Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Convenient Password Manager

Bachelor's Thesis

Noé Heim

noheim@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Simon Tanner, Roland Schmid
Prof. Dr. Roger Wattenhofer

June 8, 2020

Acknowledgements

I would like to thank Simon Tanner and Roland Schmid, for always offering their support, when I was faced with difficulties, and guiding me through my very first scientific research project. Further, I would like to thank Prof. Dr. Roger Wattenhofer for encouraging me with his enthusiasm for this project.

I'm grateful that I had the opportunity to work with and learn from so knowledgeable people.

Lastly, I would like to thank my family and friends, who supported me and motivated me as I approached the end of this thesis.

Abstract

Password managers are a useful aid for remembering and securely storing passwords. As it is crucial to provide adequate security given the sensitive data they contain, many password managers offer two-factor and multi-factor authentication. However, most of them use a single database which can introduce security risks. In this thesis, we attempt to extend PolyPass, a distributed password manager, with Bluetooth communication. Our goal is to increase the reliability of the password manager, as well as the ease of access to passwords. To achieve said goal, the communication had to be restructured. The Bluetooth communication is still problematic, since their messages are not reliably delivered, which renders the password manager unusable when not connected to the internet.

Contents

Chapter 1

Introduction

1.1 Motivation

With the amount of digital systems that affect virtually every aspect of our lives, security is a key concern. To protect our data, there exists a multitude of possibilities, many of them dealing with security by personal authentication, namely via passwords.
Given the vast amount of passwords [1] that people have to deal with nowadays, the organization of the different passwords needed can be difficult. To mitigate this problem, password managers can serve as a helpful tool. They store passwords either locally, on a server or distributed over multiple devices. Once a password is requested by a user he has to prove that he has the right to access it. What this proof entails varies a lot. Some password managers require a master password. Others require the interaction with multiple devices. This kind of authentication is called multi-factor authentication. Since password managers contain crucial data, they must be accessible both quickly and securely by its respective user.

1.2 Goal

Thus, the objective of this work is to extend PolyPass, a distributed password manager that uses multi-factor authentication. PolyPass itself is built on NoKey [2]. In contrast to other password managers, the need for a connection to a server is avoided and the option of communicating over Bluetooth is added. This way, we aim to make communication more reliable. The password manager would still be usable, even if the server, which is used to communicate between the devices, cannot be reached.

Chapter 2

Related Work

2.1 Password managers with multiple-factor authentication

There are many different password managers which use multi-factor authentication on the market, but the principle is always the same: A user has to provide some form of proof that he has the right to access the passwords. As an example, 1Password[1] uses external authentication software, either Authy[2] or Microsoft Authenticator[3], to grant access to a user's passwords. In additon to these methods of authentication, RoboForm[4] allows Google Authenticator[5] to be used. These authentication apps require additional interaction with a user's smartphone. These authenticator apps are a popular solution, since other password managers such as Bitwarden, Keeper Security and LastPass offer the possibility to use two factor authentication based on these authenticator apps. Two of these password managers, 1Password and Roboform, were analyzed [3]: At the time, that paper was written, these password managers were vulnerable to attacks on their database. Using distributed password manager mitigates this problem due to the absence of a database storing the passwords in a single place.

There exists a set of specifications called FIDO2[6]. These specfications are the foundation for standard web APIs, which can be used for two-factor authentication. YubiKey[7] is one example which implements passwordless authentication based on FIDO2.

[1] https://1password.com/
[2] https://authy.com/
[3] https://docs.microsoft.com/en-us/azure/active-directory/user-help/user-help-auth-app-download-install
[4] https://www.roboform.com/
[5] https://support.google.com/accounts/answer/1066447?co=GENIE.Platform%3DAndroid&hl=en
[6] https://fidoalliance.org/fido2/
[7] https://www.yubico.com/

2.2 Bluetooth in past projects

Additionally, there also exists research aiming to understand the Bluetooth architecture of Android by designing a Bluetooth chat [4]. It describes the design of the connection management, how devices communicate and explains the Bluetooth architecture of Android.

Related research [5] discusses about a project which used Bluetooth devices to lock and unlock a home door for disabled people. In that project, a user's Smartphone is used to establish a connection to an Arduino controller board which then can be controlled remotely.

