NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

JOÃO FILIPE CARVALHAL SOARES
Graduate in Computer Science

KEY AGREEMENT FOR DECENTRALIZED SECURE INTRUSION DETECTION

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
January, 2024

DEPARTMENT OF
COMPUTER SCIENCE

KEY AGREEMENT FOR DECENTRALIZED SECURE INTRUSION DETECTION

JOÃO FILIPE CARVALHAL SOARES

Graduate in Computer Science

Adviser: João Resende
Assistant Professor, NOVA University Lisbon

Co-adviser: João Leitão
Full Professor, NOVA University Lisbon

Examination Committee

Chair: Bernardo Toninho
Assistant Professor, FCT-NOVA

Members: João Resende
Assistant Professor, FCT-NOVA

Fábio Gonçalves
PosDoc, Universidade do Minho - Centro Algorítmico

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
January, 2024

Key Agreement for Decentralized Secure Intrusion Detection

Copyright © João Filipe Carvalhal Soares, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.10) [61].

Acknowledgements

I would like to first thank João Resende, my main adviser for this thesis, for his constant overlook and support on my work, continuously providing helpful input and guidance, with a permanent good mood. Although we met few times in person, he was always present and a tremendous help in achieving the desired objectives and overcoming the many obstacles that presented themselves along the way.

Secondly, I would like to thank my good friend Pedro Custódio, a fellow IT man, even though he graduated from the "rival" college, for the ever insightful and plentiful help he gave me throughout all these months of hard work. His experience in the cyber-security and networking fields, matched with his kindness and availability, was an invaluable ally in the completion of this thesis work.

Finally, an enormous thanks to my parents, and the rest of my family, for worrying, and giving me helpful reminders periodically to not give in to complacency, which were much needed at times. Managing my time during these months was not always an easy task, and my parents helped tremendously in keeping me in check and making sure I did not lose strength, while still maintaining some personal time. The same can be said for my friends, and thanks are in order for the constant emotional support.

Abstract

As the digital world is taking more significant roles in people's lives, empowering interconnection and integration in physical and cyberspaces is a requirement. As more organizations adopt an Internet of Things (IoT) system, a faulty implementation could pose tremendous risks to several parties by defying fundamental security principles. Such intelligent environments empower connectivity and heterogeneity across multiple industries, from financial fields to healthcare and fault detection. Therefore, ensuring reliable services and communications is a top priority.

Intrusion Detection Systems (IDS) are systems designed to monitor and detect anomalies and privacy violations. However, the existing methods for this rely on inadequate and ill-prepared security measures that cannot keep up with the real-time and constantly evolving scenario.

Consequently, the intent of the work developed for this thesis is the creation or adaptation, if there are adequate options already in existence, of a decentralized group key agreement protocol for a distributed messaging system in an IoT context, capable of ensuring that communications in such types of systems can be made both secure and efficiently, with support for network fault tolerance and subsequent offline communication.

Keywords: IoT, Security, Decentralized, Communication, Intrusion, Detection, Offline, Availability

Resumo

À medida que as tecnologias digitais se inserem cada vez mais no dia-a-dia do cidadão comum, a potenciação da conexão e integração, tanto física como virtualmente, torna-se uma necessidade. Nesse sentido, à medida que surgem mais sistemas de IoT, uma implementação menos rigorosa destes sistemas pode significar riscos tremendos para todos os envolvidos, ao ignorar propriedades fundamentais de cibersegurança. Este tipo de ecossistemas informáticos estabelecem a conectividade e suportam a heterogeneidade de várias indústrias, desde o campo financeiro ao da saúde, até à deteção de falhas. Assim, assegurar a segurança e fiabilidade das comunicações neste tipo de sistemas é uma prioridade absoluta.

Sistemas de Deteção de Intrusos (SDI) possuem a função única de monitorizar e detetar anomalias e violações de privacidade. Contudo, os métodos existentes para este propósito assentam em bases e políticas de segurança inadequadas para o contexto em questão, que não conseguem acompanhar este ambiente que está em constante evolução e desequilíbrio.

Consequentemente, o trabalho desenvolvido para esta tese consiste na criação, ou adaptação, no caso de já exisitirem opções adequadas para o efeito, de um protocolo descentralizado de acordo de chaves de grupo para um sistema de mensagens inserido num contexto de IoT. Este sistema será capaz de assegurar que as comunicações deste tipo de sistemas podem ser efetuadas de forma segura, e eficiente, com suporte para falhas de rede, permitindo aos clientes comunicar independentemente da sua conexão, garantindo a sua máxima disponibilidade ao sistema.

Palavras-chave: IoT, Segurança, Comunicação, Descentralizada, Deteção, Intrusão, Offline, Disponibilidade

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Goals . . . 3

2 Background - Concepts 4

2.1 Group Communication Protocols Fundamental Concepts . . . 4
2.2 Fundamental Security Concepts . . . 5
2.3 Summary . . . 7

3 Literature Review 8

3.1 Group Communication Protocols - A Security Overview . . . 8
3.1.1 Signal Groups . . . 9
3.1.2 Sender Keys (Whatsapp) . . . 9
3.1.3 Matrix . . . 10
3.1.4 MLS(TreeKEM) . . . 10
3.1.5 *Re-randomized* TreeKEM . . . 11
3.1.6 Causal TreeKEM . . . 11
3.1.7 Concurrent TreeKEM . . . 11
3.1.8 DBGK . . . 12
3.1.9 Baal . . . 12
3.1.10 Decentralized Group Key Management for Dynamic Networks Using Proxy Cryptography . . . 12
3.1.11 DLGKM-AC . . . 13
3.1.12 DCGKA . . . 13
3.1.13 Summary . . . 14

3.2 Group Communication Protocols - Distributed Computing and Message Dissemination . . . 14
3.2.1 Matrix . . . 14
3.2.2 DBGK . . . 22
3.2.3 DLGKM-AC . . . 27
3.2.4 Protocol Shortcomings . . . 34
3.2.5 DCGKA . . . 35
3.2.6 Conclusion . . . 41
3.3 Preparation . . . 42

4 System Model and Architecture **44**
4.1 Use case and context-dependent constraints . . . 44
4.2 Architecture . . . 44
4.2.1 Server . . . 45
4.2.2 System Communication . . . 46
4.3 Adversary Model . . . 49
4.4 Summary . . . 50

5 Implementation and Prototype **52**
5.1 Prototype . . . 52
5.1.1 Client SDK . . . 52
5.1.2 Interface . . . 53
5.1.3 Storage . . . 53
5.2 Implementation - Code . . . 53
5.2.1 Code Organization - Modules . . . 53
5.2.2 Challenges . . . 54
5.2.3 Solutions . . . 57
5.3 Summary . . . 70

6 Experimental Evaluation **72**
6.1 Testing environment . . . 72
6.1.1 Server . . . 73
6.1.2 Clients . . . 73
6.1.3 Network . . . 73
6.2 Methodology . . . 74
6.3 Test Cases . . . 74
6.3.1 Test Case: Regular Communication . . . 75
6.3.2 Test Case: One Offline Client . . . 75
6.3.3 Test Case: Both clients have unstable connections . . . 77
6.4 Results . . . 78
6.4.1 Regular Communication . . . 79
6.4.2 One Offline Client . . . 79

6.4.3 Both clients with unstable connections 80
6.5 Summary 80

7 Conclusions **81**
7.1 Limitations 82
7.2 Main Contributions 84
7.3 Future Work 84

Bibliography **86**

Appendices

Annexes

I Matrix Encryption Primitives **94**
I.1 Algorithms 94
I.2 Megolm 94

II Matrix Out-of-band Verification - SAS protocol **97**

List of Figures

2.1 Group Key Establishment Architecture 5

3.1 Scheme detailing how data flows between Matrix clients 15
3.2 Matrix Key Distribution 18
3.3 Alice establishes a Megolm channel (a) and sends a ciphertext (b) 19
3.4 Diagram with a Matrix event example 21
3.5 DBGK network model: a decentralized architecture based on an independent group key per area 22
3.6 DLGKM-AC system model 29
3.7 Full DCGKA protocol specification 38

4.1 The proposed system architecture for the prototype 45
4.2 The organization of the server component in each group 46
4.3 Schema defining how the Matrix clients and the server communicate 47
4.4 Example for an mDNS host discovery service with a) the query broadcasted to the system and b) the response for that query, returned by every other client in the network 48
4.5 TLSv1.3 Handshake protocol used by libp2p 48

5.1 Component diagram of the application, showing the interaction between the different modules 55
5.2 Protocol developed for client offline communication for the case where the offline client does not have the Megolm session key in storage 70

6.1 Example of the execution of a command in the client containers, with the interface shell also displayed 75
6.2 Diagram visualizing the tests for the regular communication pattern of the prototype 76
6.3 Diagram for the second test case, where one of the client containers cannot connect to the server 77

6.4 Diagram for the test case where the two clients have unstable internet connections 78

II.1 A sequence diagram summarising the SAS protocol. The contents of *m.key.verification.mac* are described in Fig.II.3, while pseudocode descriptions of SAS.SendMAC, SAS.VerifyMAC and SAS.SignDevice are in Fig.II.2. Message types in the above diagram have had the prefix *m.key.verification.* removed. 98
II.2 Algorithms to generate, verify and process m.key.verification.mac messages in the SAS protocol. UserVerified signs the given user's master cross-signing key with the current device's user-signing key. Similarly, DeviceVerified signs the given device's fingerprint and Olm identity keys with the current device's self-signing key. These signatures are uploaded to the homeserver. 99
II.3 The format of an *m.key.verification.mac* message for a user with cross-signing setup 99

List of Tables

3.1 Summary of reviewed protocols' security guarantees 9
3.2 DBGK protocol terminology 24

6.1 Summary of the tests' results 78

1 Introduction

As the digital world is taking more significant roles in people's lives, empowering interconnection and integration in physical and cyberspaces is a requirement. As more organizations adopt an Internet of Things (IoT) system, a faulty implementation could pose tremendous risks to several parties by defying fundamental security principles. Such intelligent environments empower connectivity and heterogeneity across multiple industries, from financial fields to healthcare and fault detection. Therefore, ensuring reliable services and communications is a top priority.

