Hochschule RheinMain
Fachbereich Design Informatik Medien
Master-Studiengang Informatik – Smarte Systeme für Mensch und Technik

Master-Thesis
zur Erlangung des akademischen Grades
Master of Science – M. Sc.

Low-Power Matrix Client for Microcontrollers

Vorgelegt von: Patrick Schönberger
Matrikelnummer: 1001314

am 14. November 2023

Referent: Prof. Dr. Martin Gergeleit
Korreferent: Prof. Dr. Marc Stöttinger

Erklärung gem. BBPO 4.1.5.4 (3)

Ich versichere, dass ich die Master-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum Unterschrift Studierende/Studierender

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Arbeit:

Verbreitungsform	**ja**	**nein**
Veröffentlichung des Titels der Arbeit im Internet		
Veröffentlichung der Arbeit im Internet		

Ort, Datum Unterschrift Studierende/Studierender

Abstract

Matrix is an openly developed protocol for decentralized communication over the internet. It can be used to communicate using end-to-end encryption while retaining complete control over one's data. The specification defines exactly how clients and servers interact, and can be used to implement custom client and server software. In this thesis, we will implement a client library that can be used for encrypted communication on both desktop operating systems and embedded devices. This will be tested on an ESP32 MCU, in order to enable it to send and receive fully encrypted messages to and from a Matrix server.

Contents

1 Introduction

Matrix is a communication protocol that enables decentralized, secure, and real-time communication across multiple platforms and devices. The protocol is open, meaning its specification is publicly available and anyone can implement their own client or server software by following that specification. While there are already a multitude of clients and libraries in many programming languages to communicate with a Matrix server, there is currently no publicly available implementation written in C that also supports end-to-end encryption. Writing such a library in C would not only allow calling it from other programming languages through an FFI, but could also make it possible to run it on different operating systems as well as embedded devices.

The main goal of this thesis is to determine whether an embedded device, specifically an ESP32, could be used to send encrypted messages using the Matrix protocol. To achieve this, a client library will be developed that implements enough of the Matrix Client-Server API to support verification with other devices, as well as sending and receiving of encrypted messages. This library will have to be compiled and run on an ESP32, which might impose memory limitations on the library and its dependencies. Apart from that, it will also be evaluated how much energy this library will use, since it involves sending HTTPS requests over WiFi, and running several cryptographic routines for encryption and signing locally on the device.

After giving an introduction to the Matrix protocol and the cryptography behind the end-to-end encryption in Chapters 2 and 3, Chapter 4 will explain how each part of the library was implemented. Chapter 5 will detail the different aspects of running the library on the ESP32, and Chapter 6 will evaluate the previously stated goals. Chapter 7 will give a conclusion, including what was and wasn't possible and what could be improved in the future.

2 Matrix Protocol

Matrix[1] is an open communication protocol. It is defined by an overall architecture, describing the entities of the communication network, and a set of APIs that describe how these entities interact. The Matrix.org Foundation is developing this protocol for several reasons, what they call the Matrix Manifesto:

- People should have full control over their own communication.
- People should not be locked into centralized communication silos, but instead be free to pick who they choose to host their communication without limiting who they can reach.
- The ability to converse securely and privately is a basic human right.
- Communication should be available to everyone as a free and open, unencumbered, standard, and global network.

Matrix allows open, secure communication by making the specification and reference implementations publicly available. This means they can be reviewed by anyone so that there is no single entity that has to be trusted. Communication is also decentralized, there is no authoritative server but instead a network of servers that talk to each other to allow network-wide communication. Users can freely choose which server to connect to, or host one themselves for more control, and will still be able to use all the features that Matrix provides — provided that their server implements the specification correctly.

Of course, there are many different communication networks already, and they cannot all be a part of Matrix. But it is possible to write adapters to allow them to communicate in Matrix rooms. This means existing messaging applications/networks can be plugged into the Matrix ecosystem and, in theory, all communication could be joined in a unified way. This is the overarching goal of Matrix, to unify communication under one general protocol so that anyone can message anyone using any application they like, similar to how E-Mail works across all providers and using any mail client. All while having control over their data, without having to share it with any third parties. This means creating a sort of communication matrix which connects all of its parts, hence the name.

2.1 Architecture

The basic building blocks of Matrix are users, devices, rooms, events, and servers. Users have any number of devices, that can send events in rooms or directly to other users or

[1]`https://matrix.org/`

their own devices. All of this is done through a user's homeserver, which is the server that they created their account on. The homeserver communicates with other servers to synchronize their states.

In order to use a Matrix client, one has to create a user account. This account has the form `@localpart:domain`, where domain is the user's homeserver. Every user account is associated with a number of devices.

2.1.1 Devices

A device in Matrix refers to any client that can log in and interact with the API. This could be a desktop or mobile application, but also a web application, which means that multiple Matrix devices can be operated from the same physical device. Every device has a device ID that is unique to the user and an optional device name, which can be chosen freely by the user. Additionally, every device is associated with two sets of cryptographic keys, a Curve25519[2] device key pair which is used for encryption, and an Ed25519[3] signing key pair which is used to authenticate messages sent by the device. Both key pairs consist of a private and a public key, the public parts are published to the server and can be used by other devices to create a shared secret for encrypted communication using the Curve25519 key or verify signatures created by the private Ed25519 key, respectively.

Managing these key pairs is the main use for devices. Depending on the use case they can be newly created for every session, for example in a web client, or persisted and used across multiple sessions, for example in a desktop or mobile application. Devices ensure that trusted data, e.g. the private parts of key pairs, never have to be sent over a network. They remain on the actual hardware device or are created anew every time the user logs in. Since the device keys are only used to establish one-on-one connections, a user can access older group conversations even when logging in as a new device with newly created device keys, since other devices will share the necessary Megolm session keys if they can verify the new device (see 4.6).

2.1.2 Events

Matrix builds on top of very simple technologies, it is basically just JSON data exchanged using HTTP requests. All JSON objects have the same basic structure:

```
{
```

[2]`https://cr.yp.to/ecdh.html`
[3]`https://ed25519.cr.yp.to/`

```
    "content": {...},
    "type": "..."
}
```

This is called an event in Matrix. It has an event type and arbitrary JSON data as content, the form of which depends on the type. All information that is sent or received is represented as an event in Matrix.

There is a standard set of event types, given by the specification, but custom event types can be created freely. They just have to be namespaced correctly, in order to avoid conflicts with other custom event types. Matrix uses the Java package naming convention for event types[4]. Event types defined by the Matrix specification use the reserved top-level namespace `m`, for example `m.room.message`.

Events are either sent in a room, where all the room's participants will receive it, or directly to another device as a to-device message, with the former being the more common approach. Events sent in a room dictate everything that happens in that room, including messages sent, members added/removed, encryption enabled, etc.

2.1.3 Rooms

A room is a virtual place where users can send events to multiple other users. Each room has a unique room ID of the form `!opaque_id:domain`, where domain is the hostname of the homeserver where the room was created. It does not actually reside on that homeserver, since all the information making up a room is federated between the homeservers of all users participating in that room.

A room is entirely defined by the events exchanged in its context. The entire state of the room is given by its event graph since every action in the room is an event. The root of this event graph is always an `m.room.create` event:

```
{
  "content": {
    "m.federate": true,
    "predecessor": {
      "event_id": "$something:example.org",
      "room_id": "!oldroom:example.org"
    },
```

[4]Defined in chapter 7.7 of the Java specification
`https://docs.oracle.com/javase/specs/jls/se6/html/packages.html`

```
    "room_version": "11"
  },
  "event_id": "$143273582443PhrSn:example.org",
  "origin_server_ts": 1432735824653,
  "room_id": "!jEsUZKDJdhlrceRyVU:example.org",
  "sender": "@example:example.org",
  "state_key": "",
  "type": "m.room.create",
  "unsigned": {
    "age": 1234
  }
}
```

Every change to the room, every message, users joining or leaving, etc., is an event in the event graph. Events in the graph are ordered by the server, and events that have been synchronized from different servers are ordered according to a state resolution algorithm, which differs depending on the room version. Figure 1 shows an event graph where two paths have to be merged because two servers handled different events starting at E1. If E3 and E4 perform the same action, for example, if they both rename the room, the state resolution algorithm has to figure out what the name of the room should be at E5.

```
  E0
  |
  E1
 /  \
E2  E4
|    |
E3   |
 \  /
  E5
```

Figure 1: Event graph diverging at event E1, which has to be merged at E5

The state of the room at any point in time is given by the event graph up to the event at that time. It can be recreated by applying all the events in order, starting from the root `m.room.create` event. This will yield the specific room state at the given point in time.

2.2 Client-Server API

The Client-Server API, as the name implies, defines how clients interact with a server. It includes pretty much every action that a user can perform, from sending a message or creating a room to verifying devices and setting up VoIP calls.

API calls are made as HTTP requests. The following HTTP methods are used in the specification, all in conformance with the definition of their usage in the HTTP specification[5]:

GET
Transfer a current representation of the target resource.

POST
Perform resource-specific processing on the request content.

PUT
Replace all current representations of the target resource with the request content.

DELETE
Remove all current representations of the target resource.

OPTIONS
Describe the communication options for the target resource.

The Matrix specification defines API calls using one or more HTTP methods, a URL, and path/request/response parameters. To make a call, you send an HTTP request using the specified method for that URL to the server. Parameters are supplied either as path parameters in the URL path or request parameters in the request body. For example, to get a room's visibility on the server's public room directory, we can call `/_matrix/client/v3/directory/list/room/{roomId}`. The specification lists one path parameter, a string called `roomId`, and GET as the method. So if we want to make a call to this API on our homeserver `matrix.org`, we make a GET request to `https://matrix.org/_matrix/client/v3/directory/list/room/{roomId}`, passing the room id in the path.

Some endpoints, such as this one, can be called with multiple HTTP methods. For the `directory/list/room` API, we can also call the same URL with PUT to set the room's visibility. The only difference is that we have to pass the `visibility` parameter in the request body.

[5] `https://datatracker.ietf.org/doc/html/rfc9110.html#table-4`

2.2.1 Authentication

Most API endpoints require authentication, both to ensure access rights and because most endpoints return data specific to the user, so the server has to know which user made the call. Authentication is done using access tokens. When logging in using username and password, or one of the other supported login methods, we effectively create a new device. The server tells us our newly assigned device ID, along with an access token that can be used to make API calls that this user has access to. HTTP requests can be authenticated by either supplying the access token as a query parameter `?access_token=<token>` or by setting the `Authorization` HTTP header to `Bearer <token>`. Since the access token is created alongside the device ID, it also identifies the device itself.

3 Cryptography

Matrix uses the libolm library[6] for encryption of one-on-one and group messages, as well as cryptographic signing. Libolm is written in C and C++ and implements the Olm cryptographic ratchet, which is an implementation of the Double Ratchet algorithm developed for the Signal messenger[3]. It also implements Megolm, an "AES-based cryptographic ratchet intended for group communications"[2] that was developed on top of the Double Ratchet specifically for Matrix.

After going over some basics we will look at how Olm works by examining the Double Ratchet algorithm that it implements, and then take a look at Megolm and signing.

3.1 Diffie-Hellman Key Exchange

In order to establish a secure connection over an insecure channel, we need a way to establish a secret without sending any confidential information[1]. A very prominent way to achieve this is using a Diffie-Hellman key exchange. Diffie-Hellman is a public-key protocol, that allows two users to establish a shared secret without sending any data that would allow an attacker that intercepted their connection to reproduce the secret.