Chapter 3

Background

The goal of this work is to expand the existing password manager PolyPass [6] into a more reliable, but equally convenient password manager. We attempt this by adding communication over Bluetooth to the already present communication via WebRTC. This chapter provides necessary background information for this project.
We start by explaining the concept of "PolyPass". This follows a general overview of Bluetooth Communication and the characteristics of the Bluetooth Protocol RFCOMM and GATT, including the libraries we used, i.e. PyBluez and Android's Bluetooth API. This will be followed by an explanation of Chrome's Native Messaging Host and Androids Message Handlers.

3.1 Previous Project

This thesis is building on PolyPass, a password manager using multi-factor authentication. Passwords can be accessed only after the user has interacted with at least two devices. This means the user has to have at least two devices with PolyPass installed. This need for multiple devices resulted in the application being developed as a browser extension and Android app, to give the users more freedom in the type of devices they can use. Passwords are stored in a distributed fashion, by using Shamir's Secret Sharing [7]. This way passwords can be split up into different shares. A subset of k shares, dependent on the security level of the password, can be used to generate the original password. But to get enough of these shares, messages containing them have to be exchanged. The messages are exchanged using WebRTC [8]. It an open-source communication protocol providing peer-to-peer communication, which can be accessed by APIs. These APIs are available on many browsers. This means there is a server running, which is responsible for the connection setup, between two devices. To be able to establish a WebRTC connection with the right device, PolyPass uses IDs. They have the same format as uuids (we will refer to them as uuids from now on) used by Bluetooth service advertisements, which will prove to be useful later on. The

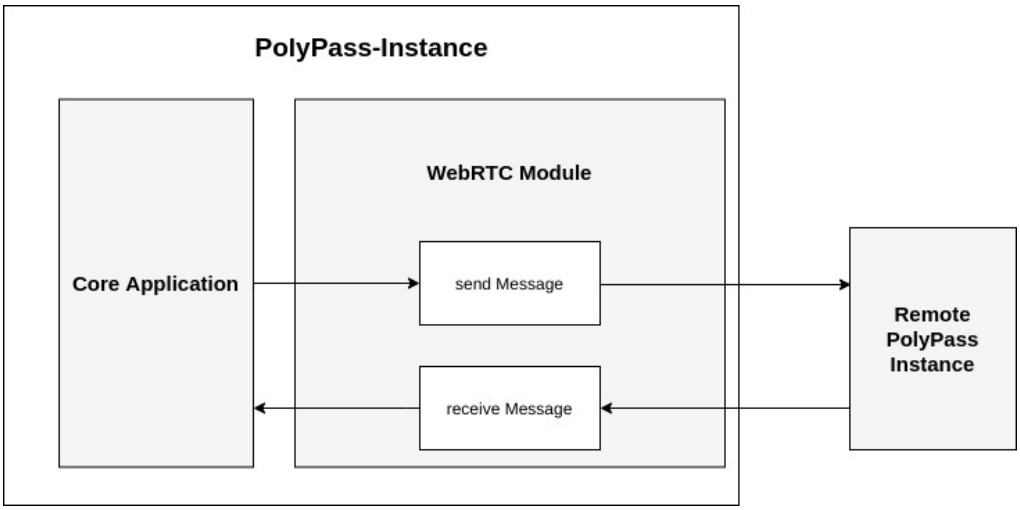

Figure 3.1: Concept of Communication: PolyPass

device initiating the connection requests the connection parameters of the other device. If the other device accepts the connection, the devices can communicate directly with each other. This allows the possibility to detect whether the devices are in the same local network or not.
When communicating with another instance, the concept for the communication can be simplified to the depiction in Figure 3.1. The WebRTC Module, as illustrated in Figure 3.1, is responsible for the communication between devices over WebRTC. It waits for messages coming from the Elm part of the application (which we will refer to as core application). Then it sends these messages to the correct remote device, if possible. If that is not possible it stores only the latest message, to resend it later, which results in not all messages being delivered. Further more, the module waits for incoming messages from remote devices and once received passes them back to the core application. There the message is then processed. The application was mostly written in Elm[1], as well as JavaScript[2]. Elm is a functional programming language, which compiles to JavaScript. Additionally, to be able to run the compiled Elm on Android and use some of Android's background services, some parts had to be written in Kotlin[3].

