. That instance sends the request and gets a response from the server, which is returned so that it can be handled somewhere else in the code.

The RequestBuilder class contains methods for each request type we use. To create a new HTTP request, a method gets called and provided with an URL, a list of headers and for POST and PUT requests a body as well.
These variables then are put together by the method and return a complete HTTP request.

Java HTTP requests

We used the HttpClient from Java for our requests, but there is another method that we read about, which uses Javas URLConnection. We decided us against the URLConnection class, because it is not as easy to handle as the HttpClient. The HttpClient is also implemented in a later version of Java[21], which lead us to the impression that it will be supported longer than the older URLConnection (Java 8 or older[22]).

4.5 Abstraction

Goals

The main goal of our UniMessenger was to provide a client that supports more than one messenger-service. To accomplish that we need to access data from different APIs. It would be a lot of work to implement each function for each messenger individually, which is why we use abstraction. Abstraction allows us to implement all the functions we need, without worrying which messenger we use at the moment.

Implementation

To implement abstraction, we used interfaces. For each messenger we are going to create classes that implement these interfaces. The UI then only needs to call an interface and can use all the methods of the interface for each messenger. This makes it easier to implement additional messengers in the future, because we do not need to change anything in the user-interface.

To ensure that we always access the correct class that implements the interface we need, we added the APIAccess class, which requires the name of a messenger-service and the interface we would like to access. With this information it is able to return the correct class that implements the requested interface.

The interfaces we created are IConversation, IData, ILoginOut, IMessages and IUtil[37]. For each new messenger we add, we have to implement all of these interfaces in classes that are only used for the specific messenger.

Because each messenger stores data in different structures, we added a class to store Wire information, like conversations and messages. This class only communicates with other Wire classes and is not accessed from the user-interface. To provide further abstraction for data storage and to simplify its access, we will change this in the future and create an interface for the storage classes to implement.

4.6 Encryption

Wire puts great value in the security of user data, which is why the encryption algorithms are very complex and would lead too far for this documentation to be described in detail.
Wire is using the open-source library "Sodium"[10] for the process of actual encryption. As a developer it is never necessary to interact with Sodium directly because there are plenty open-source solutions for that in the official Wire GitHub repository[45].

Implementation in our Project

In our project we were using a couple of chained libraries to access the before mentioned Sodium library. These are the cryptobox4j[11], cryptobox-jni[28] and cryptobox-c[31].

The cryptobox-c is an interface that is written in the programming languages Rust and C to get easy access to the methods from the Sodium library that are relevant for Wire. The Cryptobox Java native Interface is built upon this interface and is written in Java and C. It is used by the Wire android app and provides most functions in a way that are usable for us. The cryptobox4j is basically a fork of the Java Code from the JnI that is optimized for the use on non-android systems. It still uses the C code from the JnI, which is why we had to import both (more details in the Cryptobox Libraries section).

An instance of the cryptobox4j is called a box and is unique per instance of a client. This box operates on a given directory (in our case *DataStorage/Box*), where it stores its local identity, a set of prekeys and the currently open sessions. The directory the box is supposed to operate on is passed when the Cryptobox is opened.

Key Exchange

"During client registration a client uploads prekeys [...] bundled with its public identity key [...]. These are eventually used by other clients to asynchronously initiate an end-to-end encrypted conversation, i.e. given a recipient's prekey [...] and identity key [...] the sender can derive n initial encryption key even if the recipient is offline.
The prekey with ID 65535 is the so-called "last resort" prekey. Every prekey is intended to be used only once, which means that the server removes every requested prekey immediately. In order to not run out of prekeys the last resort prekey is never removed and clients should regularly upload fresh prekeys."[16]

The client on the receiving end of the encrypted message generates something called a crypto-session by using the message he received. The sessions are stored even after the program is closed. They can be used to encrypt messages to the partner without requesting a new prekey. Whenever a Wire client thinks that it might be necessary to switch to a new prekey or in periodic time frames, the client requests a new prekey from the server and creates a new session with his partner, so that a captured key is only usable for a given time.

Encryption

To encrypt data, the data is given to the Cryptobox together with the user ID and client ID of the recipient and possibly a prekey. This information is used to either open or generate a session with the given client and encrypt the message.
If there is already an open session with the given user and no prekey is passed, the session will be used to encrypt the message. Sessions are identified by a unique name consisting of user ID and client ID.