1.1 Context

With the rapid expansion of the IoT ecosystem, the number of malicious cybercriminals targeting the IoT's systems' end nodes, users, servers or all of them has also increased. This unwarranted attention has led to the development and theorizing of a specific type of system known as Intrusion Detection Systems (IDS). Intrusion Detection Systems (IDS) are designed to monitor systems and detect anomalies and privacy violations, and they vary based on the detection technique, validation strategy, and deployment strategy.

In recent years, there have been numerous studies and surveys made regarding these types of systems, and their approaches to the problem at hand. Studies such as the ones done by Khraisat et al.,[54], focus on some of the existing solutions for Intrusion Detection Systems, evaluating their efficiency when tackling the problems in question, and the techniques that are used, among others.

Additionally, Artificial Intelligence (AI) advancements such as machine learning and deep learning, have been utilized to enhance IoT Intrusion Detection Systems. Numerous related studies applied different machine learning and deep learning techniques through various datasets to validate the development of IoT IDS (such as the one performed at [24]. However, it is still not clear which dataset, machine learning or deep learning techniques are more effective for building an efficient IoT IDS. Secondly, the time consumed in building and testing IoT IDS is not considered in the evaluation of some IDS techniques, despite being a critical factor for the effectiveness of 'online' IDSs (Khraisat et al., [55]).

While the number of proposed solutions and techniques for IDSs continues to rise, another related matter is of equal importance, to allow these types of systems to flourish and perform their functions as adequately as possible, which is **how** these systems communicate, and how secure those communications are. The communication within an Intrusion Detection System (IDS) must be secure and efficient due to the sensitive nature of the information being transmitted. If a privacy breach occurs within the system being monitored, the speed at which the system recovers is directly proportional to the speed at which the IDS can communicate the incident to other participants in the system.

One potential scenario for deploying and using IDS' is in a smart city, where it could prove to be incredibly useful. This thesis will use this specific scenario as its starting point, where there exist several different types of devices, with different purposes, and where there is no ubiquitous and permanent Wi-Fi coverage, ultimately creating several "islands" of devices, each one similarly isolated from the others. Together, these features mean that the system will inherently be vulnerable to attacks or dubious exploitation from adversaries, further advocating in favour of using an IDS to mitigate these problems by quickly and securely communicating these security breaches to the rest of the system.

These devices, however different from each other, will all have decent computing and storage capabilities and are all assumed to be of the same range or type as a Raspberry PI [32], which are a series of single-board computers (SBCs), made by the Raspberry Pi Foundation, that run Linux, but also provide a set of GPIO (general purpose input/output) pins, allowing the control of electronic components for physical computing and exploring the Internet of Things (IoT). Examples of using Raspberry Pi in IoT, including IDS work (the second citation ahead), can be found at [71, 79]. The utilization of such technologies allows for the deployment of more intricate solutions on devices.

1.2 Motivation

To this end, the current methods rely on manually defined security policies and signatures that fail to fulfil the real-time, evolving, and imbalanced scenario. Consequently, a secure decentralized approach emerges as a promising solution to the problem at hand, capable of quickly communicating detected intrusions to the rest of the system.

Even so, securing said communications reliably is no small feat, due to the significant constraints that any IoT-focused system is subject to. These types of systems must deal with device heterogeneity, poor connectivity between sensors and micro-controllers, limited power levels on such devices, no guarantee of network availability, and sometimes even the mobility of end devices. Furthermore, problems common to distributed systems, such as network partitions, or tolerance to failures, are accentuated in the context of IoT.

All of these factors contribute negatively towards secure and reliable communication between an IoT system's constituents, which ultimately makes it a challenge to implement such a messaging system in this context.

1.3 Goals

The goals determined for this thesis are the following:

- Review existing protocols for group key agreement in the scope of the security properties they provide;
- Review the most applicable protocols for a secure decentralized IoT system, from the perspective of their implementation, i.e., how they achieve said group key agreement;
- Compare these protocols against each other to determine which will be the most appropriate for the context of this thesis;
- Implement a messaging system based on the chosen protocol, with additional support for offline communication;
- Deploy a functioning prototype in a lab environment;

Once all the proposed goals are met, the main objective for this thesis will be accomplished - to create a secure and efficient solution that can communicate detected intrusions in an IoT system, specifically, a smart city's IoT ecosystem, whilst mitigating some of the issues that commonly plague these types of systems.

2

Background - Concepts

The purpose of this chapter is to clarify the fundamental concepts covered in this dissertation. First, some general concepts and categorization of group communication protocols will be presented, followed by fundamental security concepts, which will be mentioned and ever-present throughout most of this document.

Furthermore, the solution's desired qualities will be outlined, which will later impact the choice of the protocol in subsequent chapters.

2.1 Group Communication Protocols Fundamental Concepts

Group communication protocols have been thoroughly researched for some time now, both in regard to their efficiency, as well as their security and applicability [14]. They can, however, be somewhat complex and difficult to understand, and as such, there are some relevant concept introductions to be made to facilitate their understanding. First, any group communication protocol is centred around establishing and using a group key, used as a basis to provide common security services (data secrecy, authentication and integrity). The establishment of such a group key can be broadly classified into two distinct categories:

- **Distributory** group key establishment, wherein the group key is created by a single entity that is both a group controller and key server (GCKS - Group Controller Key Server);
- **Contributory** group key establishment, wherein the group key is established by equal contribution of all its participants.

The former definition can be subsequently divided into centralized and decentralized approaches. In a centralized approach, the key server is solely responsible for the creation and distribution of the group key, it acts as the "centre" of the establishment. In decentralized approaches, which will be the main focus of this thesis, a protocol-defined hierarchy will instead share the burden of distributing the group key, in order to avoid bottlenecks and single points of failure. Furthermore, decentralized approaches can also

be: **membership based**, where the group key is modified whenever a membership change occurs, and **time based**, where the group key is systematically altered after each slot of time. The concepts above and their relations can be visualized in Figure 2.1, as specified by Rakesh Gangwar in [34].

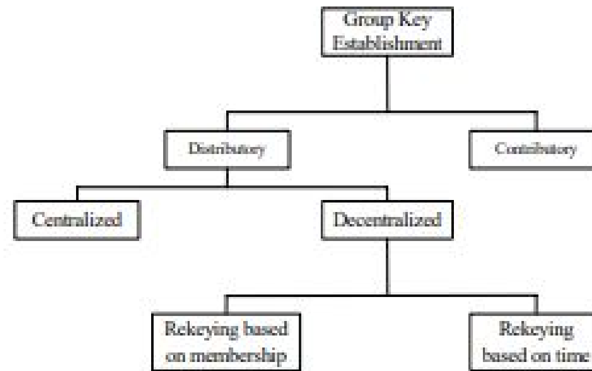


Figure 2.1: Group Key Establishment Architecture

2.2 Fundamental Security Concepts

To fully understand this thesis' goal, some relevant security concepts require a brief introduction and explanation.

Confidentiality: As per the National Institute of Standards and Technology (NIST) in [81], confidentiality in IT is the preservation of authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. In the specific context of group communication, confidentiality means that an application message sent by a group member can only be decrypted by users who are also members of that same group at the same time the message is sent, according to the sender's view of the group.

Integrity: As stated in [81], integrity refers to safeguarding information from unauthorized modification or destruction and ensuring the authenticity and non-repudiation of information. In group communication terms, integrity means that messages cannot be undetectably modified by anyone but the member who sent them.

Authentication: The sender of a message cannot be forged, and only authenticated members of the group can send messages to the group.

Non-repudiation: As per the special publication by NIST in [82], non-repudiation is the assurance that the sender of information is provided with proof of delivery and the

recipient is provided with proof of the sender's identity so that neither can later deny having processed the information.

Forward Secrecy (FS): By giving certain private keys a short "cryptoperiod"[1] and erasing them after they expire, it is possible to overcome the risk of recovery of derived keys due to the compromise of parties' cryptographic states. This prevents a passive opponent who merely recorded past communications encrypted with the shared secret keys from decrypting them some time in the future by compromising the parties' cryptographic state. This is the standard definition for forward secrecy, as per the IEEE Standard Specifications for Public-Key Cryptography [44]. In the context of group communications, this definition can be broadly adapted to: when a group member leaves a group, it must not be able to read any future messages from that group after its departure.

Backward Secrecy (BS): Ensuring that, whenever a new member joins a group, it cannot obtain access to any previous communications (including their cryptographic details), i.e., a new user cannot know anything about the system, before their arrival.

Post-Compromise Security (PCS): As stated in [16], a protocol between Alice and Bob provides Post-Compromise Security (PCS) if Alice has a security guarantee about communication with Bob, even if Bob's secrets have already been compromised. In other words, if an adversary compromises a group member, learning a snapshot of its current private state (including all secret keys), but the group member retains the ability to send messages, then the adversary can only decrypt messages until that group member sends a "PCS *update*" message that "heals" the compromise. More precisely, in the context of group communication, an adversary cannot decrypt messages sent by any group member that has the new key, i.e., that has processed the "PCS update". Additionally, if the adversary has gained persistent access to the user's device, they lose decryption ability as soon as their access is revoked (through a software update) and the PCS update is performed.

PCS With Concurrent Updates: Essentially the same as regular Post-Compromise Security, with the added constraint of the context where multiple users concurrently modify the group state.

Active Attack: An active attack is a network exploit in which a malicious adversary attempts to change data on the target or data en route to the target. Active attacks vary in their type (masquerading, denial of service, replay and message modification attacks), but the common link between them is the wrongful manipulation of data in the system.

Eventual Consistency: Not exactly a security property, but relevant to this context: all group members receive the same set of application messages (possibly in different orders),

[1] the period during which operations may be performed with the specified key.

and all group members converge to the same view of the group state, whenever there are no "write" operations (operations that produce modifications in the group's state) for a sufficiently long period.

2.3 Summary

The primary objective of this thesis is to develop a solution capable of communicating detected intrusions in an IoT system as securely and as efficiently as possible. With that objective in mind, the developed prototype should prioritize the security, availability and efficiency of its communications above everything else, guaranteeing as many of the security properties mentioned in Section 2.2 as possible, while, hopefully, maintaining efficiency and availability.

3

Literature Review

To better understand the context of the thesis and broaden the scope of the analysis, it was necessary to review the existing literature on the topic.