3.2 Bluetooth Communication

In order to introduce a a Bluetooth connection method, we looked at different possibilities of Bluetooth protocols. In order to introduce a Bluetooth connection

[1]https://elm-lang.org/
[2]https://www.javascript.com/
[3]https://developer.android.com/kotlin

method, background on the Bluetooth stack is needed. This section starts with an overview of the Bluetooth stack, followed by a more in-depth description of the Android's API and Bluez [4], the official Bluetooth stack for Linux.

3.2.1 General

Bluetooth is a wireless technology to communicate between devices in close range. The Bluetooth stack consists of several layers [9]. This work is mainly operating on the higher layers. The higher layers include of L2CAP (logical link and adaptation protocol) and HCI control. The latter offering an interface to the user, to manipulate the Link manager. L2CAP can be seen as the link layer for Bluetooth. L2CAP is followed by the Data layer which provides an interface to SDP (service discovery protocol) and RFCOMM. SDP is used to discover the services the host machine provides, and these services can be used to then connect to RFCOMM, which is the basis for point-to-point links. These higher layers are usually implemented in software. RFCOMM is a protocol for reliable data transfer, by configuring the underlying protocol, L2CAP, accordingly. Reliable data transfer here means guarantee, that all data is delivered in order, without any duplicate data arriving. On top of all that is the application layer.
For Bluetooth Low Energy, the protocol stack is mostly the same up to and including the L2CAP layer. Instead of the data layer now follow two seperate parts: the SMP (Security Management Protocol) and the Attribute Protocol (ATT). ATT forms the base for GATT (Generic Attribute Profile). [10] We focused on two of these protocols, namely GATT and RFCOMM. GATT (Generic Attribute Protocol) is a protocol where devices can be in two different roles: peripheral and central mode. When a device is in the peripheral mode, connection to only a single central device can be made at a time, whereas a device in central mode can connect to multiple devices. According to the specification ([5]p. 283 "6.4.1 Attribute Protocol") can be be in central and peripheral mode at the same time. This possibility is not equally well supported for the different operating systems though. GATT, similar to TCP, offers reliable read and write operations. But, these operations limit the the amount of data that can be sent [11].

3.2.2 Android's Implementation

Android's Bluetooth API offers three different types of connections between devices: GATT, RFCOMM and L2CAP. The functions are fairly well documented, with several examples online, which are mainly for Android-to-Android communication. Android provides insecure RFCOMM connections allowing to avoid pairing. These connections are encrypted, but suffer from possible man-in-the-

[4]http://www.bluez.org/
[5]https://www.bluetooth.com/specifications/bluetooth-core-specification/

middle attack, due to unauthenticated link keys. To setup a connection SDP lookup is used[6].

3.2.3 Bluez and PyBluez

Bluez, the official Linux stack, provides several modules, which include CLI utilities to test and configure Bluetooth devices. We used these tools to gather information and test ideas on the Linux side. Additionally it is also the basis for Bluetooth libraries, such as PyBluez, a Python module. PyBluez offers the possibility to access the machine's Bluetooth resources, while being easier understandable than the tools offered by Bluez. It also provides the possibility to use RFCOMM and L2CAP, as well as experimental support for GATT.

3.3 Local Communication

Since the Bluetooth functionality cannot be accessed directly by JavaScript, separate Bluetooth services had to be implemented. The Bluetooth services are running in parallel to the core application. Therefore, the ability to communicate between these services and the core application is important. Since Android and the Chrome extensions on Linux have different implementations of these Bluetooth services two different approaches are required to implement the local communication.

3.3.1 Chrome: Native Messaging Host

Chrome offers an API for extensions, called Native Messaging Host, to communicate with other applications running on the host machine. On the extension side it uses a port object to receive and send data in the form of JSON objects. On the host application side data is received over the standard input stream and sent over the standard output stream. But in order for this communication to work, the host application has to be registered in the chrome configuration by adding a manifest file to the configuration folder [12]. Other browsers offer similar APIs for local communication, which makes porting this functionality to other browsers easier.

3.3.2 Android

In general, Android applications are composed of so called Services and Activities. Services are long running tasks, which can be separated from the GUI and Activities are foreground tasks mostly requiring the interaction of the user. There

[6] https://developer.android.com/reference/android/bluetooth/BluetoothDevice#createInsecureRfcommSocketToSer

are several types of services, but bound services [13] are relevant for implementing local communication, as they offer an interface for components to interact with them. This interaction entails sending requests and receiving results for the requests. To communicate from a service to the component which created it, messengers can be passed to the service. Messengers are references to handlers in another component, which then can be used to send messages to it and the receiving component will handle the received message.