It was initially based on the multiplication of integers modulo p, where p is a prime number. The steps to establish a shared secret are as follows:

1. Alice and Bob agree on a shared prime p and a base g which is smaller than p, both of these can be publicly known.

2. They randomly choose private keys a and b which are numbers smaller than p. These numbers have to be kept secret and are not exchanged.

3. From their private keys they calculate public keys A and B as $A = g^a \ mod \ p$ and $B = g^b \ mod \ p$. These are exchanged and do not have to be kept secret.

4. With the other's public key and their own private key, they can calculate a shared secret K that is the same for both Alice and Bob, but which cannot be recreated by an attacker without knowing one of their private keys. The shared secret K is calculated as $K_a = B^a \ mod \ p$ and $K_b = A^b \ mod \ p$. K_a and K_b are equal since

$$K_a \equiv B^a \equiv (g^b)^a \equiv (g^b)^a \equiv g^{ab} \pmod p$$

$g^{ab} \equiv g^{ba} \pmod p$ because multiplication is commutative and so $K_a = K_b$.

[6]`https://gitlab.matrix.org/matrix-org/olm`

The task of calculating a from $A = g^a \ mod \ p$ is called discrete logarithm, and there is no known efficient algorithm for calculating the result for large values of p. This means that the security of the key exchange is based on choosing a sufficiently large prime p. Note that it is not proven that there doesn't exist an efficient algorithm for this, so in theory, it is possible that one day a fast solution is found and the key exchange becomes unsafe to use[9].

Many implementations, including libolm, do not use multiplication of integers modulo p, but instead use operations on elliptic curves instead. The idea stays the same, but the reverse operation of calculating a from $A = g^a \ mod \ p$ becomes even more computationally expensive, allowing for smaller key sizes and therefore faster calculations of the public keys and shared secret. Diffie-Hellman using elliptic curves is referred to as ECDH[4][7].

3.2 Hash-Based Message Authentication Codes

Hash-based message authentication codes (HMACs) are a mechanism for checking the integrity of messages. An HMAC can be used with any cryptographic hash function H and a previously established shared secret K. The used hash function is expected to work on blocks of data that are B bytes long, and produce outputs that are L bytes long. The shared secret K can be used directly if its length does not exceed B bytes, otherwise it is first hashed using H. It is recommended to use a secret that is at least L bytes long[5]. To calculate the HMAC of data D using the key K, we perform:

$$HMAC(K, D) = H((K \oplus opad) \mid H((K \oplus ipad) \mid D))$$

where $\oplus$ is a bitwise XOR, $\mid$ is concatenation, *opad* is the byte 0x5C repeated B times and *ipad* is the byte 0x36 repeated B times. The process of calculating, in detail, goes like this:

1. Create a B byte long string by padding K with zeros at the end
2. Bitwise XOR that string with *ipad*
3. Append D to the result and then apply H to the concatenation
4. Bitwise XOR the B byte string created in step 1 with *opad*
5. Append the hash output from step 3 to the result from step 4
6. Apply H to the concatenation from step 5, this is the result

3.3 HMAC-based Extract-and-Expand Key Derivation Function

The HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is a key derivation function that takes some source of initial keying material and derives from it one or more cryptographically strong secret keys[6]. It is intended to be a general-purpose key derivation function (KDF). HKDF has two separate steps, the extract and the expand step. The extract step takes the input keying material and "extracts" from it a fixed-length pseudorandom key K[6]. This ensures that, independently of what sort of keying material is provided, we have a cryptographically strong, pseudorandom key[6]. The expand step produces pseudorandom output keys. If the input keying material is already a viable pseudorandom key, the extract step can be skipped.

The extract step is defined as the HMAC of the input keying material (IKM) together with a salt that serves as the HMAC key, to produce a pseudorandom key (PRK):

$$PRK = HMAC(salt, IKM)$$

If no salt is specified, a string of zeros with the same length as the hash function's output is used instead.

With L being the length in bytes of the HKDF's final output, $HashLen$ being the length in bytes of the hash function H's output and N being the number of hash outputs that fit into the length of the final output ($N = ceil(L/HashLen)$), the final output is defined as the first L bytes of $T(1) \mid T(2) \mid \ldots \mid T(N)$. T is defined as:

$T(0) = empty\ string\ (zero\ length)$

$T(1) = HMAC(PRK, T(0) \mid info \mid 0x01)$

$T(2) = HMAC(PRK, T(1) \mid info \mid 0x02)$

$\ldots$

The *info* parameter is optional and can be used to "bind the derived key material to application- and context-specific information"[6]. If it is provided, it has to be independent from the IKM.

3.4 Olm

The basic idea behind the Double Ratchet algorithm[8], which is what Olm implements, is that users can exchange encrypted messages based on a shared secret, which is agreed upon using a key agreement protocol. Using two ratchets, new encryption keys are derived, such that earlier keys cannot be retrieved from later ones. Additionally, the public parts of Diffie-Hellman values are sent along with each message. These are mixed into the calculation for new message keys, to make sure that it is also not possible to derive later keys from earlier ones.

The algorithm is based on key derivation functions and KDF chains. A KDF is a function that takes a secret, random key, and some input, and produces an output key, like HKDF describes in 3.3. Without the key that was used to generate it, the output key is indistinguishable from random data. Even if the input key is known, the KDF still acts as a cryptographically secure hash on the input data, meaning the input cannot be recovered from the output.

By taking the output of one KDF, and using part of it as the input key for another one, we get a KDF chain. The rest of the output becomes the actual output key, which can be used like it normally would. Chaining KDFs like this provides three benefits, as described in [8]:

Resilience
: Without knowing the input key, output from the KDF chain appears random, even if the input data can be controlled

Forward security
: If a key is compromised, earlier outputs will still appear random

Break-in recovery
: Similarly, later outputs appear random as well, as long as sufficient entropy has been introduced since the key was compromised

In order to start a Double Ratchet session, two users store a KDF key for three chains. Each of them has a unique root chain, as well as a sending and a receiving chain, where one user's sending chain is the other user's receiving chain, and vice versa. Along with every message, they send a new Diffie-Hellman public key. The Diffie-Hellman output secrets become inputs for the root chain, and the output keys from the root chain become KDF keys for the sending/receiving chains. This is what is called the Diffie-Hellman ratchet.

The sending and receiving chains themselves are advanced with every message. Their output keys are used as message keys for encrypting/decrypting messages. This is the symmetric-key ratchet.

After a brief overview of the different cryptographic keys used in the Double Ratchet algorithm, the symmetric-key and Diffie-Hellman ratchets will be explained in a bit more detail, and finally, we look at how they are combined into the Double Ratchet.

The following keys are used throughout the algorithm:

- I_A/I_B, identity keys for Alice and Bob
- E_A/E_B, onetime keys for Alice and Bob
- R_i, 256 bit root key
- T_i, ratchet key
- $C_{i,j}$, 256 bit chain key
- $M_{i,j}$, message key

Identity keys I

The identity keys are Curve25519 key pairs. Both users generate a pair on device creation and the public parts are uploaded to the server. The identity keys are used together with the onetime keys E to create a shared secret S and to calculate new chain keys C.

Onetime keys E

Similar to the identity keys, onetime keys are Curve25519 key pairs. A fixed number of them is generated and uploaded to the server where they can be claimed by other devices to initiate one-on-one Olm sessions. When another device claims a onetime key, it is removed from the server and the device should generate and upload a new one to ensure there is a sufficient number of them available at all times.

Root key R

The initial root key (R_0) is calculated together with the initial chain key (C_0) by calculating an HMAC-based key derivation function using SHA-256 as the hashing function (HKDF-SHA-256) of the shared secret S and the string "OLM_ROOT", with a salt of 0 and a length of 64 bytes:

$$R_0||C_{0,0} = HKDF(0, S, "OLM\_ROOT", 64)$$

The 64 bytes, or 512 bits, are split, the first 256 bits become (R_0), and the remaining 256 bits become (C_0). Future root keys and chain keys are calculated by performing an HKDF-SHA-256 Elliptic Curve Diffie-Hellman key exchange (ECDH) of both the previous and current ratchet keys T_i and T_{i-1} and the pre-defined string "OLM_RATCHET", using the previous root key as the salt:

$$R_i||C_{i,0} = HKDF(R_{i-1}, ECDH(T_{i-1}, T_i), "OLM\_RATCHET", 64)$$

The root key is used to generate the chain keys $C_{i,0}$ for both devices since it can be calculated by both from the shared secret they received during the Diffie-Hellmann key exchange.

Ratchet key T

The ratchet keys are generated by each user each time they first send a message after receiving one from the other. They make up the Diffie-Hellman ratchet and ensure, that the KDF chains for generating message keys get replaced regularly.

Chain key C

Starting from the initial chain key $C_{i,0}$, which is calculated from the root key, the next chain key $C_{i,j}$ is calculated as the HMAC-SHA-256 of "\x02", using the previous chain key $C_{i,j-1}$ as the key:

$$C_{i,j} = HMAC(C_{i,j-1}, "\backslash x02")$$

The chain key $C_{i,j}$ is used to calculate the message key $M_{i,j}$, as well as the next chain key $C_{i,j+1}$.

Message key M

Instead of using the KDF output from the sending/receiving chains directly, the output is used to generate message keys for actually en-/decrypting messages. The message key $M_{i,j}$ is calculated from the chain key $C_{i,j}$ in much the same way as the subsequent chain key, by taking the HMAC of the chain key and a constant string:

$$M_{i,j} = HMAC(C_{i,j}, "\backslash x01")$$

3.4.1 Symmetric-key ratchet

The output keys $M_{i,j}$ from the symmetric-key ratchet are what is used to actually encrypt/decrypt messages. A unique message key is generated for every message (see Figure 2). The initial KDF key is taken from the root chain, and the inputs for the sending/receiving chain are constant values, which means that with just this ratchet, if a message's chain key got compromised, future messages could be compromised as well.

The message keys themselves are not used to derive any other keys though, so they can be deleted after encryption/decryption because that is their only use. But for the same reason, they can also be stored without risking the integrity of the KDF chain, which could be "useful for handling lost or out-of-order messages"[8].

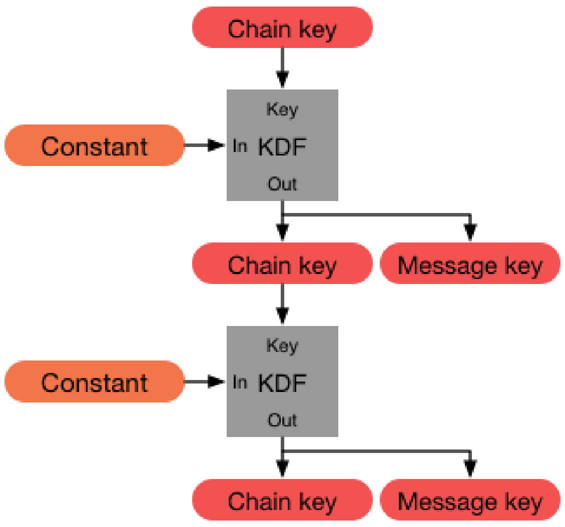


Figure 2: Symmetric-key ratchet showing two ratchet steps, calculating the next chain key along with the message key, from The Double Ratchet Algorithm[8]

3.4.2 Diffie-Hellman ratchet

With just the symmetric-key ratchet alone, there would be a unique message key for every message, but if the private part of a chain key was compromised, the KDF chain could be reconstructed, since the input is constant, and all future messages could be decrypted. So a Diffie-Hellman ratchet is used to generate new chain keys for the sending and receiving KDF chains. Both users generate a Diffie-Hellman key pair, which is referred to as a ratchet key pair, and every message they send contains the public part of their current ratchet key pair. Whenever a user receives a message where the public part of the ratchet key pair changed, they perform a Diffie-Hellman ratchet step and generate a new ratchet key pair. This means that if a key pair were to be compromised, the third party could only reconstruct the ratchet until a message with a new public key was received and the key pair was replaced.