First, a general overview of the related work regarding their security is provided. This includes the security properties they can guarantee, their use cases, and how they are categorized. Next, the protocols showing the most potential will undergo further research to evaluate their specifications and suitability for the thesis' proposed use case. Finally, a summary of the related work will be given, concluding which protocol is the most adequate for this thesis.

3.1 Group Communication Protocols - A Security Overview

As mentioned in Section 2.1, there are several types of group communication protocols. The purpose of this analysis is to obtain a better grasp of the existing protocols and their specifications, adapt and connect these protocols to the context of this thesis, and hopefully, find an adequate solution for the problem at hand.

In Table 3.1, an overview of the security properties provided by each of the reviewed group communication protocols is displayed. There are many existing secure messaging protocols [85]. Schemes for two-party communication, providing forward secrecy and perfect forward secrecy to varying degrees, have been extensively studied in recent years. This includes the Signal protocol [62], its analysis [17], as well as several new protocols and their analyses, [9, 20, 49, 52, 74] as well as a modular analysis and generalization of Signal [4]. Among group messaging protocols for more than two parties, relatively few are both asynchronous and support dynamic groups, while guaranteeing the best possible security for the system, which, for the purpose of this analysis, will be considered as critical requirements for practical group messaging on mobile devices. This section will focus on the security properties provided by each of the discussed protocols, as they seemed the most promising when the existing literature was being researched.

In this table, a "Yes" means that the corresponding protocol can provide that specific security property, a "No" signifies the opposite case, where the protocol cannot guarantee

that security property, and a "-" means that the protocol's specification does not explicitly state that it can provide that property.

Table 3.1: Summary of reviewed protocols' security guarantees

Protocol	Message Secrecy & Integrity	Central server not needed	FS	PCS	Eventual consistency	Active Attack	PCS with concurrent updates
Signal Groups	Yes	Yes	-	Poor [a]	-	-	Optimal
Sender Keys(Whatsapp)	Yes	Yes	Yes	No	Yes	Yes	Optimal [b]
Matrix	Yes	Yes	No	Yes	Yes	Yes (after recent patches)	Optimal
MLS(TreeKEM)	Yes	No	Poor	No	Yes	Yes	Only one message sequence heals
Re-randomized TreeKEM	Yes	No	Yes	Yes	-	Yes	
Causal TreeKEM	Yes	Yes [c]	Very Poor	No	-	Yes	Any sequence heals
Concurrent TreeKEM	Yes	Yes[c]	No	Yes		Yes	After two rounds
DBGK	Yes	Yes	Yes	-	-	Yes	-
Baal	Yes	Yes	Yes	-	-	-	-
Proxy Cryptography scheme	Yes	Yes	Yes	-	-	-	-
DLGKM-AC	Yes	Yes	Yes	-	-	Yes	-
DCGKA	Yes	Yes	Yes	Yes	Yes	Yes	All but last can be concurrent

[a] Optimal PCS in the face of concurrent updates is possible by using a 2-party protocol with optimal PCS+FS in place of pairwise Signal
[b] Optimal PCS in the face of concurrent updates is possible at the given costs, but not used in practice
[c] Does not specify how to determine group membership in the face of concurrent additions and removals

In the following subsections (Subsections 3.1.1-3.1.12), a more in-depth view of how such protocols achieve such security properties is presented and explained.

3.1.1 Signal Groups

Signal Groups is the extension of the famous Signal protocol [4]. It can be adapted to a decentralized setting, and it utilizes a simple protocol: the sender of each application message sends the message individually to each other group member using the two-party Signal protocol [76]. While staying off a detailed description of this protocol's functionality, it essentially secures end-to-end encryption (E2EE, for short) by first agreeing on a shared secret key via the X3DH key agreement protocol [63], and then using the Double Ratchet Algorithm [62] to send and receive encrypted messages[1]. Because this protocol's functionality only entails two separate entities, it tends to show some scalability problems, quickly becoming inefficient in large groups, since every application message requires *n-1* two-party messages in a group of size *n*. Nevertheless, it is still able to provide most of the security properties mentioned in Section 2.2, with special attention to PCS, which can only be achieved if a group member performs a PCS update only after receiving a message from every other member, which would never happen if a member is always offline.

3.1.2 Sender Keys (Whatsapp)

As its name indicates, Sender Keys is the closed-source protocol used by Whatsapp [88]. Here, each group member generates a symmetric key for messages they send, and then sends this key to every other group member using the two-party Signal protocol. Forward secrecy is easily achieved, since, for each message sent by a member to the group, a new key is derived pseudorandomly from the previous key. Whenever a user is removed, each

[1] A ratchet, in the literal sense of the word, is simply a device that only moves forward, one step at a time. In cryptography, a ratchet method allows for future states to be calculated, but only if an original seed value is known. It is not possible to calculate previous states from the current state. Typically this is done with a one-way function such as a hash with a seed value and a "salt".

remaining group member generates a new key and sends it to the other members using the same two-party channels. Beyond forward secrecy, this protocol guarantees message secrecy, integrity, eventual consistency and protection against active attackers. It could also provide PCS by updating keys periodically, but Whatsapp chooses not to do this. Like Signal Groups, Sender Keys can also be adapted to a decentralized context, with the caveat that PCS updates become expensive. If one user is compromised, all of the sender keys become known, which means that, to recover from the compromise, each of the n group members needs to generate a new key and send it to every other of the n-1 remaining group members, resulting in a $O(n^2)$ complexity, which is undesirable.

3.1.3 Matrix

Matrix is an open-source project [26, 25] that publishes its namesakes' open standard for secure, decentralized, real-time communication. The end goal of Matrix is to be a ubiquitous messaging layer for synchronising arbitrary data between sets of people, devices and services - be that for instant messages, VoIP call setups, or any other objects that need to be reliably and persistently pushed from A to B in an inter-operable and federated manner. To this end, there are Matrix APIs for several uses: Instant Messaging, Voice Over IP(VoIP) signalling and Internet-of-Things(IoT) communication.

Its end-to-end encryption protocol is a variant of Sender Keys. It is decentralized by default, provides PCS and explicitly handles concurrent updates and group membership changes. It does not, however, provide backward secrecy, although this can be mitigated by changing some encryption parameters for HTTP events since its implementation is open-source. Similarly, it is only 'partially' forward-secret, since past messages can be decrypted multiple times. There are some mitigation mechanisms for this though, such as offering the user the option to discard historical conversations. There has been a formal security analysis of this protocol [64], which yielded several concerning results, that have since been patched in more recent versions of Matrix. Despite this, however, Matrix still provides all of the desired security properties in this context (mentioned in Section 2.2), except for forward secrecy.

3.1.4 MLS(TreeKEM)

MLS, or the Messaging Layer Security protocol [70], is a standard still in development, that uses the TreeKEM [53, 6] key agreement protocol at its core. Membership operations, like adding or removing a member of a group, and PCS key updates, require the broadcast of a message of size $O(log(n))$, and can be proposed concurrently. TreeKEM achieves this through means of a binary tree (hence the protocol's name, "Tree"KEM), with one leaf per group member, and each member knowing the secret keys on their leaf node's path to the root. However, these types of operations can only take effect after being committed, and every process must process these commits in the same order, which somewhat limits implementation options. Because MLS is inherently centralized, since it typically depends

on a semi-trusted server to determine the sequence of commits, it also suffers in the context of large groups. In regards to its security guarantees, MLS, by default, can provide message secrecy and integrity, eventual consistency, post-compromise security (PCS) and protection against active attackers, while struggling to guarantee forward secrecy.

3.1.5 *Re-randomized* TreeKEM

Alwen et al. [5] introduced a variant of TreeKEM to achieve better forward secrecy, in addition to the rest of the security properties that TreeKEM already provided. This variant is even harder to decentralize than the original TreeKEM, because group members update each other's secret keys so that each key is only used once (to achieve better forward secrecy). The downside to this approach stems from the fact that if multiple concurrent messages are encrypted under the same public key, the protocol, and its security guarantees, will eventually break.

3.1.6 Causal TreeKEM

Causal TreeKEM [87] works in the opposite direction to *Re-randomized* TreeKEM, where it only requires causally-ordered message delivery, and, as a result, can only provide very weak forward secrecy. This protocol describes how to handle dynamic groups (the adding and removal of members "on the fly") in a decentralized setting, by eliminating the need for a linear order of state changes. This alteration in the original protocol's functionality improves the overall consistency but struggles to improve any of the other security properties desired.

3.1.7 Concurrent TreeKEM

Another protocol is a "concurrency-aware" variant of the regular TreeKEM protocol, proposed by Bienstock, Dodis, and Rösler in [10]. In the article where this protocol is proposed, a formal study of the trade-offs between PCS, concurrency and communication overheads in the context of group ratcheting is performed. Through this study, the authors can successfully outline a protocol with PCS updates whose cost scales with the number of previous concurrent messages, even matching MLS' $O(log(n))$ complexity, when all total messages are ordered. However, the authors also assume that PCS updates occur in fixed rounds, with all messages from one round being received before the start of the next round. This is not always the case in the context of IoT systems and distributed systems with limited availability, which would pose a limitation for the goal of this thesis. Furthermore, the authors also do not consider forward secrecy or dynamic groups in their proposition, which are key aspects of systems in the context of IoT.

3.1.8 DBGK

The DBGK protocol, as described by Admeziem, Tandjaoui and Romdhani in [3], is a group communication protocol designed specifically for usage in an IoT application, since it takes into account the resource scarcity and the mobility of the devices that are usually prevalent in an IoT environment. The authors of this protocol put extensive effort into reducing the issues that usually affect IoT applications: by designing their protocol to be decentralized, they mitigate the single point of failure issue; to reduce the impact of the *1-affects-n* phenomenon, each subgroup (of devices) of the IoT network is secured with a different group key. Moreover, a time-driven approach is used to replace the group keys after each time slot or interval, and members-only request the required keys for a particular interval, to reduce memory requirements for key storage.

Regarding the security guarantees it provides, the DBGK protocol ensures backward secrecy, forward secrecy, data confidentiality, data integrity and protection against active attackers (through periodic re-keying).