Chapter 4

Implementation

The base for this thesis is laid by the Password Manager "PolyPass", which relies on a server to establish the WebRTC connection. But what happens if the server cannot be reached? We try to offer a solution for this problem by integrating Bluetooth communication. Therefore, it was necessary to redesign the communication part of "PolyPass". The goal is to create a transparent communication module. This desired transparency leads to separating the communication from the core application.
In this chapter we will explain thoroughly how we planned to achieve the decoupling of the communication and core application in terms of software design.

4.1 Concept

Since our goal was to add another communication channel, namely Bluetooth, we had to redesign how communication is handled. First we needed to know how to communicate over Bluetooth. This meant testing various protocols and figure out their limitations. On an abstract level we needed, additionally to the WebRTC communication, to be able to send and receive messages over Bluetooth and some way to manage the Bluetooth connectable devices. Sending the same messages over Bluetooth and WebRTC would result in requesting the same user interactions multiple times. To avoid this, we needed to introduce a communication handler (Figure 4.1), which manages the communication. It has to keep track of received and sent messages. Additionally, it needs to send the new messages and re-send the ones which could not be transmitted. Managing messages should be done by keeping two queues, one for the messages, which still have to be sent, and one for the received messages. To realize that, we decided to define two message types, acknowledgement messages and standard messages (abbreviated by Std-messages). The Std-messages contain the messages, which are sent from the core application to the Communication Handler, as payload. Additionally they include the uuid of the target device, a message type, a sequence number and the sending device's uuid. The sequence number is used to identify the messages, which allows acknowledging a specific message. The acknowledgement messages

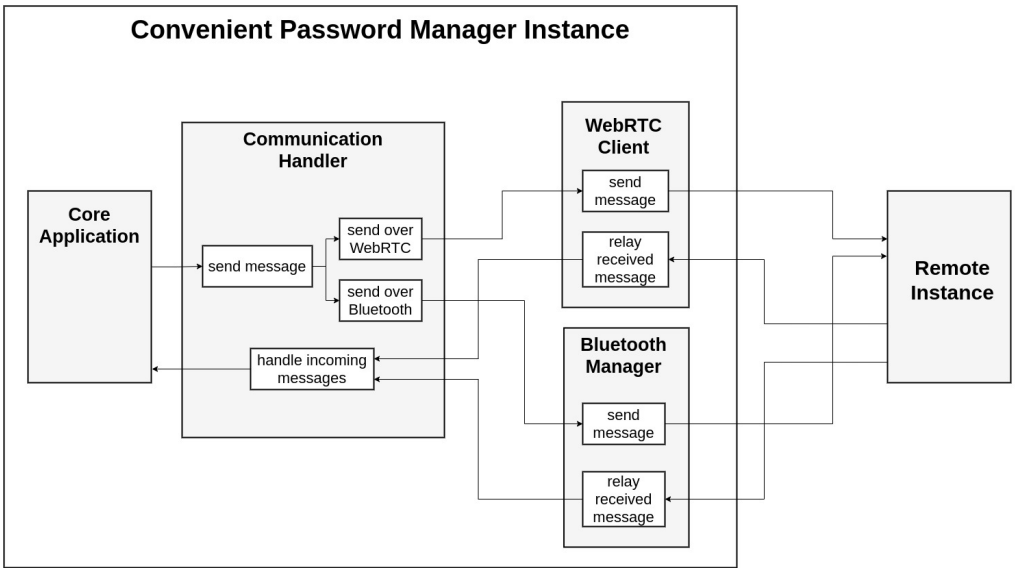

Figure 4.1: Concept of Communication: Convenient Password Manager

are used to control which messages have to be resent, by signaling to the sender that a message has arrived.

To interfere as little as possible with the Elm-code, it was decided to use encapsulation (similar to networking protocols). The structure of the two messages, including how encapsulation is accomplished, is further illustrated in Figure 4.2.