The process can be summarized like this:

1. Alice claims the public part of Bob's ratchet key pair
2. She performs a Diffie-Hellman with the private part of her own ratchet key pair
3. With her first message Bob also receives the public part of her key pair
4. He can then perform a Diffie-Hellman using his private key and the public key he just received to obtain the same output that Alice got and decrypt the message
5. To complete the ratchet step Bob replaces his key pair with a new one
6. The next message he sends will be encrypted with the Diffie-Hellman output from Alice's last public key and his new private key, and will include his new public key
7. Alice can then perform the next ratchet step using that new public key, calculating the Diffie-Hellman with her private key to decrypt the message and generating a new ratchet key pair
8. Again, she uses the public key she received along with her newly generated private key to get the next sending message key and so on

These steps are visualized in Figure 3:

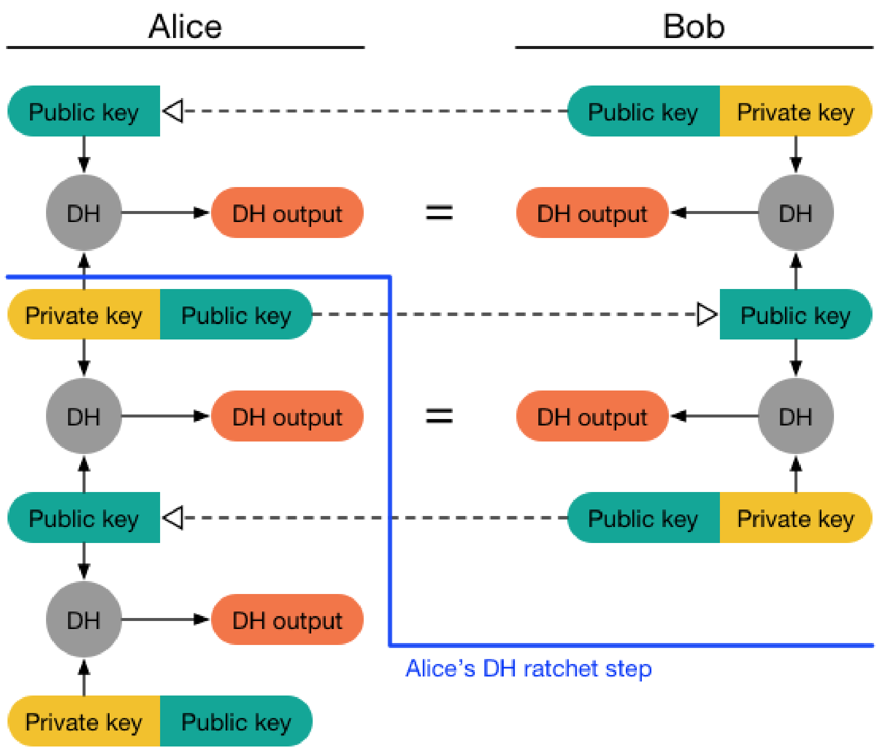


Figure 3: Diffie-Hellman ratchet showing three ratchet steps, from The Double Ratchet Algorithm[8]

3.4.3 Double Ratchet

The symmetric-key and Diffie-Hellman ratchets are combined into the Double Ratchet. With this, we get the security benefits of both, with new message keys for every message that is sent and new KDF chains every time a user receives a message after sending one themselves. A complete example of a Double Ratchet session can be seen in Figure 4, this example shows the perspective of one of the two users. The exact steps are as follows:

1. A shared secret S is generated using a key agreement protocol, specifically by computing a Diffie-Hellman from both device's identity and onetime keys:

$$S = ECDH(I_A, E_B)||ECDH(E_A, I_B)$$

2. Both users also generate Diffie-Hellman key pairs in advance
3. The shared secret is used as the initial chain key for both user's root chains
4. The first user to send a message, the one that Figure 4 is depicting, claims the public part of the other user's key pair and performs a Diffie-Hellman with their own private key
5. That result is input into the root KDF chain, the output of which is the initial chain key for the sending chain
6. Using a predefined, constant input, the output of the KDF is the message key used for encrypting the first message
7. Before sending the next message, one is received, along with a public key
8. The Diffie-Hellman of the new public key and the private key that was already used becomes the initial chain key for the receiving chain
9. Again, together with a constant input the message key for decrypting the received message is generated
10. Since a new public key was received, the ratchet key pair is replaced, and along with it a new sending chain is initialized
11. Three message keys for three messages are generated, even though a second message is received in the meantime since that message has the same public key attached to it as the one before it, which means it was sent before the first of the newly sent messages arrived since the other user's ratchet key pair would have been replaced otherwise
12. After receiving our new public key the other user eventually replaced their key pair, and sent another message with that new public key, causing a new receiving chain along with a new sending chain as soon as the next message is sent

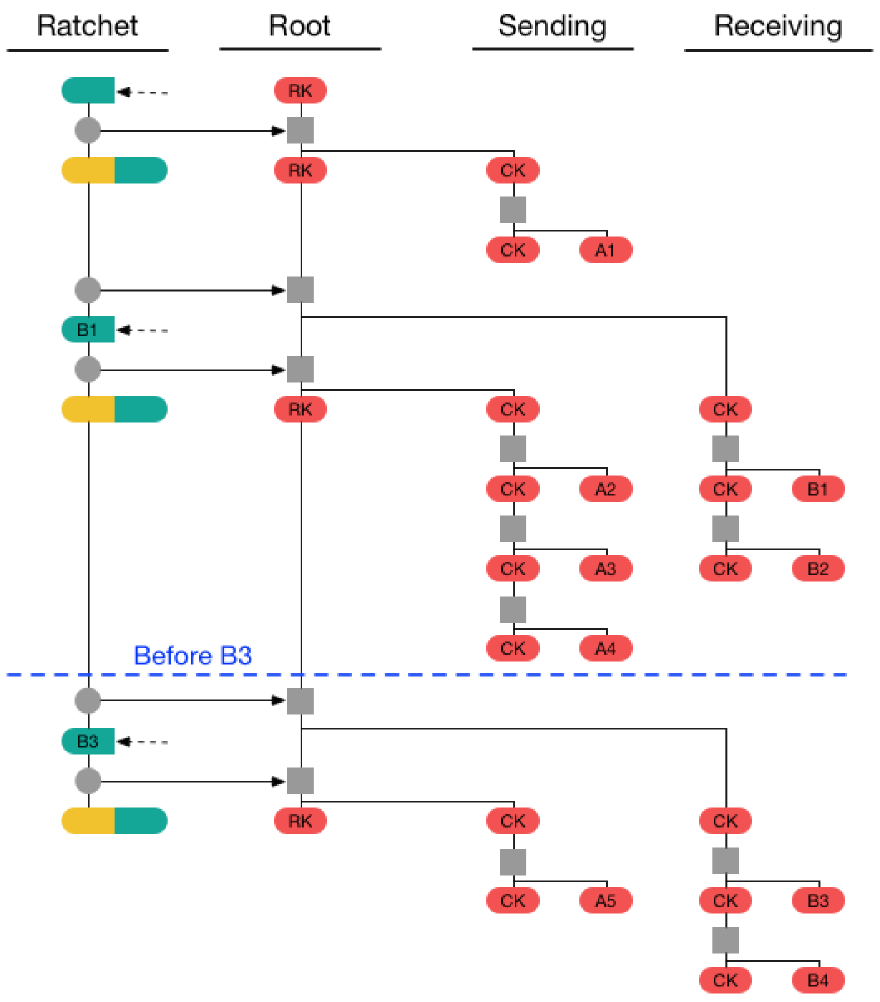


Figure 4: Double Ratchet showing a total of five messages sent and four received, from The Double Ratchet Algorithm[8]

3.5 Megolm

Megolm is a ratchet algorithm for group communication, where a message is expected to be received by a potentially large number of clients. Instead of keeping one-on-one sessions with every potential recipient, each client has their own outbound session and shares the session details with each recipient once. Other clients are then able to decrypt messages by that client. They can also request the session details from other clients without needing to talk to the sender directly (see 4.7.2) and use those to decrypt previously received group messages. In contrast to the Double Ratchet used in Olm, when a Megolm session is compromised, the attacker will be able to decrypt future messages. It is therefore recommended that new Megolm sessions are generated and shared in regular intervals, although this is not enforced by the specification at the time of writing.

A Megolm session consists of an Ed25519 keypair K, which is used to authenticate messages, a 32-bit counter i, and a ratchet R_i, which itself consists of four 256-bit values $R_{i,j}$ for $j \in 0, 1, 2, 3$. Initially, the keypair K is generated, i is set to 0 and the four ratchet values are all initialized with cryptographically secure random data. The public part of K is used as the session's identifier. Before sending the first message, the session's details, i, R_i, and the public part of K, are shared with other clients using one-on-one Olm sessions.

The AES key and IV used to encrypt messages, as well as the HMAC key for later authenticating the encrypted message, are generated by taking the HKDF of R_i and the constant string "MEGOLM_KEYS":

$$AES\_KEY_i \mid\mid HMAC_KEY_i \mid\mid AES\_IV_i = HKDF(0, R_i, "MEGOLM\_KEYS", 80$$

Here, || means splitting the HKDF output. Out of the 80 bytes/640 bits that are generated, 256 bits go to the AES and HMAC key respectively, and the remaining 128 bits become the AES IV. The encrypted message is run through HMAC using the generated key, and then the ciphertext along with the generated MAC is signed using the Ed25519 keypair.

The ratchet is advanced every time a message is sent. The four values making up the ratchet, $R_{i,0}$, $R_{i,1}$, $R_{i,2}$ and $R_{i,3}$, are updated as follows:

$$R_{i,0} = \begin{cases} H_0\left(R_{2^{24}(n-1),0}\right) & \text{if} \exists n | i = 2^{24}n \\ R_{i-1,0} & \text{otherwise} \end{cases}$$

$$R_{i,1} = \begin{cases} H_1\left(R_{2^{24}(n-1),0}\right) & \text{if} \exists n | i = 2^{24}n \\ H_1\left(R_{2^{16}(m-1),1}\right) & \text{if} \exists m | i = 2^{16}m \\ R_{i-1,1} & \text{otherwise} \end{cases}$$

$$R_{i,2} = \begin{cases} H_2\left(R_{2^{24}(n-1),0}\right) & \text{if} \exists n | i = 2^{24}n \\ H_2\left(R_{2^{16}(m-1),1}\right) & \text{if} \exists m | i = 2^{16}m \\ H_2\left(R_{2^{8}(p-1),2}\right) & \text{if} \exists p | i = 2^{8}p \\ R_{i-1,2} & \text{otherwise} \end{cases}$$

$$R_{i,3} = \begin{cases} H_3\left(R_{2^{24}(n-1),0}\right) & \text{if} \exists n | i = 2^{24}n \\ H_3\left(R_{2^{16}(m-1),1}\right) & \text{if} \exists m | i = 2^{16}m \\ H_3\left(R_{2^{8}(p-1),2}\right) & \text{if} \exists p | i = 2^{8}p \\ H_3\left(R_{i-1,3}\right) & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} H_0(A) &\equiv \text{HMAC}(A, \text{``“x00”}) \\ H_1(A) &\equiv \text{HMAC}(A, \text{``“x01”}) \\ H_2(A) &\equiv \text{HMAC}(A, \text{``“x02”}) \\ H_3(A) &\equiv \text{HMAC}(A, \text{``“x03”}) \end{aligned}$$

This can be read as:

- Every 2^8 messages, $R_{i,2}$ and $R_{i,3}$ are reseeded from $R_{i,2}$.
- Every 2^{16} messages, $R_{i,1}$, $R_{i,2}$ and $R_{i,3}$ are reseeded from $R_{i,1}$.
- Every 2^{24} messages, all four values are reseeded from $R_{i,0}$.