3.1.9 Baal

Despite its odd name, Baal [46] is a decentralized group key management protocol specifically designed to solve some of the issues that frequently arise with protocols whose purpose is secure multicast communication. Baal's general specification is as follows: it uses a group controller (GC), and local controllers (LCs), to configure and manage the system's dynamic groups; A single group key is used at any time to encrypt all of the group traffic; Whenever a member is evicted, its local controller generates a new group key and multicasts it to the rest of the LCs, and group members upstream, but not downstream; It unicasts the key to its local members with their respective public keys, while omitting the one being evicted.

The authors' reasoning behind utilising only a single group key is performance-related, to reduce the cost of re-keying operations, whenever there are changes in the groups' memberships.

In regards to the security properties it can provide, Baal guarantees data confidentiality and integrity, protection against eavesdroppers and both forward and backward secrecy. There is no mention of post-compromise security in the security model presented in the paper on which Baal is specified.

3.1.10 Decentralized Group Key Management for Dynamic Networks Using Proxy Cryptography

This paper [43] describes an approach to the problem at hand that distinguishes itself from others that were reviewed, as it utilizes a different cryptography concept known as proxy

cryptography[2]. Specifically, the protocol proposed in this paper makes use of the ElGamal proxy cryptography [48] scheme, to deliver a re-keying message to valid members upon a member change. The main reasoning behind the suggested protocol is the following: the secret key of each proxy and the group key of session *i* are used to encrypt the group key update message for the next session *i + 1*. Thus, only authorized members who know both of the secret keys can decrypt the re-keying message and acquire the updated group key. As the key update message is converted along the multicast path using proxy cryptography in a distributed environment, the need for a centralized key distribution centre (KDC) to manage the creation and updates of proxy keys can be eliminated in the proposed framework. This approach enables each proxy of sub-networks to reconfigure the multicast environment dynamically, while still confining the membership or multicast network topology changes to the local area. Thus, this protocol can guarantee both backward and forward secrecy, data confidentiality and integrity, and protection against active attackers.

3.1.11 DLGKM-AC

DLGKM-AC stands for "Decentralized Lightweight Group Key Management for Dynamic Access Control", and, as its name suggests, it is especially relevant for the context of this thesis mainly for two reasons: the paper in which it is presented is specifically directed to the context of the IoT, and it is a decentralized group key management protocol that takes into account the dynamic aspect of IoT devices. Dammak et al. describe a very detailed protocol in their article [19], defining the overall system, the attacker model and the system requirements as well, whilst always mentioning and adapting the protocol specification to the context of a mock IoT application use case. The protocol is essentially divided into three distinct layers: two layers define groups of devices (DGs) and users (UGs), respectively, and between them, there is a decentralized controller, the KDC (Key Distribution Center), which is responsible for managing keys between and within groups. The protocol implementation itself is rather lengthy and complex, however, it does ultimately achieve most of the desired security properties[3]: both backward and forward secrecy, message integrity and secrecy, protection against active attacks and it might also be able to provide PCS, with some small alterations to the algorithms it uses.

3.1.12 DCGKA

DCGKA [86] stands for "Decentralized Continuous Group Key Agreement" and it is perhaps one of the most complete protocols in analysis here, in regards to the security properties it provides. This protocol generates a sequence of *update secrets* for every group

[2] The basic idea of proxy encryption is that a proxy, given a proxy key, could transform ciphertext corresponding to one person, into the ciphertext for another person without revealing any information about the secret decryption keys or the plaintext.

[3] The security analysis of the DLGKM-AC protocol does not mention eventual consistency.

member, which are then used as input for a ratchet to encrypt and/or decrypt messages sent by that particular member. Only the members of the same group receive these secrets, and fresh secrets are generated every time a new user joins the group, an existing user leaves the group, or when a PCS update is requested. Since each message is encrypted with a different key, forward secrecy is inherently achieved by this protocol. Furthermore, since secrets are regularly updated, there is some protection against active attackers, by significantly reducing their window of attack if a device is indeed compromised.

Moreover, the protocol ensures that every user observes the same sequence of update secrets for each group member, regardless of the order in which the concurrent messages are received, guaranteeing eventual consistency.

Overall, the DCGKA protocol provides all of the security properties shown in Table 3.1, while utilizing a decentralized architecture, making it a prime candidate for further analysis from a distributed computing point of view, concerning the theme of this thesis.

3.1.13 Summary

To conclude the analysis of these protocols from a security point of view, the protocols that were deemed the most applicable to the context of this thesis were: Matrix (Subsection 3.1.3), the DBGK protocol (Subsection 3.1.8), the DLGKM-AC protocol (Subsection 3.1.11) and the DCGKA protocol (Subsection 3.1.12. All four of these protocols are specified in a decentralized architecture while taking into account the constraints of IoT systems and dynamic groups, and while providing most, if not all, of the desired security properties, which is why they seemed the most interesting to further analyze and investigate, from the perspective of how they disseminate group messages and secret keys, which will be presented in Section 3.2.

3.2 Group Communication Protocols - Distributed Computing and Message Dissemination

In this section, the protocols mentioned in Subsection 3.1.13 will be analyzed in regards to their implementation, and how they disseminate messages across the system's participants, i.e., from a distributed computing perspective. At the end of this section, a conclusion will be made, regarding the protocol most suited to the needs of the system on which the prototype will be built.

3.2.1 Matrix

Matrix is a set of open APIs that communicate and operate with each other for multiple purposes (stated in Subsection 3.1.3), and its main goal is to provide an open, decentralized pub-sub layer for the internet for securely persisting and publishing/subscribing JSON objects.

3.2.1.1 Architecture

As specified in [29], Matrix defines APIs for synchronising extensible JSON objects known as "events" between compatible clients, servers and services. Clients are typically messaging/VoIP applications or IoT devices/hubs and communicate by synchronising communication history with their "homeserver" using the "Client-Server API". Each homeserver stores the communication history and account information for all of its clients, and shares data with the wider Matrix ecosystem by synchronising communication history with other homeservers and their clients.

To employ communication between fellow clients, Matrix makes use of the concept of a "virtual room" where clients can emit events. Room data is replicated across all of the homeservers whose users are participating in a given room. As such, no single homeserver has control or ownership over a given room. Homeservers model communication history as a partially ordered graph of events known as the room's "event graph", which is synchronised with eventual consistency between the participating servers using the "Server-Server API". This process of synchronising shared conversation history between homeservers run by different parties is called "Federation". Matrix optimises for the Availability and Partitioned properties of CAP theorem [36, 12] at the expense of Consistency.

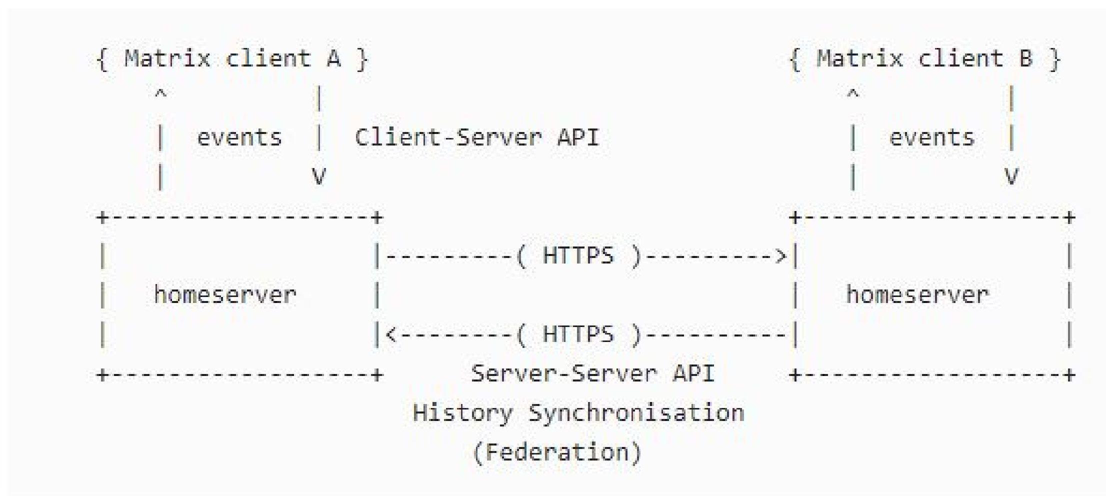


Figure 3.1: Scheme detailing how data flows between Matrix clients

In Figure 3.1, the general data flow between Matrix clients is exemplified: for a client A to send a message to client B, client A must perform an HTTP PUT of the required JSON event on its homeserver (HS), using the Client-Server API. Client A's HS appends this event to the copy of the room's event graph, signing the message in the context of the graph for integrity purposes. That same HS will then replicate the message to client B's HS, by performing an HTTP PUT using the server-server API. Client B's HS then

authenticates the request, validates the event's signature, authorises the event's contents and then adds it to its copy of the room's event graph. Client B then receives the message from its homeserver via a long-lived GET request.

3.2.1.2 Devices

In Matrix, there is a particular meaning for the term "device", since a single user might have several different devices: a laptop, a desktop client, an Android device, an iPhone, etc. Each device broadly relates to a real device in the physical world, but a user might have several browsers open on a physical device, or several Matrix client applications on a mobile device, each of which would be its own device in Matrix.

This definition of a device is used primarily to manage the keys used for end-to-end encryption (each device gets its own copy of the decryption keys), but it also helps users manage their access - for instance, by revoking access to particular devices.

3.2.1.3 Events

All data exchanged over Matrix is expressed in the form of an "event". Typically, each client action (e.g. sending a message) correlates with exactly one event. Each event has a *type*, which is used to differentiate between different kinds of data. Values for this field must be uniquely and globally namespaced, following Java's package naming conventions [67], e.g. *com.example.myapp.event*. There is also a special top-level namespace, *m*, which is reserved for events pertaining to Matrix's specification - for instance, *m.room.message* is the event type for instant messages. Additionally, events are usually sent in the context of a "room".

3.2.1.4 Encryption Algorithm

Encryption and Authentication in Matrix are based on public-key cryptography, with the support of the Olm [30] and Megolm cryptographic ratchets, which are themselves based on the Double Ratchet cryptographic ratchet [62] described and popularized by Signal. The Matrix protocol provides a basic mechanism for the exchange of public keys, though an out-of-band channel is required to exchange fingerprints between users to build a web of trust. Alongside access tokens, Matrix makes use of several keys for its encryption methods:

Ed25519 fingerprint key pair Ed25519 is a public-key cryptography system for signing messages. Matrix makes use of this system by assigning an Ed25519 key pair to each device, to identify that device.