Decryption

To decrypt data, the encrypted data is given to the Cryptobox together with user ID and client ID as before, but now the box can open a new session without a prekey, if no existing session is found. If there is a session it can be identified by user ID and client ID.

4.7 Libraries

4.7.1 Cryptobox requirements

The Cryptobox (see Encryption) requires a lot of .dll-files. Installing those can be a bit difficult, because most of the libraries that are required as a dll are not available as pre-compiled files but only as source code. That means we had to compile most of the dlls on our own.

The dll for Sodium can be downloaded from the Sodium homepage[10] and was the easiest to acquire, but finding a working version (1.0.8) still took some time.

The non-Java Cryptobox libraries had to be recompiled through their respective compilers. For the rust parts we used the RustUp Toolchain[30] and for the C files we used the Visual Studio command prompt. The Cryptobox-C[31] was compiled using the "cargo build" command from RustUp, which compiles Rust files by using the information given in the Cargo toml-file and downloading the required Cryptobox[27] in the process.

The C file was a little more complicated. For one, the C file given in the Cryptobox-C[31] repository is not the right one to work with the Cryptobox-jni[28]. The proper C file that worked for the Cryptobox-C dll was a slighty modified version located inside the Cryptobox4J[11] repository (figure 5). To compile it we used a batch script in the Visual Studio command prompt. We will not go into detail on the batch script because it is system dependent and was mostly try and error in our case. We also had to replace a uint16::max for the absolute value (65.535), because our compiler did not work with that properly.

The compiled dlls have to be placed somewhere where the Cryptobox4J is looking for them, which is either the Java home directory or the Windows system path (System32). This will be automated in the future to make it more user-friendly.

4.7.2 Maven

We use maven to compile our project. Maven manages the repositories we use and dependencies we have. To compile our program into a runnable jar, maven requires multiple "plugins" to be defined in a pom.xml file. This file contains information on how to use which dependency and repository to compile our code together with the dependencies into an executable jar file.

The IDE we used for this project has the option to select external libraries

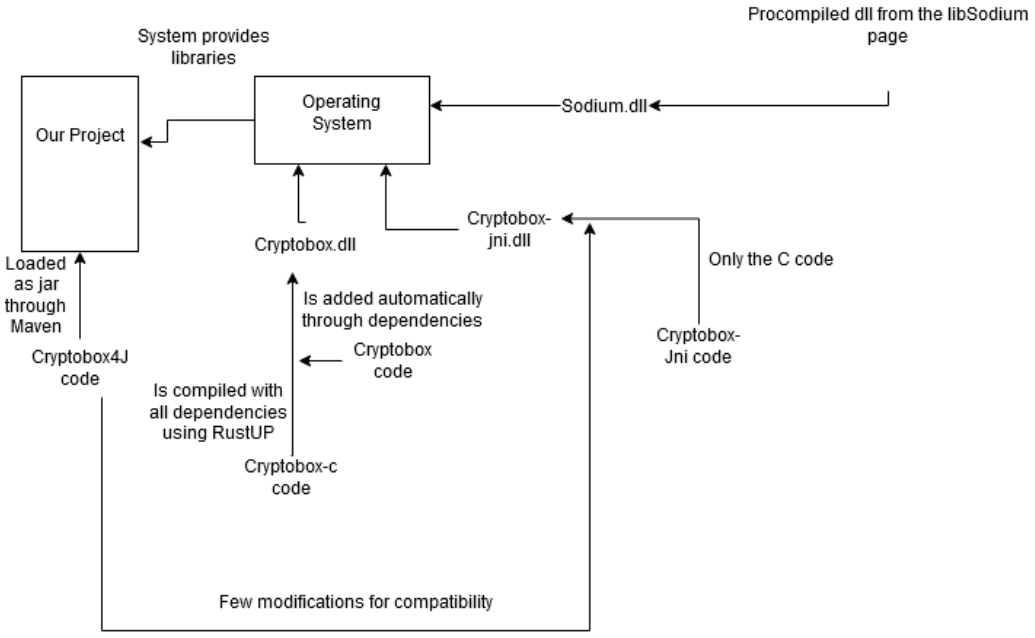

Figure 5: Detailed Cryptobox dependencies[27][31][28][11]

that should be added to the project[2]. When libraries are added this way, every developer has to add the library individually in his IDE.