Outbound sessions only store the current values for i, K, and R_i, while inbound sessions can also keep the earliest versions of the values they received to retain the ability to decrypt past messages.

4 Implementation

The main part of this thesis is the implementation of a library allowing communication with a Matrix server. This library is written in C for two reasons, the first one being that the code should run on an ESP32, for which C and C++ are some of the go-to programming languages. The second reason is that, even though the use-case for this thesis is embedded development, the library runs on any modern desktop operating system and libraries written in C can be more easily interfaced from other programming languages, making it possible to speak to Matrix servers from virtually any programming language using just one library. To my knowledge, there are currently no other Matrix client libraries written in C readily available that support end-to-end encryption.

The different features of the library, as well as the corresponding parts of the Matrix specification, will be explained here.

4.1 Overview

Almost all functions operate on one of the following structs, each providing a sort of wrapper around a part of the Matrix/Olm ecosystem:

`MatrixClient`
: A session, basically a logged-in Matrix device, including a `MatrixOlmAccount`, known Olm/Megolm sessions, and information about other devices the user might want to interact with

`MatrixDevice`
: A device as specified by Matrix (see 2.1.1)

`MatrixOlmAccount`
: Wrapper around `OlmAccount` from libolm

`MatrixOlmSession`
: Wrapper around `OlmSession` from libolm

`MatrixMegolmInSession`
: Wrapper around `OlmInboundGroupSession` from libolm

`MatrixMegolmOutSession`
: Wrapper around `OlmOutboundGroupSession` from libolm

All of these structs and the accompanying functions can be found in `src/matrix.h` and `src/matrix.c`.

There is also an HTTP layer providing GET, PUT, and POST requests. The corresponding function definitions are in `matrix.h` as well, and there are platform-specific implementations using the Mongoose networking library[7] and one specifically for the ESP32 in `src/matrix_http_mongoose.c` and `src/matrix_http_esp32.c` respectively. See 4.2 and 5.3 for details.

Many of the basic features are implemented by simply sending an HTTP request to a specific endpoint. These requests usually include some data, either in the form of query parameters of the form `https://url?param1=value&param2=value`, or in the form of JSON data in the body of a POST/PUT request.

Most requests require an access token. An access token can be obtained by logging in to a Matrix server and is associated with a device. It can be copied from an existing device, for example from a logged-in web client such as Element Web[8], or by logging in through the API. Password-based login can be done using the `login`[9] API by specifying `m.login.password` as the login type:

```
{
  "type": "m.login.password",
  "identifier": {
    "type": "m.id.user",
    "user": "<user_id or user localpart>"
  },
  "password": "<password>"
}
```

This creates a new device and returns its device ID, as well as an access token. So an easy way to get an access token for making requests is to log in, which using the client library can be accomplished by calling the `MatrixClientLoginPassword` function:

```
bool
MatrixClientLoginPassword(
    MatrixClient * client,
    const char * username,
```

[7]https://github.com/cesanta/mongoose
[8]https://github.com/vector-im/element-web
[9]https://spec.matrix.org/v1.8/client-server-api/#login

```
    const char * password,
    const char * displayName);
```

This takes as the first parameter an initialized MatrixClient struct, logs in using the supplied username and password, and sets the `client`'s device ID and access token internally automatically. The client can then make requests which require authentication, such as sending and receiving messages in non-encrypted rooms. In order to be able to send and receive end-to-end encrypted messages, a device first has to be verified. This process is described in 4.6.

4.2 HTTP Requests

`matrix.h` defines six functions that are used to establish an HTTP connection and make requests. All functions operate on an opaque struct called `MatrixHttpConnection`, the individual implementations of which differ between supported platforms. The six functions that have to be implemented for any backend are:

`MatrixHttpInit`
: Allocates and initializes the platform-specific `struct MatrixHttpConnection` and connects to the specified host.

`MatrixHttpDeinit`
: Closes the connection and then uninitializes and frees the `struct MatrixHttp-Connection`.

`MatrixHttpSetAccessToken`
: Sets the access token for requests that require authentication, see 2.2.1.

`MatrixHttpGet/Post/Put`
: Sends a GET/POST/PUT request to the specified URL and with the specified request body if applicable.

There are currently two backends, one using the Mongoose networking library which supports both desktop and embedded systems, as well as one specifically for the ESP32 using the esp_http_client API that the ESP-IDF provides[10].

[10]`https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/esp_http_client.html`

4.3 Syncing

Syncing refers to the act of getting an update from the server with new events for this device. This includes room events and messages, as well as to-device messages. Syncing is done by sending a GET request to `/_matrix/client/v3/sync`[11].

The first sync request is sent without a `since` parameter and returns the state of the server at that time. Future calls to sync should include a `since` parameter that is a token included in the response to the previous call as the `next_batch` field. The GET request returns immediately if new events are present, otherwise it returns after a set timeout, even if there are no new events. If the timeout is set to zero, which is the default value if the parameter is not explicitly included, the request always returns immediately, even if there are no new events. The following parameters can be passed[12]:

filter

The ID of a filter, created using the filter API or a filter JSON object encoded as a string.

full_state

Controls whether to include the full state for all rooms the user is a member of or only state which has changed since the point indicated by since.

set_presence

Controls whether the client is automatically marked as online by polling this API. One of: [offline, online, unavailable].

since

A point in time to continue a sync from. This should be the next_batch token returned by an earlier call to this endpoint.

timeout

The maximum time to wait, in milliseconds, before returning this request. If no events (or other data) become available before this time elapses, the server will return a response with empty fields.

By default, this is 0, so the server will return immediately even if the response is empty.

[11] `https://spec.matrix.org/v1.8/client-server-api/#get_matrixclientv3sync`
[12] Taken from `https://spec.matrix.org/v1.8/client-server-api/#get_matrixclientv3sync`

In order to sync using the library, the function `MatrixClientSync` can be called, with an output buffer to store the response and a string `nextBatch` which, if not empty, is used to only get the most recent events. Calling `MatrixClientSync` not only returns the response to the sync request, some types of events are also handled automatically. The function parses the JSON response and passes the event object to a handler function. Events are handled in two steps, to_device and room events.

Events sent directly from other devices, also referred to as "to_device" events, are containted in the response in the field "to_device.events" and are stored as an array of objects.

Room events are returned sorted by room. The response contains an object under the field "rooms.join" which maps the room ID of rooms that the user has joined to objects containing the relevant sync information. These objects have a field "timeline.events" which holds a list of events similar to the one in "to_device.events".

Both to_device and room events are handled virtually the same way, with the only difference being that the handler for room events is passed the room ID as a string because it is needed when processing certain events. Apart from the events associated with verification, which are detailed in 4.6, the following events are handled automatically when calling `MatrixClientSync`, identified by event type:

m.room.encrypted

Depending on the "content.algorithm" field this can be encrypted using a one-on-one Olm, or a group Megolm session. The corresponding Olm/Megolm session is looked up and, provided a matching session is known, the event is decrypted and passed back to the event handler.

m.room_key/m.forwarded_room_key

This contains the details of a Megolm session, either sent proactively by another device because it created a new one or as a response to an m.room_key_request that we previously sent. The session ID and key are read from the event and a new Megolm session is created and stored, allowing us to decrypt future events encrypted by that Megolm session.

4.4 Storing External Information

Many features require external information, for example, when establishing a new Olm session the client needs to have a copy of the other device's Curve25519 public key. This information can be requested from the server, but requesting the same information multiple times adds unnecessary overhead. For that reason, information about other devices, as well as Olm and Megolm sessions, is stored locally. When the library needs some information it checks if it is available locally, and otherwise requests it and then stores it for the next time it is needed.

Local information is stored in four different arrays that are part of the `struct MatrixClient`: `megolmInSessions`, `megolmOutSessions`, `olmSessions` and `devices`. Inbound and outbound Megolm sessions are stored separately (see 4.7.2). The different arrays containing session information have corresponding types wrapping the base libolm session types, `MatrixMegolmInSession`, `MatrixMegolmOutSession` and `Matrix-OlmSession`, and all of them store the session's current state. The device array has the type `MatrixDevice`, which stores the device ID, and the public parts of the Curve25519 identity keypair and the Ed25519 signing key pair. All arrays are of a fixed size, which can be set in `matrix.h` using the preprocessor defines `NUM_MEGOLM_SESSIONS`, `NUM_OLM_SESSIONS` and `NUM_DEVICES` respectively.

4.5 Signing JSON

Some events, such as uploading onetime keys, require the JSON event body to be signed to ensure the event actually came from the claimed sender. To do this, the event is transformed into its canonical form, signed using the device's Ed25519 keypair, and then encoded as unpadded Base64.