Curve25519 identity key pair Curve25519 is a public-key cryptographic system that can be used to generate a shared secret. In Matrix, this is used for each device to have its own long-lived Curve25519 identity key which will be used to establish Olm sessions with that

device. Here, the private key should never leave the device, but the public key is signed together with the Ed25519 fingerprint key and published to the network.

Ideally, this identity key should be rotated periodically, but that has not been implemented yet.

Curve25519 one-time keys Besides the identity keys, each device also contains several Curve25519 one-time key pairs, which can only be used once. Their purpose is the same as the identity keys but for Olm sessions between different devices.

Megolm encryption keys The Megolm key is used to encrypt group messages (in reality, it is used to derive an AES-256 key and an HMAC-SHA-256 key). It is initialized with random data, and each time a message is sent, a hash calculation is performed on the Megolm key to derive the key for the next message. This makes it possible to share the current state of a Megolm key with a user, allowing them to decrypt future messages, but not past messages.

Additionally, when a sender creates a Megolm session, it also creates another Ed25519 signing key pair. This pair is used to sign messages in that particular session, to authenticate the sender, and, once again, the private key is kept only on the device, while the public key is shared alongside the encryption key with the rest of the devices in the room.

The first three types of keys are distributed as portrayed in Fig.3.2, where "HS" means homeserver and Alice and Bob are two Matrix clients.

Functionality Using all of the types of keys mentioned earlier in this section, and following the names of the schema present in Fig.3.2, Matrix's encryption algorithm between Alice and Bob functions as follows:

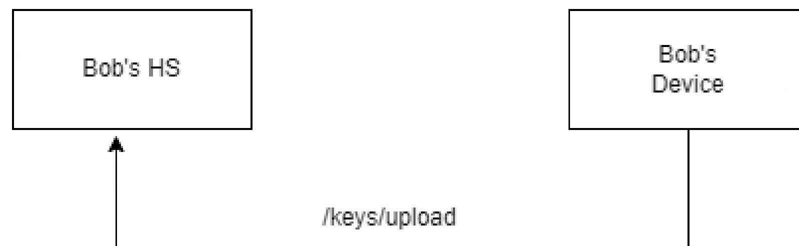
1. After generating its Curve25519 one-time keys, the Curve25519 identity key and the Ed25519 fingerprint keys, Bob uploads his keys' public parts to the server;

2. Then, Alice's device, the one wishing to communicate with Bob, requests Bob's public identity keys and supported algorithms from the server. Assuming the supported algorithms of both clients match, Alice then claims one of Bob's one-time keys;

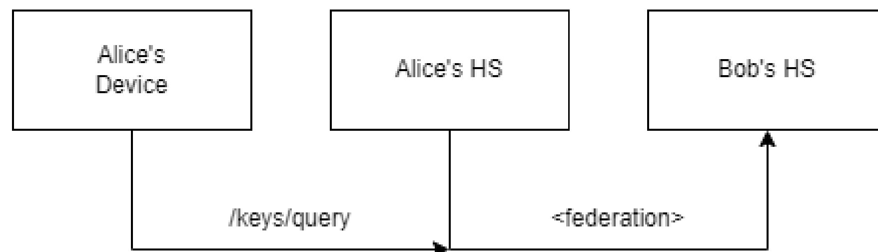
3. Alice should then check the signatures on the signed key objects in the response from the server, and provided that the key object passes verification, the one-time key and Bob's identity key should then be used to create a new **Olm** session[4] between Alice and Bob;

[4] Olm sessions are not used for encrypting room events, as they require a separate copy of the ciphertext for each device, and because the receiving device can only decrypt received messages once. However, they are used as a secure channel through which the clients can share their Megolm keys.

1. Bob publishes the public keys and supported algorithms for his device.
This may include long-term identity keys, and/or one-time keys.



2. Alice requests Bob's public identity keys and supported algorithms.



3. Alice selects an algorithm and claims any one-time keys needed.

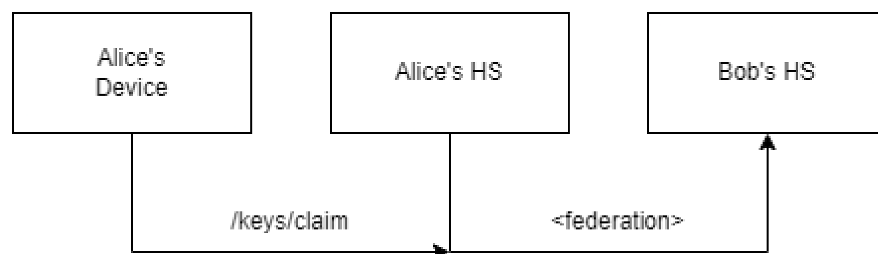
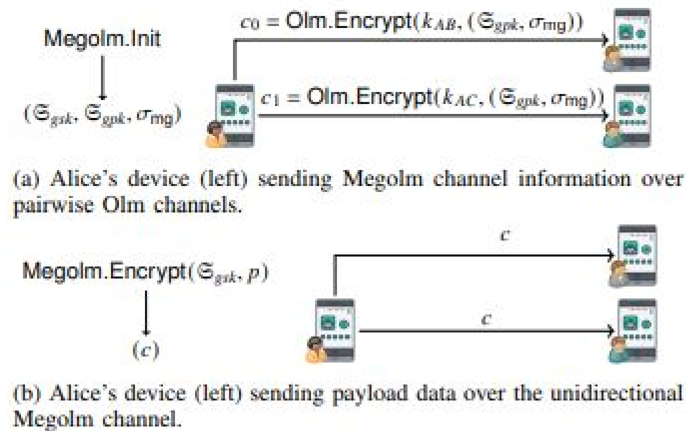


Figure 3.2: Matrix Key Distribution

4. Once an **Olm** session has been established between the two clients, they can now share their Megolm encryption keys inside a room where the encryption setting has been enabled, which will be the encompassing context for the next few listed items;

5. Inside a room where encryption has been enabled, and provided that Olm sessions have been established as described in the first few items of this list between all



(a) Alice's device (left) sending Megolm channel information over pairwise Olm channels.

(b) Alice's device (left) sending payload data over the unidirectional Megolm channel.

Figure 3.3: Alice establishes a Megolm channel (a) and sends a ciphertext (b)

the participants of the room, whenever a message is first sent in that room, a new outbound **Megolm** session should be created;

6. To this end, Alice can generate its **Megolm** session key from her Megolm cryptographic ratchet, and share it with Bob through their **Olm** channel with a *m.room_key* event, specific for this purpose:

7. When Bob receives the **Megolm** session sent by Alice, he MUST ensure that the session was received via an **Olm** channel, to ensure the authenticity of the messages. Additionally, he must also ensure that the session data comes from a trusted source, such as via a *m.forwarded_room_key* event from a verified device belonging to the same user, or from a *m.room_key* event, and that the session key that was received has a lower message index than the one that is currently known to Bob;

When all of these steps are complete, clients inside a room can now exchange encrypted messages until their **Megolm** session expires, at which point the process will restart from step 4 of the list, generating a new Megolm session with a new session key derived from the same cryptographic ratchet. An example of the process described above can be found in Fig.3.3. For a more in-depth and precise definition of the Megolm ratcheting protocol, see Annex I.

3.2.1.5 Event Graphs

Events exchanged in the context of a room are stored in a directed acyclic graph (DAG), called an "event graph". The partial ordering of such a graph maintains the chronological order of events within the room. Each event in a DAG possesses a list of zero or more "parent" events, which refer to any preceding events that do not have chronological successors from the perspective of the homeserver that created the event.

Typically, an event has only one parent - the most recent message in the room at the point the event was sent. However, this means that homeservers will inevitably race with each other when sending messages, resulting in a single event having multiple successors. The next event added to the graph will have multiple parents. Every event graph has a single root event with no parent.

To resolve these cases and ease the ordering and chronological comparison of events within the graph, each homeserver keeps a *depth* metadata field for each event. An event's *depth* is a positive integer that is strictly greater than the depths of any of its parents. The graph's root event has a depth of 1. Therefore, if an event is before another, then that event must have a strictly smaller *depth* value.

3.2.1.6 Rooms

A room in Matrix is a conceptual location where users can send and receive events. Events are sent to a room, and every participant with sufficient access to that room will be able to receive the event, i.e. each room corresponds to exactly one group (of participants). Each room is uniquely identified internally with a "RoomID" of the form: *!opaque_id: domain*. Whilst the room ID does contain a "domain", it is there simply for global namespacing purposes, and that room does not reside on the specified domain.

In Figure 3.4, a conceptual diagram is shown, representing an *m.room.message* event being sent to room *!qporfwt:matrix.org*.

As mentioned in the **Architecture** paragraph, the "Federation" maintains shared data structures per room between multiple servers, and data is split into two: *message events* and *state events*.

- Message events describe, transient, one-time activity in a room, such as instant messages, VoIP call setups, file transfers, etc. They generally describe communication activity.

- State events describe changes to a room's state, which is a set of persistent information, such as the room's name, topic, membership, etc. In Matrix, a room's state is modelled as a table of key-value pairs per room, with each key as a tuple constituted by a *state key* and the *event type*. Each state event updates the value of a given key.