This can be simplified with maven, because if a dependency is added to the pom.xml file and maven is executed, the dependency is added to the project in a way that the IDE recognizes it.

The pom.xml file can then be added to the project files that are shared between developers, which makes it only necessary for one person to add a dependency and then update the pom.xml file for others to use.

pom.xml file

The pom.xml file we use for maven contains plugins to compile the project into two runnable jars. One that requires the libraries we use and one that has them added inside it.
To be able to compile a jar with dependencies, we had to add the maven-dependency-plugin, which saves all dependencies we require. We also added a plugin that creates a class from a .proto file (more information in the Waz-Messages section).

We added one dependency that is required for the cryptobox and one we use as a json-parser, because of the many json-Strings we get from HTTP requests. The last thing in our pom.xml file is a repository for a java cryptobox version.

4.7.3 Waz-Messages

The generic messages we use to send and receive messages from Wire are added with a proto-file. This is because we copied the files that use the generic messages from lithium[13].
Lithium does not implement the generic messages, but uses a library that adds them. This library is imported with the help of a proto-file and a maven plugin that executes it.

Since we required the same library and wanted to implement it with maven as well, we decided to do it with the proto-file that lithium uses, instead of adding the library through the IDE.

5 Analysis of Messenger-Service-Providers

5.1 Wire

5.1.1 API Documentation

Our Experience with the documentation of the Wire API was very diverse. Often, as it was the case with the cryptobox[27], the documentation was very sparse and we had to rely on comments in the code to understand its functionality. On the other hand the Wire support actually forwarded our questions to someone who was a part of the development team of the official client.
Staying in contact with a Wire developer and being able to ask someone who already implemented the functionality we needed really pushed our progress. The HTTP requests were well documented in a list of possible operations[23], even though we spotted and reported a few little mistakes. The Wire Security Whitepaper[16] also proofed to be a great help in understanding the concept of Wire end to end encryption.

But the way we obtained most information about how to use the endpoints we need, was by analyzing official repositories on GitHub, like lithium[13], xenon[14] and helium[12].

5.1.2 Security

While using the Wire API, we learned a lot about the security of this messenger-service. There are multiple security features that make the transport of messages and the login process safe.

User verification

To login a user, first a login request needs to be send to the server. This results in a response that contains a bearer token and a cookie[4].

The cookie can only be used for the `/access` endpoint, which returns a new bearer token if the old one has expired. To ensure the security of an account, it is possible to revoke all cookies, which requires a new login for each client[23].

A bearer token provides access to all other endpoints of the API. Since these tokens are only valid for 15 minutes, they do not create a big security issue like the cookie[4].

Text message transport

To find out how Wire prevents unauthorized access to private messages, we first searched for endpoints that could be used to send or receive messages. The only ones we found were `/conversations/cnv/otr/messages` and `/notifications`[23]. We also read the Wire Security Whitepaper[16], to find out how prekeys work and how they should be used.

The `/conversations` endpoint is used to send encrypted messages to another user. To ensure that only the recipient can read the message, Wire uses prekeys. Sending a message to another user requires the sender to obtain a prekey from each client of the recipient. The message is then encrypted individually for each client.

The `/notifications` endpoint is used to receive notifications, which sometimes contain encrypted messages. These messages can be decrypted with the private part of the prekey that was used to encrypt the message[16]. Because the private part of the prekey is never uploaded and each client that wants to send a message to the recipient acquires a new prekey, we think that this system is very secure.

File message transport

To analyze the transport of files in Wire, we started by searching for endpoints that might be able to send files. The only endpoint which seemed to be useful was the `/conversations/cnv/otr/messages` endpoint, which we already used for sending text messages[23].

To send a text message we had to create a generic message that we then encrypt with a prekey and send it to the recipient. At this point we only used generic text messages, which can not be used to send files. Therefor we searched in the code of other Wire java applications, to find out how they send files[13]. There we learned that the generic message has variants for each possible message type, for example texts, files and pings.

The code from lithium also contained valuable information about how files are sent, because they are not sent directly to the recipient, which would require a lot of bandwidth and encryption time for the sender.

Instead of sending a message with the encrypted file to each recipient, the sender uploads the file to a Wire server by using the `/assets` endpoint. The response of the server contains a key that can be used to allocate the uploaded file and a key that is required to decrypt the file.

The sender then creates generic messages for each recipient, that contains the location of the file and the key to decrypt it, instead of a normal message

text. These messages are then sent to the recipients, which requires less bandwidth than sending the whole file directly to them[13].