"Canonical JSON" refers to a convention of formatting JSON data given by the Matrix specification, which guarantees that every JSON object has exactly one representation, which allows other devices to format the JSON the same way and validate a signature. The following rules lead to canonical JSON, and leave only one possible byte representation:

- Encode as UTF-8
- Remove all whitespace outside of strings
- Sort dictionary keys lexicographically
- All numbers must be integers in the range [-(2**53)+1, (2**53)-1], represented without exponents or decimal places
- Negative zero (-0) is not allowed to appear

4.6 Device Verification

When interacting with other devices, for example when requesting a session key to decrypt an end-to-end encrypted group message that we don't have the key for, the other device might check if our device was verified before, and possibly refuse to share the session key with us if that is not the case. So in order to properly communicate with other users and devices, we have to verify our device to prove that we are who we are claiming to be, this prevents man-in-the-middle attacks. Verification involves generating and sharing information with another device, in part over an external channel to ensure validity. The steps to verify a new device roughly look like this:

1. Device A initiates verification of device B and provides a list of supported verification methods
2. Device B accepts the request and responds with a list of supported verification methods
3. Device A then starts the verification process, making sure to obtain a copy of Device B's device key
4. Device B selects a key agreement protocol, hash algorithm, message authentication code, and SAS method supported by device A, in turn making sure to obtain a copy of device A's device key
5. Device B creates a Curve25519 key pair
6. Device B sends a commitment hash to device A as an m.verification.accept message
7. Device A stores the commitment hash for later and creates a Curve25519 key pair as well, sending an m.key.verification.key message with the unhashed public key
8. Device B sends back an m.key.verification.key message with its own unhashed public key
9. Device A checks the commitment hash using the received public key
10. Both devices perform an ECDH and display a SAS to their users
11. The users compare their SASs and confirm whether they match or not
12. Both devices send m.key.verification.mac messages

13. Device A checks the MAC and both devices send an m.key.verification.done message

Following is a description of how these steps are implemented in the library. Verification starts with another device sending a verification request in the form of a m.key.verification.request message. Every step is a response to a message from the device that initiated the verification. They are handled automatically when syncing (see 4.3), and the verification process will be described by outlining the response to each of these messages. Verifying other devices using the library, i.e. initiating the verification process for another device, is not implemented.

m.key.verification.request

This is the message that initiates verification. The requesting device sends us a list of supported verification methods, as well as a transaction ID:

```
{
  "content": {
    "from_device": "DeviceA",
    "methods": ["m.sas.v1"],
    "timestamp": 1559598944869,
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.request"
}
```

We reply with a m.key.verification.ready message. This includes the subset of verification methods from the request that we support. This is currently only “m.sas.v1”. We also store the transaction ID and include it in our response, since this will be referenced throughout the whole verification process to associate individual messages. The response looks like this:

```
{
  "content": {
    "from_device": "DeviceB",
    "methods": ["m.sas.v1"],
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.ready"
}
```

m.key.verification.start

We communicated that we are ready to begin the actual verification, and this message includes additional information necessary for creating the commitment hash that we will include in our response:

```
{
  "content": {
    "from_device": "DeviceA",
    "hashes": ["sha256"],
    "key_agreement_protocols": ["curve25519"],
    "message_authentication_codes": [
      "hkdf-hmac-sha256.v2",
      "hkdf-hmac-sha256"
    ],
    "method": "m.sas.v1",
    "short_authentication_string": [
      "decimal",
      "emoji"
    ],
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.start"
}
```

We first create and initialize an `olm_sas` struct. This is part of libolm and supports the necessary hashing and elliptic curve key agreement functions. It also creates the Curve25519 key pair for us.

We create the canonical representation of the m.key.verification.start message and append it to the public key from our generated key pair.

This concatenation is hashed using SHA256 (as specified in the `m.key.verification.start` message), this is our commitment hash. The Base64 encoded hash is sent back as part of an `m.key.verification.accept` message, alongside the hashing function, key agreement function, and message authentication protocol we chose:

```
{
  "content": {
    "commitment": "fQpGIW1Snz+pwLZu6sTy2aH...",
    "hash": "sha256",
    "key_agreement_protocol": "curve25519",
    "message_authentication_code": "hkdf-hmac-sha256.v2",
    "method": "m.sas.v1",
    "short_authentication_string": [
      "decimal",
      "emoji"
    ],
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.accept"
}
```

m.key.verification.key

The other device sent us the public part of their Curve25519 key pair, and we reply with our own. This allows the other device to check the commitment hash we just sent. The response we send looks almost identical to the `m.key.verification.key` we received, just with our own public key.

```
{
  "content": {
    "key": "fQpGIW1Snz+pwLZu6sTy2aH...",
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.key"
}
```

Short Authentication String

Given the commitment hash could be verified, we are ready to generate an SAS, or Short Authentication String. This is shown to both users and they have to compare them using an out-of-band channel, like a voice call or in person. The first thing we do to create our SAS is to construct an info parameter. Depending on the key agreement protocol we selected when sending the `m.key.verification.accept`, this looks like one of two options:

If the "key_agreement_protocol" is "curve25519-hkdf-sha256", the info parameter is the concatenation of the following elements, separated by "|" characters:

- The string MATRIX_KEY_VERIFICATION_SAS
- The Matrix ID of the user who sent the m.key.verification.start message
- The Device ID of the device that sent the m.key.verification.start message
- The public key from the m.key.verification.key message sent by the device which sent the m.key.verification.start message
- The Matrix ID of the user who sent the m.key.verification.accept message
- The Device ID of the device that sent the m.key.verification.accept message
- The public key from the m.key.verification.key message sent by the device which sent the m.key.verification.accept message
- The transaction_id

If instead the "key_agreement_protocol" is "curve25519", the info parameter is the concatenation of the following elements, without a separation character[13]:

- The string MATRIX_KEY_VERIFICATION_SAS
- The Matrix ID of the user who sent the `m.key.verification.start` message
- The Device ID of the device that sent the `m.key.verification.start` message
- The Matrix ID of the user who sent the `m.key.verification.accept` message
- The Device ID of the device that sent the `m.key.verification.accept` message
- The transaction_id

[13]According to the specification, this protocol is deprecated and "curve25519-hkdf-sha256" should be used instead

The info parameter is passed into `olm_sas_generate_bytes`, to generate either 5 bytes if the selected SAS method is decimal, or 6 bytes if it is emoji. If the SAS method is decimal, the 5 bytes are used to calculate 3 numbers, which are displayed to the user and are what should be compared between the users. The three numbers are calculated from the bytes b_i using the following equations:

$$(b_0 << 5 \mid b_1 >> 3) + 1000 \tag{1}$$
$$((b_1 \ \& \ 0x7) << 10 \mid b_2 << 2 \mid b_3 >> 6) + 1000 \tag{2}$$
$$((b_3 \ \& \ 0x3F) << 7 \mid b_4 >> 1) + 1000) \tag{3}$$

If the SAS method is emoji, the first 42 bits of the 6 bytes are grouped into 7 groups of 6 bits. Each of these 6-bit packets is interpreted as a number between 0 and 63 (since 6 bits can hold $2^6 = 64$ values), and the corresponding emoji is looked up from a list[14].

m.key.verification.mac

If both users confirm that the SASs match, each device creates a list of keys that they want the other device to verify. This is usually their ed25519 key (signing key), as well as their master cross-signing key — those are also the ones we include in our response. The devices then calculate MACs for each of the keys, as well as the comma-separated, lexicographically sorted list of keys. Both devices then send the list of keys and the calculated MACs to each other:

```
{
    "content": {
        "keys": "+Wj8fzbCb17XexRoYZHztCi3EYVvyE3yklonGWeJOY8",
        "mac": {
            "ed25519:FKSVPCGSUV": "qVXnRQXJpAYgRhKdgC+VOojFSdkrq...",
            "ed25519:vt8tJ5/...": "/OHkJPUVorAY686qNTGuOjTXJgm2y..."
        },
        "transaction_id": "<transaction_id>"
    },
    "type": "m.key.verification.mac"
}
```

[14]`https://github.com/matrix-org/matrix-doc/blob/master/data-definitions/sas-emoji.json`

Each device then calculates the MACs for the given keys and the key list itself. If they match the received MACs, the device is marked as verified, and an `m.key.verification.done` message is sent to conclude verification:

```
{
  "content": {
    "transaction_id": "<transaction_id>"
  },
  "type": "m.key.verification.done"
}
```

4.7 End-to-end Encryption

End-to-end encryption is implemented in two parts, one-on-one communication using the Olm cryptographic ratchet[15] and group communication using the Megolm group ratchet, both of which are implemented in libolm, which is used extensively throughout this library.

4.7.1 Olm

Any device that wishes to create Olm connections with other devices has to create an Ed25519 and a Curve25519 key pair and multiple onetime Curve25519 key pairs and upload the public parts to their homeserver. The Ed25519 key pair uniquely identifies a device and is used to sign messages and other keys to prove that they are actually from this device. The Curve25519 key pair is used in combination with another onetime Curve25519 key pair to create new one-on-one Olm connections. Each device has to ensure that there is a sufficient number of onetime keys stored on the homeserver at any time, as they can only be used once, as the name suggests, and are deleted once they are claimed by another device.

Each `MatrixClient` has one `OlmAccount`, and several `OlmSessionss` with other devices. When trying to send an encrypted message directly to a device, as a to-device message (see **??**), it is checked whether an Olm session with that device was already established. If not, a new one has to be initiated.

For this, the other device's Curve25519 public key is fetched by calling `MatrixClientRequestDeviceKey` and one onetime key is claimed from the homeserver by call-

[15] developed initally for the Signal messenger `https://signal.org/docs/specifications/doubleratchet/`

ing `MatrixClientClaimOnetimeKey`. A new outgoing Olm session is then created by calling `MatrixOlmSessionTo`.

In order to actually establish the connection to the other device, any message can be encrypted using the newly created Olm session and sent as an `m.room.encrypted` to-device message.

The `m.room.encrypted` event is used both for sending Olm as well as Megolm messages and looks different depending on which one it is being used for. When sending Olm messages, it looks like this:

```
{
  "type": "m.room.encrypted",
  "content": {
    "algorithm": "m.olm.v1.curve25519-aes-sha2",
    "sender_key": "<sender_curve25519_key>",
    "ciphertext": {
      "<recipient_curve25519_key>": {
        "type": 0,
        "body": "<encrypted_payload_base_64>"
      }
    }
  }
}
```

The difference is in the content of "ciphertext": For Olm messages, this is a mapping of device Curve25519 public key to an object containing the encrypted, Base64 encoded payload, as well as a "type" field. This "type" field simply indicates whether this is a new Olm connection or one that has already been established, it is 0 if this is the first message sent using that connection and 1 otherwise.

While we can establish an Olm session ourselves, it can also happen that we receive an `m.room.encrypted` message with algorithm set to "m.olm.v1.curve25519-aes-sha2", which means another device is trying to establish a new Olm session. This means they claimed one of our onetime Curve25519 keys from the server and created a new Olm session. In order to create the Olm session on our part, we have to call `olm_create_inbound_session_from`, passing the encrypted ciphertext. This automatically checks the onetime keys registered with the `OlmAccount`. When we successfully decrypted the message, we have to invalidate the used onetime keys using `olm_remove_one_time_keys` and replace it on the server so that we still have the recommended amount of keys available. We should store this session in case we want to send an encrypted message to that device later. That is why it is always a good idea to check existing Olm sessions when trying to send an encrypted to-device message, even if we haven't created one ourselves.

4.7.2 Megolm

Even though we have to handle incoming and outgoing Olm sessions separately, no matter whether we created an outgoing session ourselves or we created an incoming from a type 0 message, we can use an established session to both encrypt and decrypt messages. With Megolm, incoming and outgoing sessions are completely separate, one Megolm session can only be used by one device to send messages and every other device has to create an incoming Megolm session to decrypt them. If another device wants to send an encrypted group message, they have to create their own outgoing Megolm session. So participating in a room that is configured to use end-to-end encryption poses two separate problems:

1. Creating an outgoing Megolm session and sending the session details to other devices so they can decrypt sent messages

2. Receiving/requesting session details for other device's Megolm sessions so we can decrypt their messages

In fact, the second problem itself is somewhat twofold because there is a difference between receiving encrypted group messages at the time they are sent and trying to decrypt messages that were already sent retrospectively.

Sending encrypted group messages involves creating an outgoing Megolm session, sharing the session details with other devices in the room, and then encrypting and actually sending the message. To create an outgoing Megolm session, we can call `MatrixClientNewMegolmOutSession` with the desired room ID. The created session is stored in our `MatrixClient` and is used automatically when we use `MatrixClientSendEventEncrypted` to send a message to the specified room.