How is the state of a given room calculated? The state of a room at any given point is calculated by considering all events before and including a given event in the graph. When events describe the same state, a merge conflict algorithm is executed. This merge resolution algorithm is transitive and does not depend on the server state, as it must consistently select the same event, irrespective of the server or the order the events were received. An event's originating server, i.e. the one who sends the event, signs the event

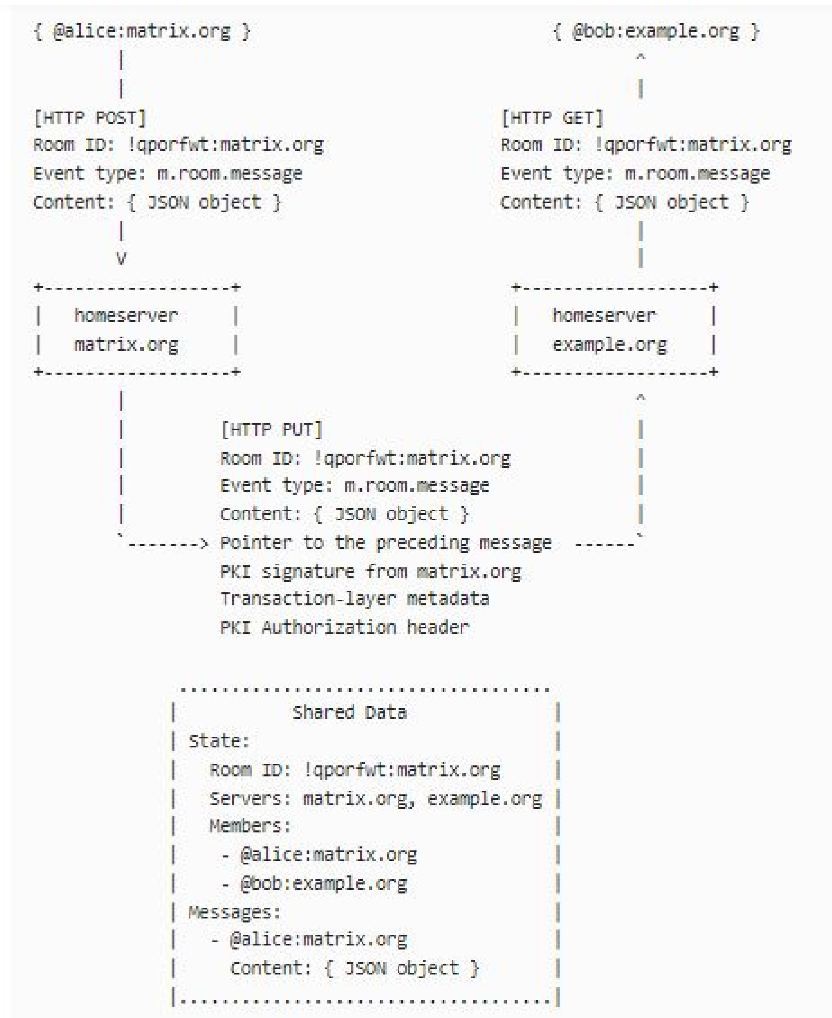


Figure 3.4: Diagram with a Matrix event example

(this signature includes the parent relations, type, depth and payload hash) and pushes it "over federation" to the room's participating servers, currently using full mesh topology[5].

Lastly, servers can also request the history of events over federation from the other servers participating in the room.

3.2.1.7 Protocol shortcomings

The most obvious shortcoming of Matrix is the fact that it does not provide perfect forward secrecy and backward secrecy by default. Although this can be easily mitigated, by adjusting internal encryption parameters, it is still a relevant issue to keep in mind.

[5]A mesh network is a network in which devices - or nodes - are linked together, branching off other devices or nodes. In a **full mesh network topology**, each node is connected directly to all the other nodes.

3.2.2 DBGK

The main goal of the DBGK protocol, as mentioned in Subsection 3.1.8, is to address the security concerns of group communications in mobile IoT environments. This poses a great incentive for the work of this thesis, and, as such, the analysis of the ensuing protocol will be most relevant.

Provided Features The features outlined in DBGK's paper specification are the following:

- Mitigation of the *l-affects-n* issue. In fact, in addition to the adoption of a decentralized architecture, this protocol only involves the active members in a rekeying process, which allows the other members to remain inactive, and, therefore, save energy.
- The joining process for a new group member does not require it to store a high number of keys. Alternatively, based on the new group member's storage capabilities and future behaviour in the group, the new member will only ask for the keys it requires.
- Mobility of the participants is handled inherently, without the need for an assumption to be made regarding the authenticity of a node after the handover.
- The rekeying process is not based on a prediction of the handover or departure time, since the protocol assumes a highly dynamic IoT network with unpredictable leave and join events.

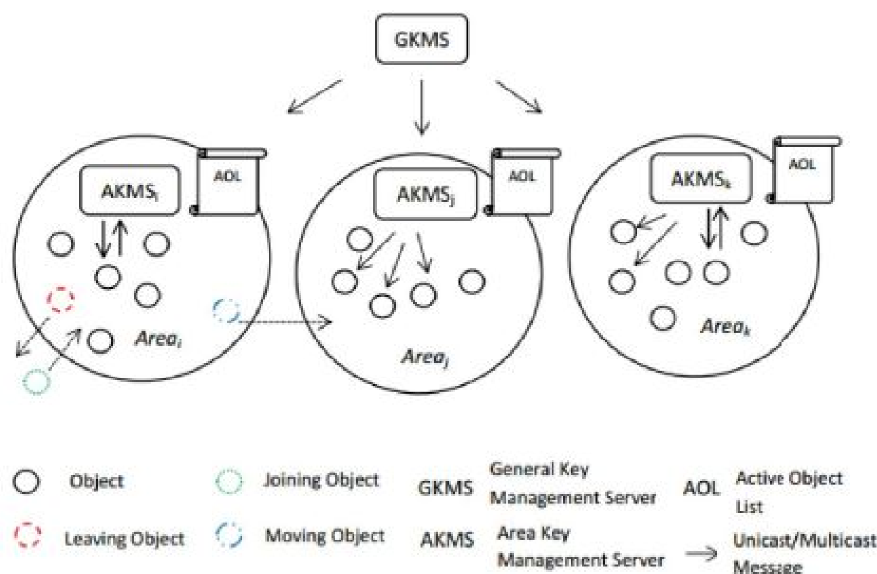


Figure 3.5: DBGK network model: a decentralized architecture based on an independent group key per area

Network Model As shown in Figure 3.5, the DBGK protocol relies on a slightly complex network model, divided into several areas, and involving several different types of entities. Each one of these areas covers several objects: these objects can be any entity of the average person's daily life, that can process, store and communicate data to other entities and the Internet. Each area is managed by one Area Key Management Server (*AKMS*) (See Figure 3.5), or "controller". The *AKMS* establishes a Traffic Encryption Key (TEK) for each object in its area. Each TEK is unique among all the TEKs for each different area. Using this key, the objects secure their communications inside the area to which they belong. In case of an event, the controller is responsible for updating the TEK. The event can be triggered by a new object joining the area, an existing object leaving the area or a moving object between different areas. Furthermore, each *AKMS* maintains a list of Active Objects, an AOL - Active Object List, which stores the delivered credentials to the objects for each time slot. Those such credentials are utilized in the rekeying process. Additionally, a *GKMS* (General Key Management Server) manages all the different *AKMS* and defines the security policy for the entire group. More specifically, it ensures that the appropriate access control policy is in effect for each area. Moreover, it can also act as a backup for a given *AKMS* in case it fails or is overloaded (due to a DoS attack, or hardware failure).

This protocol uses multiple modes of communication, i.e. unicast, multicast and anycast depending on the stage that is currently being executed.

With the earlier definition of an entity in mind, two categories of entities are distinguished, with various capabilities both in terms of computing power and energy resources:

- highly resource-constrained entities (i.e. the objects), with few MHz of computational power, several kilobytes of RAM, and several tens of kilobytes of ROM;
- entities with high energy, computing power, and storage capabilities (i.e. *GKMS* and *AKMS*);

Protocol Assumptions To better understand the protocol itself, a few assumptions are required:

- The objects mentioned in this section are IP-enabled and run 6LoWPAN [11, 56] adaptation layer. Thus, the authors of the protocol assume there is always a gateway, for instance, a 6LoWPAN border router, through which the 6LoWPAN headers are compressed and decompressed.
- All *AKMS* and the *GKMS* are considered trusted entities.
- Before the start of the protocol, the different entities have already successfully established the Key Encryption Keys (KEK) between them.
- A long-term key, SK, is shared inside each area. It is distributed to any new node that joins the area, and it is used in the generation of the Traffic Encryption Key (TEK).

- Time is split into several slots of an established length. A different ticket is assigned to each slot, and a ticket is a piece of data that is used in the generation of the different TEKs.

- an object can join a group asynchronously, but it must always wait for the beginning of the next slot before becoming an active object. Similarly, an object should only leave a group at the end of a time slot. Hence, the authors assume a synchronous batch rekeying protocol, which introduces a time delay in the overall execution of the protocol but provides a reduced number of rekeying operations.

$GKMS$	General Key Management Server
$AKMS_i$	Area Key Management Server of area i
AOL_i	Active Object List of area i
O_i	Object i
$TEK_{i,t}$	Traffic Encryption Key for area i and time slot t
$KEK_{O_i,AKMS_j}$	Key Encryption Key shared between O_i and $AKMS_j$
$KEK_{O_i,GKMS}$	Key Encryption Key shared between O_i and $GKMS$
$KEK_{AKMS_i,GKMS}$	Key Encryption Key shared between $AKMS_j$ and $GKMS$
SK	Long term Secret Key shared between the members of each area
$Slot_t$	Interval of time between time t and time $t + 1$
$T_{i,t}$	Ticket issued in area i for slot t
ID_{O_i}	The identity of Object i
$Data_{0_i}$	Information on O_i's storage and processing capabilities
N Slot	Number of requested tickets (that correspond to slots)
F	One way function

Table 3.2: DBGK protocol terminology

DBGK Messaging Protocol Here, the specification of DBGK will be presented exactly as in its original paper [3], concerning the different messages that are exchanged throughout the protocol's execution, and to what events trigger those message exchanges.

Join event An object O_i willing to join a group sends a JOIN_REQUEST message to the anycast address of the *AKMS'*. The nearest *AKMS* handles the request. The message has the following structure:

$$< ID_{O_i}, Data_{0_i} >$$

Assume an $AKMS_j$ handles this request. It then forwards the JOIN_REQUEST to the *GKMS*. Based on the object's specificities, its trusted level (i.e. if it has previously been a member of a group) and the current access policy, the *GKMS* decides on whether to deny or grant access to object O_i. The *GKMS* sends a JOIN_RESPONSE message in unicast to $AKMS_j$, containing its decision. If O_i was accepted, $AKMS_j$ sends a GRANT_NOTIFY message to it. Here, an assumption is made about the occurrence of an initialisation phase

where O_i receives $KEK_{O_i,AKMS_j}$, $KEK_{O_i,GKMS}$ and SK in a secure fashion. In fact, an offline "dealer" is in charge of O_i's initialisation. Upon the conclusion of this initialisation phase, O_i is considered a valid and authenticated member of area j.