To prevent access if someone unauthorized stole the location of the file, Wire offers the user the option to delete the uploaded asset[16][23].

5.1.3 Accessibility

To access some endpoints of the Wire API, you need to use a valid Wire account, which makes it a bit difficult to test certain features of the API.

Wire provides a website[23] which contains a list of all endpoints and the option to test them, which was not possible for us, since the website does not use the official Wire API url to send requests.

5.2 Other Messenger-Services

Since the documentation of the Wire API was very sparse and we had to obtain a large portion of information from the code we analyzed, we did not have the time to analyze other APIs in detail.
We did however look at a few other options like Whatsapp and Telegram without going too deep into the implementation and came to the conclusion that most messenger services are pretty similar in the way they communicate with their backend.

6 Conclusion

Looking at where we started, the Project was pretty successful. Even though we missed our initial goal of creating a client that supports multiple messengers, we learned enough about the inner workings of messenger services to make a meaningful statement about the possibility of a multi messenger.

Judging by how manageable it was for us with little to no prior knowledge about messenger services to implement a working Wire client, the implementation of most open source messenger-services should be well possible.

The Telegram API for example looked very well documented, probably even better than the Wire API. We expect it to be manageable to implement Telegram support into our client in less time than Wire because of the knowledge we earned during this project. The abstraction we used should also help to implement other messengers.

But then there is Whatsapp, which is still the most popular and most commonly used messenger-service in the western world[24].
Facebook, who has bought Whatsapp years ago[8], seems to tolerate the few public open-source Whatsapp clients that exist, but it was not possible for us to find any concrete documentation or reliable information on this subject.
So implementing Whatsapp support in our client is at best much more time consuming than comparable messengers that are already open-source. It is also unknown how well Facebook documents updates and changes to the Whatsapp API and how long it would take to adapt the client to them.

Our UniMessenger provides an open-source base for anyone who is interested in attempting to create their own messenger client, by showing ways to abstract the base features of a messenger-service and basic interaction with online APIs[42].
It also provides a baseline for simplifying the rich set of messenger features to the ones necessary for a basic chat client that is implemented with limited time and resources.
At last we provided a basic functioning client for the Wire service. The client is usable for communication in a CLI environment, which is less resource heavy than the official client.

Figures

References

[1] *Abstraction*. URL: https://techterms.com/definition/abstraction (visited on 07/12/2020).

[2] *Add libraries in IntelliJ*. URL: https://www.jetbrains.com/help/idea/library.html#define-library (visited on 10/12/2020).

[3] ankitjainst. *Json Parsing*. URL: https://www.geeksforgeeks.org/parse-json-java (visited on 07/12/2020).

[4] *Authentication API Documentation from Wire*. URL: https://docs.wire.com/understand/api-client-perspective/authentication.html (visited on 07/12/2020).

[5] *Chrome Developer Tool*. URL: https://developers.google.com/web/tools/chrome-devtools (visited on 11/12/2020).

[6] *Contact definition by Cambridge dictionary*. URL: https://dictionary.cambridge.org/dictionary/english/contact (visited on 11/12/2020).

[7] *Conversation definition by Cambridge dictionary*. URL: https://dictionary.cambridge.org/de/worterbuch/englisch/conversation (visited on 08/12/2020).

[8] Adrian Covert. *Facebook buys WhatsApp for $19 billion*. URL: https://money.cnn.com/2014/02/19/technology/social/facebook-whatsapp/ (visited on 09/12/2020).

[9] *Cypher block chaining on Wikipedia*. URL: https://de.wikipedia.org/wiki/Cipher_Block_Chaining_Mode (visited on 09/12/2020).

[10] Frank Denis. *Libsodium Introduction*. URL: https://doc.libsodium.org/ (visited on 07/12/2020).

[11] dkovacevic. *Cryptobox for Java*. URL: https://github.com/wireapp/cryptobox4j (visited on 07/12/2020).

[12] dkovacevic. *Helium repository*. URL: https://github.com/wireapp/helium (visited on 07/12/2020).

[13] dkovacevic. *Lithium repository*. URL: https://github.com/wireapp/lithium (visited on 07/12/2020).

[14] dkovacevic. *Xenon repository*. URL: https://github.com/wireapp/xenon (visited on 07/12/2020).