Since we wanted to keep the control on which messages have to be resent for both channels, we decided to not have receive- and sendqueues implemented per communication channel, but rather one step before that, in the "Communication

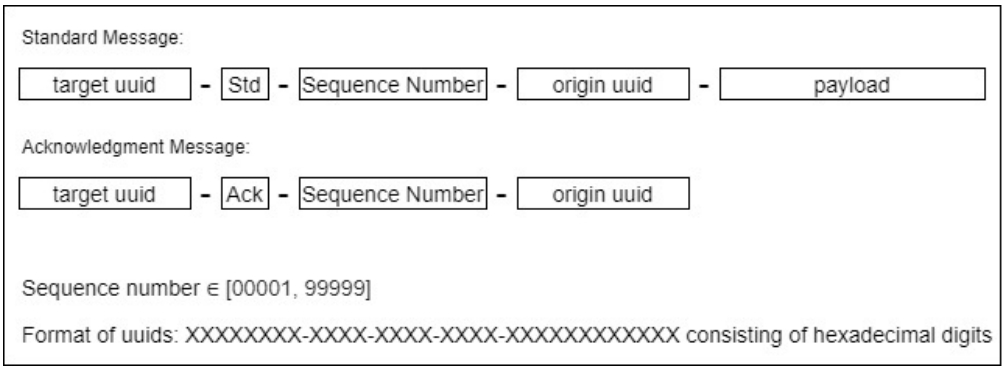

Figure 4.2: Concept of Communication: Encapsulating Messages and Acknowledgments

Handler". Even though this causes more messages being passed locally, this allowed us to decide which messages should be sent over which channel and saves messages being sent unnecessarily often. As an example, assume there are two devices, A and B. A tries to send a message over Bluetooth and WebRTC to B, but B is only reachable over WebRTC. In the design chosen, A attempts to send the message over both channels. As soon as the message reaches B, B sends an acknowledgement back to A. This in turn means A does not need to resend the message over any channel and can just continue with sending other messages. Assume now the sending queue would be implemented in the Bluetooth and WebRTC modules directly. The initial message would be sent over WebRTC and Bluetooth, and its acknowledgment would reach the WebRTC module. Unless we had additional local communication between the modules, A's Bluetooth module would try to resend the initial message indefinitely, which would use additional resources, especially for scanning Bluetooth devices, which consumes a lot of power.
Due to the just described scenario and the assumption that most of the time at least one mobile device is involved, saving power was more an important factor in the design choice. Therefore, keeping track of the messages in the communication handler was logical. By not implementing a queue on both, the WebRTC- and Bluetooth-module, additional communication between the two can be avoided, which would make makes the design easier to understand and less error-prone.

4.2 Bluetooth Communication

Before we can communicate over Bluetooth, we first need to establish the connection. A connection involves a client and a server and is initiated by the client. To accomplish a successful connection setup, we decided to use the uuids used by the core application (as described in Section 3.1) to advertise a Bluetooth service. This advertisement informs scanning devices, what service is offered by the this service, which means if the uuid we want to connect to is present on this device. Since this functionality is needed for Linux as well as Android, it is advertised using the platform specific advertising functions. When the local Bluetooth device is listening on the RFCOMM channels, we can simply connect to them which will give us sockets on both sides, client and server, providing the possibility to send data to each other. Once the data is sent, we can close the socket and wait for new messages to send.

4.2.1 Android Side

Since Android provides good documentation, writing the Bluetooth communication for Android was fairly straight forward. Because we want the Bluetooth module running in the background, we decided to implement it using Android

Services. The service we implemented is started and bound, because we want it to be running, even if the main application temporarily disconnects from the service. But since we also want to interact with the service from the main activity, we additionally bind the service (as described in Section 3.3.2). This enables us to reference the service instance and call its member functions, which we want to use for communication from the activity to the service. To communicate from the service to the activity we use a "Messenger", which allows us to send the received messages back to the main activity.
The implemented service for the managing the Bluetooth communication, in addition to offering the communication interface for the main activity, also sets up a controller managing the Bluetooth server instances. It also creates a Bluetooth client thread, when a message from the main activity should be sent using Bluetooth. We will now go into the servers and clients more in-depth.

The controller and Bluetooth server

The controller for Bluetooth server is responsible for listening for incoming connection requests. Further more, once a client wants to connect to this device, the controller accepts the connection and creates a Bluetooth server thread. This newly created server reads the data, which was transmitted by the remote client, from the input stream. Then it passes the message back to the Bluetooth service, which sends the received data back to the main activity. Additionally, the Bluetooth server creates a global mapping from the remote client's uuid to the actual remote device. As a result, this avoids the need to scan for Bluetooth devices, when a message is sent out to this uuid again. After reading all the data received, the Bluetooth server closes the open streams of the socket and the socket itself.