Exchanging messages First, to secure its communications within the group, O_i will have to derive the group's corresponding $TEK_{j,t}$, which is computed as follows:

$$TEK_{j,t} = F(SK, T_{j,t}) \tag{3.1}$$

This F function could be implemented as a one-way hash function, such as SHA-1 [78]. To obtain the corresponding tickets necessary to derive $TEK_{j,t}$, O_i sends a TICKET_REQUEST. This message contains the following information: the identity of the object, its storage and processing capabilities, and the number of requested tickets. In fact, O_i can request several tickets according to its expected activity during the future slots. The TICKET REQUEST message is sent in unicast to the corresponding $AKMS_j$. This message has the following structure:

< ID_{O_i},$Data_{0_i}$, NSlot>

Upon receiving a ticket request, the $AKMS_j$ decides whether to deny or grant access to O_i, based on the object's trust level and the access control policy currently in effect. Moreover, it decides on the number of ticket to be granted. When the $AKMS_j$ accepts the request, it sends a TICKET_RESPONSE message to O_i, which will then generate the corresponding $TEK_{j,t}$ for the different tickets it requested, using equation 3.1.

Leave event For the DBGK protocol, the ejection of an object is handled based on whether that object is still in possession of valid tickets or not. If the leaving object O_i, present in area j, is not active, and thus not in possession of the current $T_{j,t}$ or any future $T_{j,t+k}$ (k > 0), no rekeying operation is required, as this means this node cannot compromise any future operations of the system. This way, forward secrecy is ensured as O_i will not be able to derive any $TEK_{j,t}$. In the case where O_i does have some remaining valid tickets, $AKMS_j$ will have to perform a rekeying operation to ensure forward secrecy in its area. As mentioned before in this section, each $AKMS_j$ maintains an Active Object List (i.e. AOL_j), and each entry in that list is of the form:

$T_{j,t}$ —> $List_t$

$List_t$ refers to the list of objects in possession of ticket $T_{j,t}$. Upon the departure of O_i, $AKMS_j$ checks its AOL_j, looking for the objects that have the same tickets as those delivered to O_i. Afterwards, $AKMS_j$ sends a multicast LEAVE_NOTIFY notification to every object to whom it concerns. This message has the following structure:

$$< T_{j,t}, T_{j,t+1}, ..., T_{j,t+k} >$$

The recipients of this notification will invalidate the tickets they received in the message, and ignore it if the tickets are no longer in use. However, if they are still using the tickets, then they must send a TICKET_REQUEST to the $AKMS_j$. This way, forward secrecy is always ensured, with minimum overhead, since only active objects are involved in the rekeying operation.

Node Mobility Here, an object O_i is considered "mobile" when it moves from an area J to another area K. The process of moving a node from one area to another starts with the object sending a MOVE_REQUEST message to $AKMS_j$. Assume that object O_i arrived at area K during time slot x. Even though O_i has secret key SK, it is not enough to derive $TEK_{k,t}$ (with $t < x$). Therefore, backward secrecy is inherently ensured. Forward secrecy for area J is handled by treating this event as a leave event. Indeed, if O_i is not in possession of a valid ticket, no rekeying operation is required.

***AKMS* Unavailability** Area Key Management Servers are responsible for maintaining the keying materials of their respective areas. In the case where the *AKMS* of a particular area is not available (e.g. hardware failure, DoS attack, etc.), the communications inside the group need to remain secure. This is where the main advantage of a decentralized setting is in full display since it will avoid the single point of failure issue.

Each time an object O_i sends a message to its *AKMS*, it waits for a certain period of time. If no response is received by the end of it, the object will assume its *AKMS* is unavailable and will proceed to send a SPARE_REQUEST notification to the *GKMS*. The structure of this message is as follows:

$$<ID_{O_i}, AKMS>$$

As $KEK_{O_i,GKMS}$ is shared between O_i and the *GKMS*, the latter will be capable of managing the affected area's requests until the restoration of the *AKMS*, acting as a back-up *AKMS*, hence mitigating the single point of failure issue.

GKMS Unavailability Although the protocol's specification does not consider this possibility, it is easy to infer how it would tremendously affect the system's functioning. If the General Key Management Server (*GKMS*) is unavailable, it can result in crucial aspects of the protocol's functionality simply not functioning. This is because the *GKMS* not only decides if an object can join a group but also acts as a backup *AKMS* in case of failure. For all purposes, this was considered to be a design flaw of the protocol.

Periodic Updates Long-term attacks could be carried out in an attempt to retrieve the shared long-term secret key, SK. To prevent these attacks, the authors of DBGK proposed a periodic rekeying of SK in each area. For this purpose, the authors advocate the use of a

tree-based hierarchical approach (e.g. LKH [90]), because these types of protocols achieve a reasonable computation and communication overhead (i.e. logarithmic complexity) without compromising any security property.

Protocol shortcomings Although DBGK is an efficient and secure protocol, it heavily relies on unconstrained key management servers to maintain the group keys. As such, including additional servers to improve fault tolerance would impose a high storage overhead on some constrained members of the groups. On top of this, manual intervention is required for those constrained members to store the security credentials, which might not be convenient in some IoT applications. Finally, the authors of DBGK do not consider an event where the General Key Management Server (*GKMS*) is unavailable, and given how this entity participates in important parts of the protocol, this is a major design flaw that was not addressed in its specification.

Due to these factors, this protocol's main shortcoming is the fact that it is not appropriate to be directly implemented for sensitive collaborative applications.

However, the authors developed an extension of this protocol, called **DsBGK** [2], or Distributed Batch-based Group Key, which keeps the basic functioning of DBGK, explained in Subsection 3.2.2 above, while significantly improving both fault tolerance and scalability, and removing the need for a General Key Management Server.

3.2.3 DLGKM-AC

The main idea behind DLGKM-AC is the creation of an efficient and flexible mechanism to secure the distribution of contents to eligible subscribers. The motivation and use case scenario for this protocol is a "smart" hotel, where key cards and smartphones might be interchangeably used to give access permissions to guests in different rooms. They can also be used to control the usage of various facilities according to room classes' and purchased services. When a guest checks out, and the room becomes vacant, the devices should stop sending the room's information and receiving information from other devices.

3.2.3.1 Mechanisms

In this paragraph, the mechanisms that this protocol makes use of will be presented. First, the LKH scheme is described, which is used for efficient key management of different device groups (DGs). Then, the Master Key Encryption based on the Generalized Chinese Remainder Theorem (GCRT) is presented, which is used for the management of multiple user groups (UGs) and their various users.

Logical Key Hierarchy (LKH) LKH [90] employs a tree structure to manage the distribution of keys. This method is very effective in maintaining reduced communication costs, by multicasting multiple key-encryption keys $O(log(n))$ for n devices per group. This tree structure is composed of devices located at the leaf nodes of the tree, and a central control

center called KDC, which maintains the keys' virtual tree. Each leaf node shares a secret key with the KDC. The root of the tree holds the Group Key (GK), and the internal nodes hold Key Encryption Keys (KEK).KEKs are known by each device in the leaf nodes within the same subtree rooted to a specific internal node. Additionally, KEKs compose a Path Key (PKt), which is used later to update group keys efficiently, by encrypting said group key with that Path Key.

This way, the LKH scheme is used to manage the group communication between within IoT device groups, since multiple users may subscribe to the same IoT device group, it would be more efficient if all of the devices and their users shared the same group key for encryption. Thus, the Traffic Encryption Key (TEK) is a traffic key used to encrypt data published by a device group to its subscribers. This key should be efficiently updated whenever a new user joins, or an old one leaves, to ensure both forward and backward secrecy.

Master Key Encryption (MKE) In the proposed architecture, Users subscribe to many *DGs* in the system to get data. To this effect, each user obtains all *TEKs* of the *DGs* to which it is subscribed. In this context, the concept of master key encryption (MKE) is introduced: MKE is a key management scheme based on the GCRT, which allows multiple decryption keys to decrypt the same message encrypted by an encryption key [72]. The main idea behind this scheme is the generation of a "master" key and several "slave" keys, where the master key encrypts a message that can be decrypted by all legitimate slave keys.

3.2.3.2 Overview

DLGKM-AC is composed of three layers, shown in Figure 3.6. The upper and lower layers define groups of devices (DGs) and groups of users (UGs), respectively. In contrast, the middle layer defines the decentralized controller, KDC, which is responsible for key management between, and within groups.

- *Device Groups DGs*: DLGKM-AC for IoT environments establishes a fixed number of DGs based on their functionalities, security levels, localization, etc. When a new IoT device joins the system, it is assigned to precisely one of the existing DGs. The LKH structure provides group communications within a DG.

- *User Groups UGs*: DLGKM-AC for IoT environments creates user groups (UGs) based on user's interest and reservation's period. Each user joins one of those UGs, and encryption keys are distributed using the MKE technique within a UG.

- *Decentralized Group Key Manager*: this protocol employs a decentralized architecture of servers, which is composed of one KDC and several SKDC. The number of SKDC is not fixed and depends on the requirements of the IoT application, i.e., storage, computation capacities and the number of registered users.

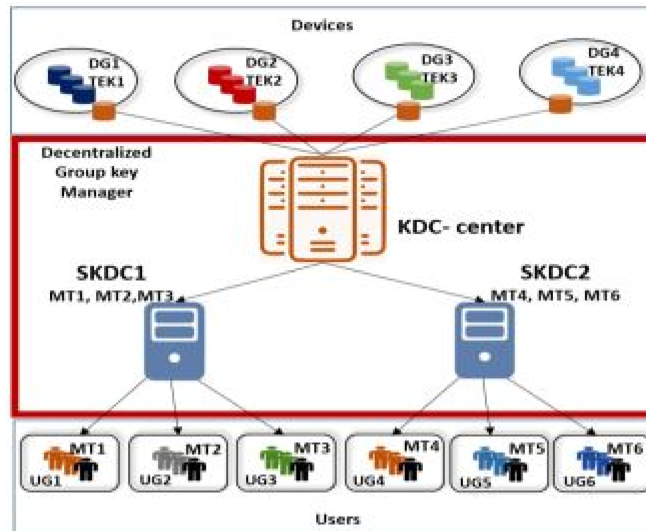


Figure 3.6: DLGKM-AC system model

These three layers correspond to the publisher, subscriber and group key manager layers, respectively, as this protocol functions essentially as a publish-subscribe protocol.