We can then share the session key, which can be used to create an incoming Megolm session, with another device by calling `MatrixClientShareMegolmOutSession`. We could loop over all known devices, but usually sending the session details to one device is enough because other devices can send room key requests to that device. An m.room_key message looks like this:

```
{
  "content": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "room_id": "!Cuyf34gef24t:localhost",
    "session_id": "X3lUlvLELLYxeTx4yOVu6UDpasGEVOOJbu+QFnmOcKQ",
    "session_key": "AgAAAADxKHa9uFxcXzwYoNueL5Xqi69IkD4sni8LlfJL7qNBEY..."
  },
  "type": "m.room_key"
}
```

This is sent as an encrypted to-device message. Afterward, when the other devices have created incoming sessions, we can send encrypted messages and they should be able to decrypt them without any further action on our part.

Receiving encrypted group messages can happen in two different ways. Either we are currently logged in and syncing events while the session key is shared and the message is sent, or we are requesting a past message and trying to decrypt it. In the case that we are receiving events when the session is created and the key is being shared, we simply receive an `m.room_key` message, analogous to the procedure described for sharing our own session. We simply decrypt the message, sent as `m.room.encrypted`, using an existing Olm session or by creating a new incoming Olm session from that message.

If we are however trying to decrypt a group message in retrospect, we have to take some additional action. We can ask another device for a session's key by calling `MatrixClientRequestMegolmInSession`, passing the room ID, Megolm session ID, and the Curve25519 key of the device that sent the message, as well as the user ID and device ID of the device that we are requesting the key from. Requesting a room key/Megolm session key from another device involves setting up an Olm session by sending an empty, encrypted `m.dummy` message, followed by the unencrypted `m.room_key_request`:

```
{
  "content": {},
  "type": "m.dummy"
}
```

```
{
  "content": {
    "action": "request",
    "body": {
      "algorithm": "m.megolm.v1.aes-sha2",
      "room_id": "!Cuyf34gef24t:localhost",
      "sender_key": "RF3s+E7RkTQTGF2d8Deol0FkQvgII2aJDf3/Jp5mxVU",
      "session_id": "X3lUlvLELLYxeTx4yOVu6UDpasGEVO0Jbu+QFnm0cKQ"
    },
    "request_id": "1495474790150.19",
    "requesting_device_id": "RJYKSTBOIE"
  },
  "type": "m.room_key_request"
}
```

Even though this is sent unencrypted, we need the Olm session because the `m.forwarded_room_key` message that contains our requested session key is sent as an encrypted to-device message. Decrypted it looks like this:

```
{
  "content": {
    "algorithm": "m.megolm.v1.aes-sha2",
    "forwarding_curve25519_key_chain": [
      "hPQNcabIABgGnx3/ACv/jmMmiQHoeFfuLB17tzWp6Hw"
    ],
    "room_id": "!Cuyf34gef24t:localhost",
    "sender_claimed_ed25519_key": "aj40p+aw64yPIdsxoog8jhPu9i7...",
    "sender_key": "RF3s+E7RkTQTGF2d8Deol0FkQvgII2aJDf3/Jp5mxVU",
    "session_id": "X3lUlvLELLYxeTx4yOVu6UDpasGEVO0Jbu+QFnm0cKQ",
    "session_key": "AgAAAADxKHa9uFxcXzwYoNueL5Xqi69IkD4sni8Llf..."
  },
  "type": "m.forwarded_room_key"
}
```

It contains the same information about a Megolm session as an m.room_key would, but includes three additional fields:

`forwarding_curve25519_key_chain` Contains a chain of Curve25519 keys of all the devices that forwarded this key, so if device B receives a Megolm key from device A and then forwards it to device C, it includes device A's key. If device C then forwards it again, it includes device B's key.

`sender_claimed_ed25519_key` The Ed25519 key of the device which created the session, "claimed" because this cannot be verified by the receiving device.

`sender_key` The Curve25519 key of the device that created the session, not needed with `m.room_key` because the message is received directly from the device that created the session

5 Running on the ESP32

The ESP32 is a family of microcontroller units (MCUs) designed and manufactured by Espressif. A number of different variants are sold, for example, the original ESP32 uses an Xtensa CPU, but there are also RISC-V-based models with the ESP32-C and the ESP32-H series. Since they are cheap, relatively powerful and provide Wi-Fi and Bluetooth off the shelf they have become an increasingly popular choice of MCU. This also makes them a good candidate for this project, since they support Wi-Fi for communicating with a Matrix server, and have enough performance and memory to actually support a library such as libolm and HTTPS at the same time. Most of the testing was done on a base model ESP32, with some later tests also running on a RISC-V-based ESP32-C6.

5.1 Build Environment

Two popular choices for programming the ESP32 are the Arduino IDE[16], for which Espressif provides official support[17], and the ESP integrated development framework[18] (ESP-IDF), which is developed directly by Espressif. ESP-IDF was used exclusively throughout this thesis. It provides both a set of tools to build, flash, and monitor code, as well as APIs to make accessing features such as WiFi easier.

In order to run an application using the Matrix library on the ESP32, the following steps had to be taken:

1. Create a new ESP-IDF project
2. Add the main program
3. Add dependencies

The process was relatively straightforward. Instead of creating a new project, I copied one of the included examples and added the code to it. After replacing the example source code with the one that I wanted to run, the only change that had to be made to the code was adding the entry point that ESP-IDF uses and setting up the WiFi connection. The code to connect the WiFi was taken from another example and placed into the files `wifi.h/c`, resulting in the following addition to the code:

```
#include "wifi.h"
```

[16]https://www.arduino.cc/en/software
[17]https://github.com/espressif/arduino-esp32
[18]https://www.espressif.com/en/products/sdks/esp-idf

```
void
app_main(void)
{
    // initialize wifi
    wifi_init("<SSID>", "<PASSWORD>");

    // call original main
    main();
}
```

But before the code could run, two dependencies had to be supplied; the Matrix library itself and libolm. They were added as "components"[19], which is how ESP-IDF handles external code. Creating a component was as simple as creating a new directory in the project's `components` directory with a `CMakeLists.txt` inside. The `CMakeLists.txt` simply contains source files, include paths, compiler flags, and ESP-IDF dependencies that are used. The component for adding the Matrix library looks like this:

```
idf_component_register(
    SRCS
        "../../../../src/matrix.c"
        "../../../../src/matrix_http_esp32.c"
        "../../../../ext/mjson/src/mjson.c"
    INCLUDE_DIRS
        "../../../../ext/olm/include"
        "../../../../ext/olm/lib"
        "../../../../ext/mjson/src"
        "../../../../src"
    REQUIRES
        esp-tls
        esp_http_client
        esp_netif
        nvs_flash)

SET(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -fpermissive")
```

[19] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html#component-cmakelists-files

In addition to the Matrix source files, mjson[20] is added and include paths for mjson and libolm are specified. Necessary ESP-IDF components are listed under "REQUIRES", this includes esp_http_client for the HTTP API and nvs_flash which is used by the code that establishes a WiFi connection.

5.2 Memory

There were two major issues when trying to communicate with a Matrix server from the ESP32. The first one is memory, the second one is enabling the library to send HTTP requests. The base model ESP32 has 512 KB of RAM and about 4 MB of ROM. The RAM is divided into Data RAM (DRAM) and Instruction RAM (IRAM), with the DRAM taking up 320 KB and the IRAM taking up the remaining 200 KB. Only about half of the available DRAM can be allocated statically and the rest is available as the heap at runtime. When developing the library, I tried to declare any larger amounts of data, such as buffers for sending/receiving messages, as static to make sure there was enough memory for them at runtime and avoid overflowing the stack. Since there is a relatively large amount of data statically allocated by the library, and because libolm as a dependency occupies 30 KB of DRAM on its own, the amount of static data had to be reduced in order for the library to compile and work correctly on the ESP32.

Since the ESP32 is based on a Harvard architecture CPU, data and instructions are separate in RAM[10]. In total there are six memory types on the ESP32:

- Data RAM
- Instruction RAM
- Instruction ROM
- Data ROM
- RTC Slow Memory
- RTC Fast Memory

Data RAM

The data RAM houses the application's .data and .bss sections, which are used to store non-constant and zero-initialized static data respectively. These two regions together cannot exceed 160 KB due to technical limitations, the remaining DRAM is available at runtime as heap memory.

[20]`https://github.com/cesanta/mjson`

Instruction RAM

Of the 200 KB of available IRAM, the first 64 KB serve as MMU caches. The remaining IRAM is used for code which should run from RAM. While most code is stored in ROM, interrupt handlers and code that is especially time-critical are placed in RAM. The reason for this is that loading code from ROM may cause delays if the load results in a cache miss. So the compiler places some parts of the code in RAM automatically, and we can explicitly specify code to be stored in RAM either by instructing the linker to place single functions or entire components in RAM via a linker script, or by marking functions using the `IRAM_ATTR` macro.

ESP-IDF provides an (experimental) option to place the entire application in RAM, which significantly reduces the amount of code we can use, but also speeds up execution, and more importantly makes execution time more deterministic because code doesn't have to be loaded from ROM, which can be unreliable depending on whether the code that has to be loaded is in cache or not. This was not an option for this project though. While code size doesn't prove a challenge when the majority of the application is placed in ROM, it becomes a significant factor when trying to fit the whole application into RAM. Libolm is written mostly in C++, and therefore parts of an implementation of the C++ standard library (libstdc++ in this case) are included in the compilation. While this only includes the parts that are used, those still add around 200 KB in code size, so placing the whole application in RAM was not an option.

Instruction ROM

The majority of an application, basically all of the parts not explicitly placed in RAM, are placed in ROM by default. To improve performance, the memory regions containing the application's code are mapped to the instruction space. The ROM accessed this way is cached in RAM, so when there aren't too many cache misses performance when loading instructions should be similar to loading them from RAM.

Data ROM

In contrast to RAM, ROM isn't actually divided into separate regions for instructions and data. The regions are simply mapped differently by the MMU. Data stored in ROM has to be constant (since, as the name implies, ROM is read-only). All constant data is placed in ROM by default, with the exception of literal constants, which are stored next to the containing function's code in the regions used for storing instructions. Constants can be explicitly stored in ROM by declaring them with the `DRAM_ATTR` macro.

RTC Slow/Fast Memory

Realtime clock (RTC) memory can be used to store data during deep sleep and make sure it is still available after waking up. RTC Fast memory can store both instructions and data, and if it is not needed for preserving data during sleep it can be used as RAM if code isn't run on more than one core. Code that runs from RTC Fast memory can access variables from both Slow and Fast memory, but global and static variables that are accessed from RTC memory have to be placed into RTC Slow memory. Variables can be placed in RTC memory using the `RTC_DATA_ATTR` macro.

5.2.1 Optimizing Memory Usage

The most memory was taken up by the following components, each of which had to be reduced somehow before the library could function correctly:

- Buffers for formatting, encrypting, decrypting, sending, and receiving messages
- Structs from libolm containing information about sessions and device keys

Since the libolm structs themselves could not be made any smaller, the amount of them that were used had to be reduced. The largest struct that is used is `OlmAccount` at 7528 bytes, which stores the device keys including onetime keys. This is only used once though and its use is not optional. The next largest struct is `OlmSession` at 3352 bytes, which represents a single one-on-one Olm session with another device. Initially, memory for 10 sessions was allocated to allow for sending and receiving to-device messages from multiple devices. This ended up not being necessary since a new Olm session can be initiated anytime. The first message sent can be read immediately, there is no extra message needed to establish a session. It would be more efficient to reuse Olm sessions for devices with which sessions were already established in the past, but a lot of DRAM can be saved at the cost of having to start a new Olm session every time encrypted to-device messages have to be sent/received by a different device.