[15] *Encrypting and Decrypting Files in Java*. URL: https://www.baeldung.com/java-cipher-input-output-stream (visited on 09/12/2020).

[16] Wire Swiss GmbH. *Wire Whitepaper*. URL: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf (visited on 07/12/2020).

[17] *HTTP request methods*. URL: https://developer.mozilla.org/de/docs/Web/HTTP/Methods (visited on 09/12/2020).

[18] *Interface definition by Cambridge dictionary*. URL: https://dictionary.cambridge.org/dictionary/english/interface (visited on 09/12/2020).

[19] *Interface definition by Oracle*. URL: https://techterms.com/definition/dll (visited on 09/12/2020).

[20] *Interface definition by Oracle*. URL: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html (visited on 09/12/2020).

[21] *Java version of HttpClient*. URL: https://openjdk.java.net/groups/net/httpclient/intro.html (visited on 11/12/2020).

[22] *Java version of URLConnection*. URL: https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html (visited on 10/12/2020).

[23] *List of Wire Operations*. URL: https://staging-nginz-https.zinfra.io/swagger-ui/ (visited on 07/12/2020).

[24] *Market share of messenger-services*. URL: https://www.messengerpeople.com/global-messenger-usage-statistics/#Germany (visited on 09/12/2020).

[25] *Message definition by Cambridge dictionary*. URL: https://dictionary.cambridge.org/de/worterbuch/englisch/message (visited on 08/12/2020).

[26] *Ping definition by Wire*. URL: https://support.wire.com/hc/en-us/articles/204140584-Send-a-ping (visited on 08/12/2020).

[27] romanb. *Cryptobox repository*. URL: https://github.com/wireapp/cryptobox (visited on 07/12/2020).

[28] romanb. *Java-Native-Interface Cryptobox repository*. URL: https://github.com/wireapp/cryptobox-jni (visited on 07/12/2020).

[29] Margaret Rouse. *API endpoint definition by TechTarget*. URL: https://searchapparchitecture.techtarget.com/definition/API-endpoint (visited on 08/12/2020).

[30] *RustUp introduction*. URL: https://rust-lang.github.io/rustup/ (visited on 10/12/2020).

[31] twittner. *C-Cryptobox repository*. URL: https://github.com/wireapp/cryptobox-c (visited on 07/12/2020).

[32] *UniMessenger Conversation class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/wire/structures/WireConversation.java (visited on 11/12/2020).

[33] *UniMessenger ConversationHandler*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/storage/ConversationHandler.java (visited on 11/12/2020).

[34] *UniMessenger Data Interface*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/interfaces/IData.java (visited on 11/12/2020).

[35] *UniMessenger Generic Message*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/resources/proto/messages.proto (visited on 11/12/2020).

[36] *UniMessenger HTTP classes*. URL: https://github.com/Fi0x/UniMessenger/tree/v0.2-prototype/src/main/java/unimessenger/communication (visited on 11/12/2020).

[37] *UniMessenger Interfaces*. URL: https://github.com/Fi0x/UniMessenger/tree/v0.2-prototype/src/main/java/unimessenger/abstraction/interfaces (visited on 11/12/2020).

[38] *UniMessenger Message class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/storage/Message.java (visited on 11/12/2020).

[39] *UniMessenger MessageReceiver class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/interfaces/wire/WireMessageReceiver.java (visited on 11/12/2020).

[40] *UniMessenger MessageSender class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/interfaces/wire/WireMessageSender.java (visited on 11/12/2020).

[41] *UniMessenger MessageSorter class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/interfaces/wire/WireMessageSorter.java (visited on 11/12/2020).

[42] *UniMessenger on GitHub*. URL: https://github.com/Fi0x/UniMessenger/tree/v0.2-prototype (visited on 09/12/2020).

[43] *UniMessenger Ping Message*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/abstraction/wire/messages/Ping.java (visited on 11/12/2020).

[44] *UniMessenger Updater class*. URL: https://github.com/Fi0x/UniMessenger/blob/v0.2-prototype/src/main/java/unimessenger/util/Updater.java (visited on 11/12/2020).

[45] *Wire on GitHub*. URL: https://github.com/wireapp (visited on 08/12/2020).

[46] *Wire Web-Client*. URL: https://app.wire.com/ (visited on 11/12/2020).

[47] *Worldwide 'OS' market share*. URL: https://gs.statcounter.com/os-market-share (visited on 08/12/2020).