Bluetooth client

As stated before, a Bluetooth client thread is created once the Bluetooth service is told to send a message. When creating it, we pass the message and target device to it and we start the client's sending function. Once started, the Bluetooth client will try create an RFCOMM socket to the service, which is specified by the uuid. If it can connect to the other end, it writes the message to the output stream of the RFCOMM socket and closes the RFCOMM socket after closing the associated streams. Once the streams are closed, the thread terminates.

4.2.2 PyBluez and Native Messaging

When we implemented the Bluetooth-module for Linux using PyBluez, we followed a similar pattern to the implementation in Android. We have a connection manager in place, which initializes the Bluetooth-server and Client. But first, to

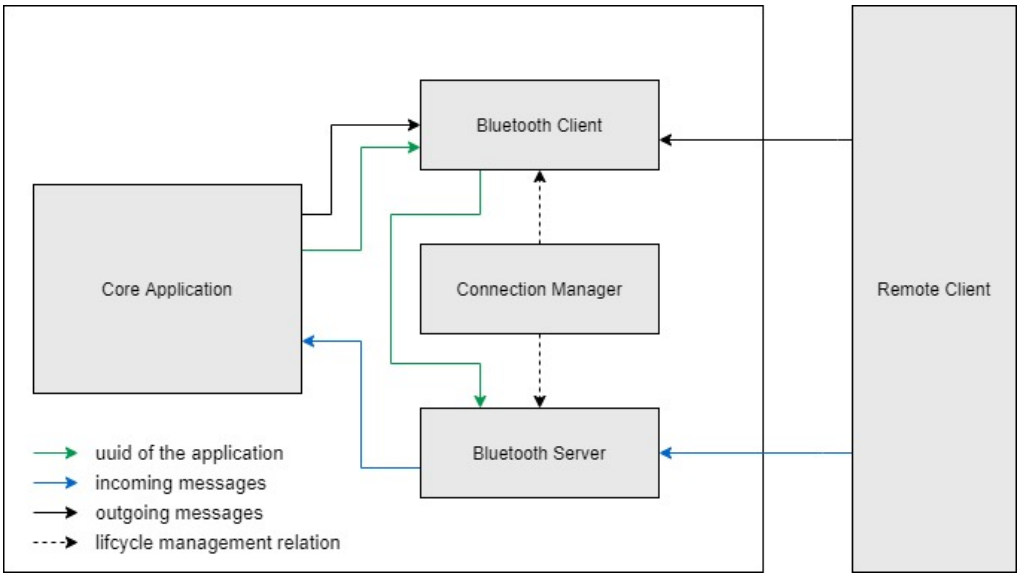

Figure 4.3: Concept of Communication: Convenient Password Manager

run it in combination with the extension, we needed to add some communication from the Bluetooth-module to the extension. Using Chrome's Native Messaging Host API, setting this communication up was simplified immensely. On the extension side we just need to connect to this Native Messaging Host, using its name. This in turn will execute a script defined in a manifest and start the Bluetooth-module. Then we can start communication in both directions, from the extension to the Bluetooth module and vice versa, using the Native Messaging Host API.
On the PyBluez side to send data we write to the standard output stream. To receive data we need to read from the standard input stream. Following a similar design as with the Bluetooth service, we created a connection manager which handles the creation and termination of the client and server threads at the beginning and end.

Client

Once the client is created it waits for a message from the extension containing the device's specific uuid, which will be passed to the server thread. Once this message is received, we continue to wait for messages from the extension. On receiving a message from the extension, the destination uuid is extracted and we check, if our uuid-to-MAC-addresses map contains the uuid. If it does contain the destination uuid, we continue directly with connecting to the MAC-address and sending the message. If the destination uuid is not present in the previously mentioned map, we start scanning for devices. For all the devices, for which

we receive a scan result, we lookup the services advertised by the device. As soon as we find a service with a uuid matching the one we are looking for, we return a "Peer" object, which holds the data needed to create a connection to the remote device. After finding the correct "Peer" object, we use the built-in socket functions to connect to the remote device, send the data and close the just created socket. After that, the client returns to waiting for messages from the extension and repeats this process.

Server

After the server is started, it waits for the uuid, which it will receive from the client. Once the uuid has been received, it is used, on the server thread to advertise a service. After that the server will wait for remote-clients, which want to connect to it. Once it receives a connection request, the server will accept the connection and return a socket. The server will read the data sent by the client from this socket. After reading all the data successfully the server closes the client socket. It then sends the received data to the extension, using stdout stream and correct formatting. After that the server returns to waiting on clients which want to connect to it and repeat this process.