The other big source of memory usage was buffers used for formatting events, storing encrypted or signed copies of events, or sending/receiving messages. These were typically declared statically inside of the functions they were used in, which ensured the same buffer would not be accessed from two different places at the same time (unless a function recursively called itself, which none of them do), but also means there is a unique buffer for every function that is implemented. Since some of the messages that are exchanged are relatively large, and sometimes there have to be signed/encrypted copies of buffers, these add up quickly. In order to reduce the amount of memory used by these buffers, the largest and most frequently used ones were refactored into global variables. This works because the same buffer is not used by any two functions that call each other.

5.3 HTTP Requests

On desktop platforms, Mongoose is used for making HTTP requests. Mongoose was chosen because it is small and supports both desktop environments and embedded devices. And while it compiled without problems a connection couldn't be established when first trying to run on the ESP32. Since the HTTP layer is very simple, I decided to add a dedicated implementation for the ESP32 to ensure that any connection issues were not caused by Mongoose. The ESP version is using the `esp_http_client` interface provided by ESP-IDF. This section will quickly walk through the different parts making up the HTTP layer, as described in 4.2, to highlight the implementation specifics.

`struct MatrixHttpConnection`

> The main data structure used in the HTTP layer. This is defined differently for each implementation because it includes the platform-specific context which is passed to the different HTTP interfaces. For the ESP32 this is a `esp_http_client_handle_t`. It also commonly includes copies of the server URL and access token, as well as a pointer to a data buffer used for copying responses.

`MatrixHttpInit/Deinit`

> `MatrixHttpInit` is passed a `MatrixHttpConnection**` and a host, which is the base URL of the Matrix server, as a C-style string. The address of a `MatrixHttpConnection*` is passed because the pointer is allocated in the init function, using `calloc` to ensure all members are set to zero. The host is set in the newly allocated `MatrixHttpConnection*` and it is then passed to `MatrixHttpConnect`.
>
> `MatrixHttpDeinit` is passed only a `MatrixHttpConnection**`. After calling `MatrixHttpDisconnect`, the `MatrixHttpConnection*` is free'd, and set to NULL, which is again why we pass the address of a `MatrixHttpConnection*`.

`MatrixHttpConnect/Disconnect`

> The connect/disconnect functions aren't actually part of the public HTTP API and are only used internally by `MatrixHttpInit/Deinit`. `MatrixHttpConnect` creates a `esp_http_client_config_t` that is needed to initialize the `esp_http_client`.

During initialization, you have to pass an initial target for requests by specifying either the `.url` or both `.host` and `.query` members of the config. Since the URL is always updated before making any actual requests, this is initialized with the address of the Matrix server, since this is always available during initialization. On connect the timeout is also set to 10 seconds, because in testing the ESP's connection wasn't always stable enough to receive a response immediately and requests would fail because of it.

`MatrixHttpDisconnect` frees the HTTP client by calling `esp_http_client_cleanup` and the pointer is set to NULL to avoid use after free issues.

`MatrixHttpGet/Post/Put`

All three HTTP methods are implemented in a very similar way. All of them are passed the following parameters:

`MatrixHttpConnection * hc`

Pointer to a `MatrixHttpConnection` that contains implementation-specific information about the connection, in this case a `esp_http_client`, as stated above.

`const char * path`

The path to perform the HTTP request on, not including the host since this is always the Matrix server that was set in `MatrixHttpInit`.

`char * outResponseBuffer`

String buffer into which the response should be written.

`int outResponseBufferCap`

Size of the response buffer.

`bool authenticated`

Whether or not the API call has to be authenticated using an access token, for example the login request cannot be authenticated because we have not received an access token yet.

Additionally, `MatrixHttpPost/Put` also require a `const char * requestBuffer` that contains the request body to be sent. The basic procedure for all methods is as follows:

1. If `authenticated` is true, create the authorization header using the access token set in the `MatrixHttpConnection*`.

2. The response buffer, including its capacity, is set in the `MatrixHttpConnection*`. The data length is set to 0 since the response might arrive in multiple parts and the length is used to keep track of what was already written for a particular request.
3. The request URL is set to the host followed by the passed path.
4. The appropriate HTTP method (GET/POST/PUT) is set.
5. A "Content-Type" header is set to "application/json", and if applicable, the "Authorization" header is set to "Bearer <access_token>".
6. For POST and PUT requests, the request body is set.
7. The request is performed by calling `esp_http_client_perform`. The response is handled by `_http_event_handler`, as detailed below.

`_http_event_handler`

This is the callback that handles responses to connection attempts and requests. Most events are simply logged, only `ON_DATA` and `DISCONNECT` are actually handled. On disconnect it checks whether the disconnect was caused by any failure and if that's the case outputs the error that caused it. On data, it checks the maximum amount of data that can be copied. If the provided buffer has less memory available than is needed by the received response the response is truncated. The buffer is null-terminated and its length is updated.

The event handler calls `vTaskDelay()` with a small value, currently 10 ms. Depending on the connection speed and the set timeout value, sending an HTTP request can take several seconds. Applications developed using ESP-IDF run on FreeRTOS, which creates a main task for the application and an idle task that performs some maintenance work. If the main task runs for too long without calling any function that allows the idle task to run, such as blocking IO functions, FreeRTOS will throw an error, and so we call `vTaskDelay()` explicitly when waiting for a response to allow the idle task to run.

6 Evaluation

Three different things will be evaluated: The general functionality of the library, how it runs on different platforms, and how energy efficient it runs, with the latter being the main focus of the evaluation.

The library can be used to log in to a Matrix server, get the newly created device verified, and then send and receive encrypted messages. This can be accomplished with relatively little code:

```c
// Create and initialize a new MatrixClient and set the user ID
MatrixClient * client = (MatrixClient*)malloc(sizeof(MatrixClient));
MatrixClientInit(client);
MatrixClientSetUserId(client, "<user ID>");

// Initialize HTTP API
MatrixHttpInit(&client->hc, SERVER);

// Login
MatrixClientLoginPassword(client,
    "<username>",
    "<password>",
    "<device ID>");

// Upload device and onetime keys
MatrixClientGenerateOnetimeKeys(client, 10);
MatrixClientUploadOnetimeKeys(client);
MatrixClientUploadDeviceKeys(client);

// Create a buffer that will hold the sync response
#define SYNC_BUFFER_SIZE 1024
STATIC char syncBuffer[SYNC_BUFFER_SIZE];

// Automatically handle events until verification was
// started by another device and finished successfully
while (! verified) {
    MatrixClientSync(client, syncBuffer, SYNC_BUFFER_SIZE);
}

// Create and share an outgoing Megolm session
MatrixMegolmOutSession * megolmOutSession;
MatrixClientNewMegolmOutSession(client,
    "<room ID>",
    &megolmOutSession);
```

```
MatrixClientShareMegolmOutSession(client,
    "<user ID>",
    megolmOutSession);

// At this point we are verified and can send and receive E2EE messages
while (true) {
    // This handles incoming encrypted messages automatically
    MatrixClientSync(client, syncBuffer, SYNC_BUFFER_SIZE);

    // Send an E2EE message to the specified room
    MatrixClientSendEventEncrypted(client,
        "<room ID>",
        "m.room.message",
        "<message>");
}

// Delete the device that was created
// on login if we don't need it anymore
MatrixClientDeleteDevice(client);

// Clean up HTTP API
MatrixHttpDeinit(&client->hc);
```

Listing 1: Minimal example using the Matrix client library

This is a complete example, messages sent this way can be decrypted by other devices and encrypted messages that are received will be decrypted successfully. Loading older encrypted messages and requesting the Megolm session details to decrypt them does not work properly. We receive the encrypted message and we can send the proper session key request and receive a response, but the included session details could not be used to decrypt the message in testing. This is in contrast to messages sent after our device was created, the initial message containing the Megolm details that the sending device sends is received and can be used to create the correct Megolm session that can decrypt messages sent by that device. Verification works as well, although it has to be initiated by a different device. The corresponding events are handled by `MatrixClientSync`. This was especially important because devices will only share Megolm session details with devices that were successfully verified[21].

The same code runs on an ESP32 as well. It can be compiled using ESP-IDF, the only code that has to be added is the ESP-IDF-specific application entry function `void app_main()`. It was tested on two base model ESP32, an ESP32-WROOM-32,

[21] At least when using the Element Web client

and an ESP32-CAM, as well as a RISC-V-based ESP32-C6. Login, verification, and sending/receiving worked on all devices.

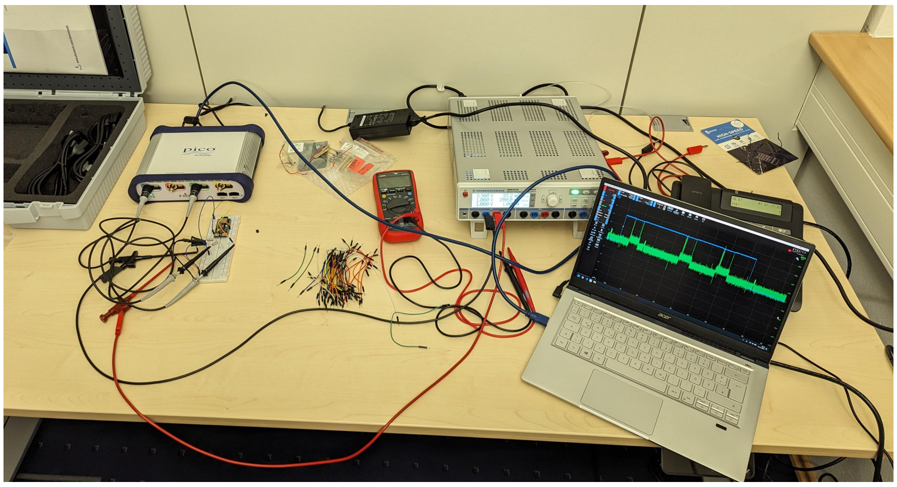


Figure 5: Setup used for measuring power consumption

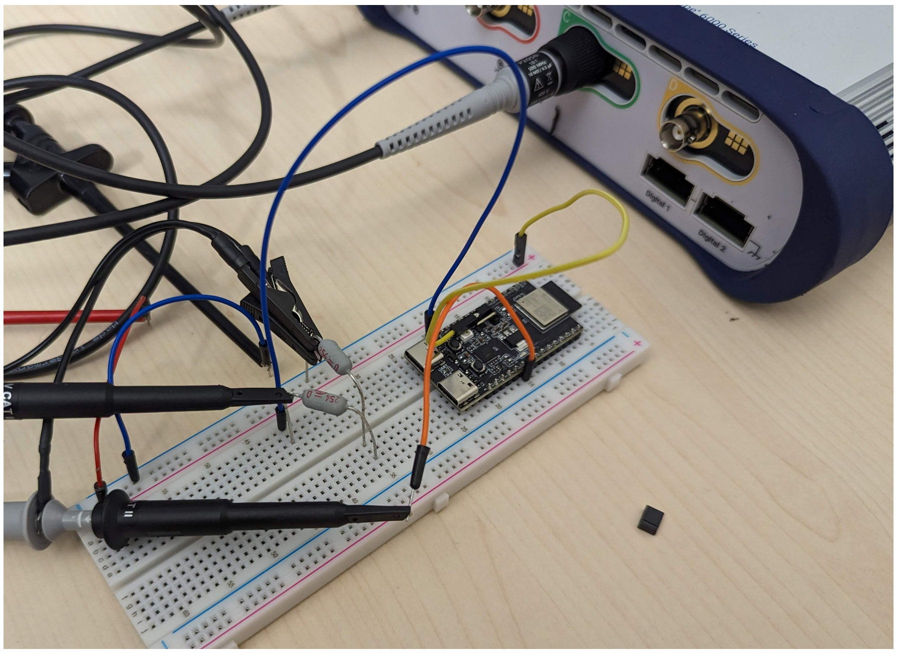


Figure 6: Close-up of the ESP32-C6 connected to the power supply and oscilloscope

The power consumption was tested using the ESP32-C6. The setup used for evaluation consisted of the ESP32 itself, an external Rohde & Schwarz HMP2030 power supply, and a PicoScope 6405E oscilloscope. The setup can be seen in Figure 5, with a close look at the ESP's wiring in Figure 6.