4.3 Overall Communication

When we implemented the Communication Handler, there were a few things which had to be done. First an interface for the Bluetooth module had to be provided. Additionally sending the acknowledgement messages had to be implemented. Then incoming messages, depending on the type of message, needed to be handled appropriately. Finally, keeping track of the sent and received messages had to be set up. We will look at these four steps more in-depth in the following subsections.

4.3.1 Interface for Bluetooth Module

For the Bluetooth modules to be of any use, we needed to provide an interface for the Bluetooth modules. Since we are using webviews in Android, this was an easy task on the android side. We just needed to create a function, which is dedicated for the use on Android devices only. This function can be called from the Kotlin code directly. Additionally a function provided by the Android app is needed. This function can then be called from the JavaScript code and is executed in the Android app itself. With that providing the interface for the Android side was done.
For the applications running on a Linux machine, the process was similar but not exactly the same. Instead of functions which can be directly called, the Native

Messaging Host API provides a port in the extension. Data can be sent to and received from a native messaging host via this port. On the other hand, the native messaging host can use the standard input and output streams to send and receive messages.

4.3.2 Acknowledgment Messages

Every time a standard message is received by the communication handler an acknowledgement message is sent back. Acknowledgment messages are formatted as described by the acknowledgment message in Figure 4.2. After generating the message, we send it back to the other device over Bluetooth and WebRTC, without adding it to the messages we keep track of.

4.3.3 Handling Incoming messages

Regardless where the app is running, we then handle the message received by processing it. First, is determined, if it was an actual message from another device. In that case we check the type of the message. For standard messages we proceed by first sending the acknowledgement message, then continue with checking, if the message was already received. If it was was already received we do nothing further. For message not yet received, we send the payload, as defined in Figure 4.2, to the core application. If the message received is an acknowledgement package, we update the queue of unacknowledged messages and return. How the updating is actually done will be described in a following section.

4.3.4 Tracking Unacknowledged and Received Messages

The communication handler uses two Maps. One is used to associate remote device uuids with their queues of unacknowledged messages. The other maps remote device uuids to queues of sequence numbers. Additionally it offers functions to manipulate these queues, allowing to add unacknowledged messages and remove acknowledged ones, including the respective sequence numbers.
Received messages are tracked by saving all the sequence numbers of received standard messages. This is a necessary step to determine which messages still have to be passed on to the core application. Otherwise the received messages could trigger the same interaction, with the same effect, twice.

Chapter 5

Evaluation

5.1 Reliable communication over WebRTC

As a result of implementing the communication handler, messages which have not been delivered are stored in a queue of unacknowledged messages within the communication handler. This achieves a more reliable communication over WebRTC, because compared to the original PolyPass, only the last undelivered message was resent. Now though, all messages to a target uuid which could not be delivered are being resent every time a new message is sent to that uuid.

5.2 Bluetooth communication

In the Bluetooth module, implemented for Linux using PyBluez, scanning for Android devices works. The service with the device specific uuid gets advertised correctly from Android devices. On Android though, finding services which should be advertised by PyBluez does not work, but it is unclear if Android cannot find services with custom uuids or if PyBluez does not advertise services correctly. Since the Bluetooth module on Linux can find Android devices and send messages to it, this issue is partially avoided. The receiving device can create a mapping from device specific uuids to the sending device. This mapping is created using the information provided by the connection and its associated MAC address. Once a message is sent from a Linux device to an Android device, these two can communicate by using the mapping from uuids to devices to send messages. Depending on the cause for the issue of scanning for Linux devices on Android, the issue might persist, when two Linux devices or two Android devices try to communicate with each other over Bluetooth. During this thesis, it was not possible to investigate this issue further.
Additionally when there are messages from two devices sent at roughly the same time, so that the period of one device sending and receiving overlap, the error "connection reset by peer" occurs. This error occurs so often, because after every new message, which is supposed to be sent, we try to resend unacknowledged messages via Bluetooth and WebRTC. Furthermore the Bluetooth module can-

not process and send the messages, which it receives fast enough. This leads to congestion of messages, which are supposed to be sent. This leads to so much traffic over Bluetooth, that acknowledgement messages can no longer be received, causing a continuously growing queue of unacknowledged messages. This issue renders the application unusable, when there is no internet connection available. Once the application can reconnect though, all the messages which could not be delivered, will be resent over WebRTC.
As part of this thesis the cause of this error could not be investigated any further. Further more, periodic re-sending of unacknowledged messages could not be implemented due to time restrictions.