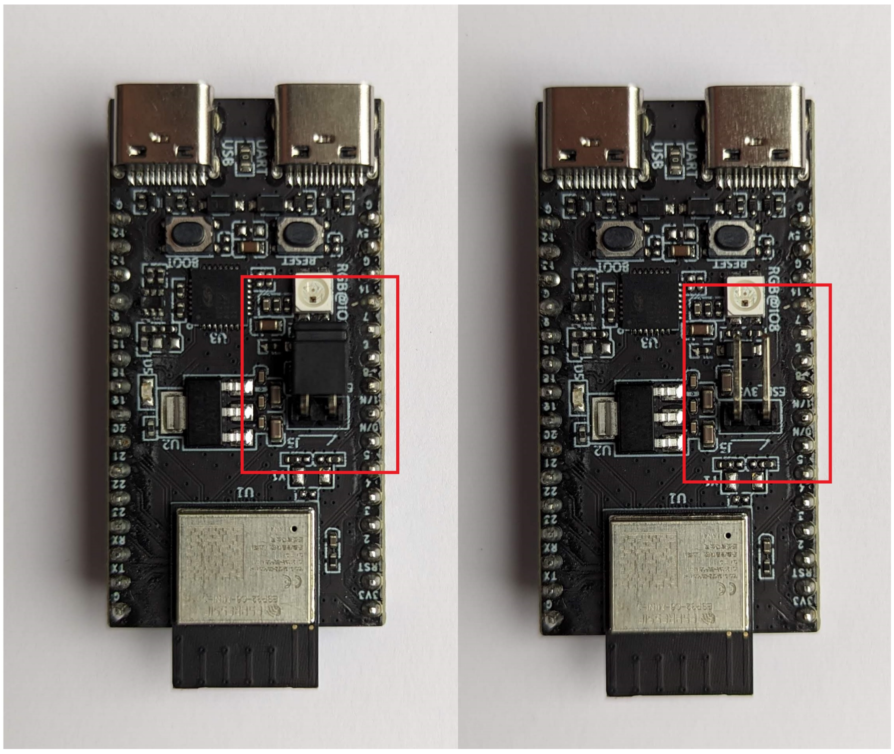


Figure 7: ESP32-C6 with and without the jumper on the J5 headers

This ESP32 model has a set of J5 headers that can be used to power the MCU directly, bypassing the board's peripherals. They can be seen in Figure 7 with the jumper on the left and without the jumper on the right. Removing the jumper cuts off the power supply to the rest of the board, which makes it possible to measure the power the chip would draw if it was used directly without a development kit.

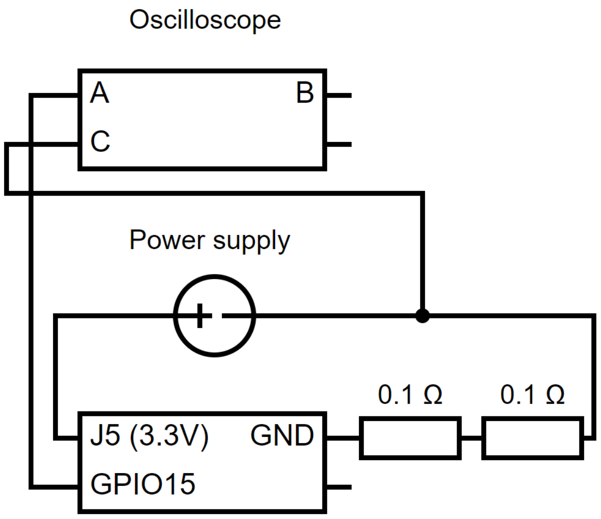


Figure 8: Wiring of the test setup

Figure 8 shows a diagram of the wiring. The power supply is connected directly to the 3.3V J5 header and to ground through two 0.1 Ohm resistors. The oscilloscope is connected to the circuit between the ESP32 and ground, and is also connected to GPIO pin 15 of the ESP which is pulled high when sending a message to tell the oscilloscope to record data.

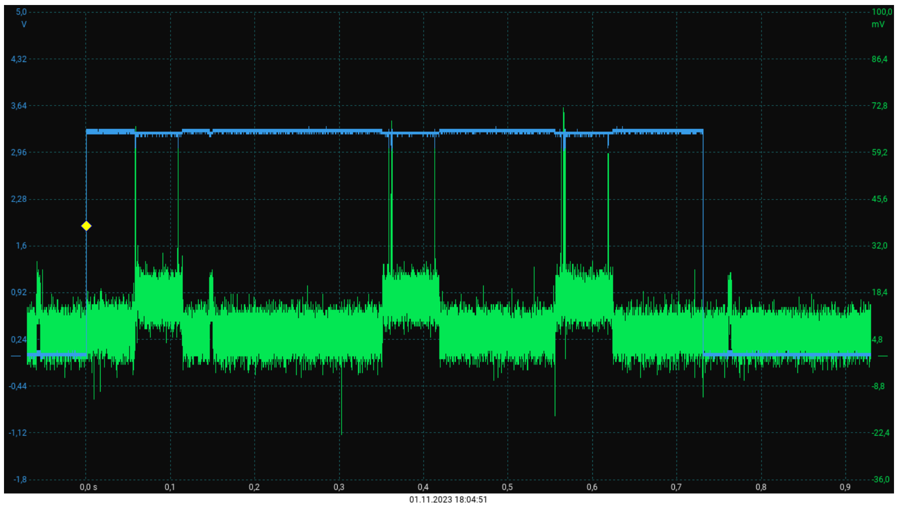


Figure 9: Voltage reading while sending an encrypted message

The measurement was done while sending encrypted group messages, with the ESP32 running at 80 MHZ from a supplied voltage of 3.3V. The result of one of the measurements can be seen in Figure 9. The blue signal is the trigger sent from the ESP32 which tells the oscilloscope to start recording data, it is pulled high and so goes from 0 to 3.3V. The green signal is the measured voltage. It shows a baseline power draw of 8.538 mV, with three spikes of around 16.5 mV. This measurement was taken with the two resistors as shown in Figure 8. Considering the resistance of 0.2 Ohm we can calculate the current during and between sends:

$$I = \frac{0.008538V}{0.2\Omega} = 0.04269A \tag{4}$$

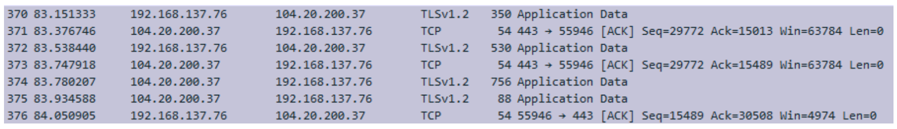
$$I = \frac{0.0165V}{0.2\Omega} = 0.0825A \tag{5}$$

Assuming a constant voltage of 3.3V we can also estimate the power draw:

$$P = 3.3V \times 0.04269A = 0.140877W \tag{6}$$

$$P = 3.3V \times 0.0825A = 0.27225W \tag{7}$$

When comparing the readings with the traffic recorded via Wireshark[22] (see Figure 10), we can see that the spikes correspond to two messages sent by the client and one sent by the server. Lines 1 and 3 show packets sent from 192.168.137.76, which is the ESP32, while lines 5 and 6 show a single packet received by the ESP. Lines 2, 4, and 7 contain the corresponding acknowledgments.



```
370 83.151333    192.168.137.76    104.20.200.37     TLSv1.2   350 Application Data
371 83.376746    104.20.200.37     192.168.137.76    TCP        54 443 → 55946 [ACK] Seq=29772 Ack=15013 Win=63784 Len=0
372 83.538440    192.168.137.76    104.20.200.37     TLSv1.2   530 Application Data
373 83.747918    104.20.200.37     192.168.137.76    TCP        54 443 → 55946 [ACK] Seq=29772 Ack=15489 Win=63784 Len=0
374 83.780207    104.20.200.37     192.168.137.76    TLSv1.2   756 Application Data
375 83.934588    104.20.200.37     192.168.137.76    TLSv1.2    88 Application Data
376 84.050905    192.168.137.76    104.20.200.37     TCP        54 55946 → 443 [ACK] Seq=15489 Ack=30508 Win=4974 Len=0
```

Figure 10

[22]`https://www.wireshark.org/`

7 Conclusion

The initial goal of this thesis was to implement a library that can be used to communicate with a Matrix server, using end-to-end encryption and that works on both desktop operating systems and embedded devices. In addition to running on a microcontroller, the energy consumption should also be taken into consideration, to get an idea of how high the power cost would be for exchanging encrypted data this way.

The library can be used to communicate with a Matrix server, either by taking an access token acquired from some other client or by logging in using a username and password. Messages can then be sent, and they can be received by explicitly requesting a single message by message ID, or by continually synchronizing with the server. End-to-end encryption works mostly as intended. Encrypted group messages can be sent and are correctly decrypted and displayed by other clients. Encrypted messages by other devices can be decrypted upon receiving, only encrypted messages that were sent before the device could have received them could not be decrypted. Verification by another device works, although it is not possible to initiate verification of other devices using the library.

All of this works on a PC as well as on an ESP32. The code was tested with different ESP32 models, but since RAM was the limiting factor when trying to get it to work, and there are no ESP32 models with less RAM available than the ones used in testing, it can be assumed that this should work on any ESP32 or comparable MCU. Power consumption was measured, but no optimization was done.

Another more subjective goal was to make the library easier to work with than previous attempts. The code in listing 1 shows that the library makes it possible to call Matrix Client-Server APIs similar to how they are defined in the specification. There is no abstraction, if someone understands how the Matrix specification works, the library allows them to interact with it in a natural way.

Apart from the issues with decrypting past group messages, the goals of this thesis were largely achieved. With more time, more measurements of the impact of encryption using libolm and TLS through HTTPS could have been done, and it could have been investigated how much the power consumption could be optimized. The functionality that the library provides works pretty much like intended, but more features like creating or joining rooms could be added, although given the fact that most of these features would simply mean adding one or two HTTP requests, many of these features could be implemented easily and so were not a major focus.

References

[1] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[2] The Matrix.org Foundation. Megolm group ratchet. `https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md`. Accessed 21. October 2023.

[3] The Matrix.org Foundation. Olm: A Cryptographic Ratchet. `https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md`. Accessed 21. October 2023.

[4] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[5] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[6] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[7] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[8] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. `https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf`. Accessed 30. October 2023.

[9] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[10] Espressif Systems. ESP32 Technical Reference Manual. `https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf`. Accessed 13. November 2023.