5.3 System integration

The communication over WebRTC works flawlessly. Messages get resent, if they could not be delivered. If Bluetooth messages arrive, they are handled correctly as well. These findings together imply that, if the Bluetooth communication was reliable and fast enough the goal of adding Bluetooth as communication method would be achieved.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis PolyPass was extended by adding Bluetooth communication. We chose RFCOMM as transport protocol allowing us to send and receive data in the form of messages. To implement this, we used PyBluez and Android's Bluetooth API. To implement this feature a communication handler was needed. It manages the sending and receiving of messages. Additionally, it tracks which messages have not yet been unacknowledged. Further more, it is responsible for only passing messages to the core application, which were not already received.
The time it took to find a suitable method for communicating over Bluetooth was underestimated. This resulted in too little time to fix the problem of Bluetooth messages not arriving, when there are too many messages to be sent. As a result, the password manager is not reliable enough to be used.
Another unsolved problem is, that the Android app still relies on the server. The core application is not delivered with the app itself. It is loaded as a web app, which has to be retrieved from the server responsible for the WebRTC communication setup. This issue could be solved by loading it from local storage.

6.2 Future work

To make this application usable, the problem of messages not arriving quickly enough should be fixed. Additionally, finding a solution for the advertised Bluetooth services not being discovered is needed to make the application usable.
Furthermore, the application's security could be increased, by protecting against man-in-the-middle attacks on the Bluetooth communication.

Bibliography

[1] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Crano, "Why people (don't) use password managers effectively." Santa Clara, CA, USA: USENIX Symposium on Usable Privacy and Security (SOUPS), 2019. [Online]. Available: https://www.archive.ece.cmu.edu/~lbauer/papers/2019/soups2019-pwd-mgrs.pdf

[2] F. Zinggeler, "Nokey - a distributed password manager," 2018. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2017-HS/MA-2017-24.pdf

[3] P. Gasti and K. B. Rasmussen, "On the security of password manager database formats," in *Foresti S., Yung M., Martinelli F. (eds) Computer Security – ESORICS 2012. ESORICS 2012. Lecture Notes in Computer Science*, vol. 7459. Berlin, Heidelberg: Springer, 2012.

[4] W. Pan, F. Luo, and L. Xu, "Research and design of chatting room system based on android bluetooth," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, 2012. [Online]. Available: http://pgembeddedsystems.com/securelogin/upload/project/IEEE/1/pg2012-2013e141/32%20Research%20and%20design%20of%20chatting%20room%20system%20based%20on%20Android%20Bluetooth.pdf

[5] M. E. Safi and E. I. Abbas, "Android-based home door locks application via bluetooth for disabled people," in *Proceedings of the International Conference on control System Computing and Engineering Penang IEEE*, November 2014, pp. 191–195. [Online]. Available: http://eprints.uthm.edu.my/id/eprint/6567/1/Android-based_Home_Door_Locks.pdf

[6] N. Studach, "Polypass - a convenient password manager," 2019. [Online]. Available: https://pub.tik.ee.ethz.ch/students/2019-FS/MA-2019-04.pdf

[7] A. Shamir, "How to share a secret," in *Communications of the ACM*, vol. 22, no. 11, 1979, pp. 612–613.

[8] "Webrtc 1.0: Real-time communication between browsers," 2019 [Acessed June 6, 2020]. [Online]. Available: https://www.w3.org/TR/2019/CR-webrtc-20191213/

[9] P. Bhagwat, "Bluetooth: Technology for short-range wireless apps," 2001. [Online]. Available: http://home.engineering.iastate.edu/~morris/543/paper/bluetooth.pdf

[10] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," 2012. [Online]. Available: https://www.mdpi.com/1424-8220/12/9/11734/pdf

[11] "Bluetooth core specification: Version 5.2." Bluetooth SIG. [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification/

[12] Chrome, "Native messaging." [Online]. Available: https://developer.chrome.com/extensions/nativeMessaging

[13] Google, "Bound services overview," 2019 [Acessed June 6, 2020]. [Online]. Available: https://developer.android.com/guide/components/bound-services