The KDC is the central server that relates publishers (e.g. IoT devices) to the remaining parts of the system while managing the keys' update process within the different *DGs*. Furthermore, the KDC maintains the last updated version of the keys in a backup server, updating it every time a rekeying process concludes. The KDC can also establish a one-time secure channel with both users and devices, which can be used to authenticate and configure a newly-joined user or device (e.g. by installing a shared secret key) before sharing the encryption keys.

SKDCs manage the group communication within *UGs*, where users may frequently leave and join the system. Multiple user groups may be under the control of one SKDC, depending on the users' location.

In this protocol, the different encryption keys used can be summarized into two categories: (i) **Traffic Encryption Keys (TEK)**, which are used to encrypt and decrypt data; and (ii) **Key Encryption Keys (KEK)**, that are used to encrypt/decrypt traffic keys, so they can be distributed securely.

3.2.3.3 Scheme Construction

There are five different stages for the DLGKM-AC protocol, as described in the protocol's original specification [19]: (1) **Initialization**, (2) **Device group registration**, (3) **User group registration**, (4) **Dynamic changes in users' membership (Join/Leave events)**, and (5) **IoT device changes (Join/Leave events)**. The implementation of these five stages is represented by the pseudocode present in Algorithms 1-8.

1) *Initialization*: This step includes both the SKDCs and KDC initialization.

a) KDC *initialization*: KDC runs the **MkeyGen** algorithm, based on GCRT, to generate a master key and several slave keys to communicate with the different SKDCs under its control. Additionally, whenever a new SKDC is added to the system, the KDC must run the same algorithm to generate a slave key for the newly joined SKDC and to update its master key. Moreover, the KDC also establishes a secure channel with devices and users and creates the device groups (DGs) and assigns UGs to SKDCs.

b) SKDC *initialization*: Each SKDC in the system also runs the **MkeyGen** algorithm, to initialize the system for multiple users. Let N be the maximum number of slave keys provided by an SKDC. First, the SKDC generates a master key (e_M, d_M) and a set of N public-private key pairs, named slave keys, $SK = \{(e_i, d_i); 1 \leq i \leq N\}$, through **MkeyGen**.

SKDC make use of a function f, which maps a pair key from a set of slave keys $0, 1$ as follows:

f : S {0,1} where

$$f : \begin{cases} f((e_i, d_i)) = 1 & ((e_i, d_i) \text{ is assigned to a user}) \\ f((e_i, d_i)) = 0 & ((e_i, d_i) \text{ is not assigned to any user}) \end{cases} \tag{3.2}$$

After the generation of the master and slave keys, SKDC initializes all pair keys using 3.2 as follows: $1 \leq i \leq N, f((e_i, d_i)) = 0$.

2) *Device Groups Registration*: Multiple IoT *DGs* are established, and each device group accommodates devices with similar attributes (i.e. security levels, location, power, etc.). *KDC* generates *KEKs* for devices in each *DG*. To this end, the KDC will set up an LKH tree, exactly as described in the Logical Key Hierarchy (LKH) paragraph.

3) User Groups Registration: In this phase, multiple user groups UG_K are built, with each of them allowing r_k users with the same purpose for a period T. Each user U_i in a UG_K is authenticated before joining the system, and shares a secret key UK_i with its SKDC. The SKDC assigns an ID to the user group, denoted by the expression: $ID_{UG_K} = \{A_{j,b} | 1 \leq j \leq M | b \in [0, 1]\}$, and using 3.3, where j refers to the device group DG_j, and b outlines the user group's subscription to that device group, when $b = 1$. Otherwise, when $b = 0$, this means that the user group is not subscribed to the corresponding DG_j.

$$ID_j = \begin{cases} A_{j,0} = 0, & UG \text{ is not subscribed to the } DG_j \\ A_{j,1} = 1, & UG \text{ is subscribed to the } DG_j \end{cases} \tag{3.3}$$

The communication within the user groups is based on Master Token Encryption (MTE). Therefore, the SKDC executes Algorithm 2, **MTokenGen**, to generate the group key MT_K and a set S_K of slave tokens STs for UG_K. Then, each user U_i in UG_K receives an ST through a secure unicast.

The SKDC adds user group information ($ID_{UG_K}, MT_K, S_K, r_K, T$) to the list of active user groups.

Algorithm 1 Master Key Generation **MKeyGen**

Inputs: A set of safe prime numbers p_1,p_2,...,p_N and q_1,q_2,...,q_N
Output: One master key e_M and N slave public-private key pairs
$S = \{(e_i, d_i); 1 \leq i \leq N\}$
$S = \{\}$
For $i = 1$ **to** N:
  $\varphi_i = (p_i - 1) \text{ x } (q_i - 1);$
  $x_i = (p_i - 1)/2;$
  $y_i = (q_i - 1)/2;$
  $e_i = 4 \text{ x } Random + 1;$
  $d_i = e_i^{2(x_i-1)(y_i-1)} - 1 \; mod \; 4x_iy_i;$
  $S = S + \{(e_i, d_i)\};$
End For
$product = 1$
For $i = 1$ **to** N:
  $product = product \text{ x } (x_i, y_i);$
End For
For $i = 1$ **to** N:
  $M[i] = n/(x_i, y_i);$
  $N[i] = M[i](x_i - 1)(y_i) - 1 \; mod \; (x_i, y_i);$
End For
$e_M = 0;$
For $i = 1$ **to** N:
  $e_M = (e_M + (e_i \text{ x } M[i] \text{ x } N[i]));$
End For

4) *User membership changes (Join/Leave)*: In order to describe the keys' update process of DLGKM-AC, and for simplicity, the only case considered is one where a user joins/leaves the user group UG_1, where users are subscribed to DG_1, DG_2, and DG_4.

a) *When a user joins a group:* Consider a user U_{join} that joins an existing group UG_K. First, U_{join} should register to the SKDC after being authenticated. Then, that user obtains a shared secret key UK_{join} from SKDC. Afterwards, the SKDC executes the **JoKeyUpdate** algorithm, shown in Algorithm 3, to update the group key MT_K, e_M and generate a new slave token for the new user U_{join}. Notice that the existing users in this group can also decrypt newly-sent messages, encrypted with the new MT_K, using their STs. Subsequently, the SKDC runs the **JoKeyDistribute** algorithm (Algorithm 4) to distribute the rekeying message to the system. In this algorithm, the SKDC first notifies the KDC about the join event, and then it notifies all of the users subscribed to the same device groups through a multicast message, in order for them to update their TEK_j through a hash function. Consequently, old users update TEK_j to reduce communication overhead. This way, the new user cannot access previously exchanged data. Finally, the SKDC sends the updated keys to the new user U_{join} through a unicast message, including his ST key.

Algorithm 2 Master Token Generation **MTokenGen**

Inputs: Number of user r, $Time\ T$, e_M, S
Output: MT_K Master Token of UG_K and list S_K
$e_{M_K} = e_M;$
$Comp = 0$
$S_K = \{\};$
SK is the list of slave keys for UG_K, of the form $S_K = \{e_i^{1K}, e_i^{2K}, ..., e_i^{rK}$, where $e_i^{1K} = e_i$ assigned to user in UG_K
While (Comp< r) do
Select a random (e_i, d_i) from $S = \{(e_i, d_i)|1 \leq i \leq N\}$
    If $f((e_i, d_i) == 0)$ **Then**
        $S_K = S_K + \{(e_i, d_i)\};$
        $f((e_i, d_i)) = 1;$
        Comp++;
    End If
End while
For $i = 1$ **to** N:
    If $e_i \notin S_K = \{e_i^{1K}, e_i^{2K}, ..., e_i^{rK}\}$ **Then**
        $e_{M_K} = e_{M_K} - e_i M[i]N[i];$
    End If
End For
$MT_K = (e_{M_K} + T)$

Algorithm 3 JoKeyUpdate

Inputs: UG_K information (ID, MT_K, S_K, r_K, T) and (e_M, S)
Output: updated $MT'_K, S'_K, r'_K, e'_M)$ and S'.
A new user joins the UG_K
Find e_i from
$S = \{(e_i, d_i); 1 \leq i \leq N\}$ where $f((e_i, d_i)) = 0$
$S'_K = \{e_i^{1K}, e_i^{2K}, ..., e_i^{rK}\} + e_i^{\{join\}K}\}$
$r'_K = r_K + 1;$
$e_{M_K} = (MT_K - T);$
$e'M_K = e_{M_K} + e_i^{\{join\}K} M[i]N[i];$
$MT_K = (e'M_K + T);$

b) When a user leaves a group: When a user U_{leave} leaves a group UG_K, it is not allowed to obtain the exchanged messages after the revocation of its cryptographic material, to ensure forward secrecy. Therefore, the SKDC runs the **LeKeyUpdate** algorithm, present in Algorithm 5, to update the group key MT_K, e_M and user group information. This algorithm achieves this by deleting the *ST* of the user that is leaving, while the remaining slave tokens are valid to decrypt data encrypted with the new MT_K. After updating the master key MT_K, the SKDC distributes the rekeying message throughout the system, with Algorithm 6. This process starts with the user announcing to the SKDC its willingness to leave the group, after which the SKDC verifies that request and notifies the KDC that

Algorithm 4 JoKeyDistribute

Inputs: $TEKs, DKs, MT$
Output: new and updated keys $ST, Ui, MT', TEKs', DKs'$
$SKDC \xrightarrow{\text{unicast}}$ **User** i : established a shared secret key with user i U_i
$SKDC \xrightarrow{\text{multicast}}$ **All:** Notify KDC, old users of the joined group and other user groups which subscribed to the same DG to update $TEK' = h(TEK)$.
$KDC \xrightarrow{\text{multicast}}$ **Devices**: update their key $DK' = h(DK)$
$SKDC$: update MT of the group that was joined
$SKDC \xrightarrow{\text{unicast}}$ **User**: $[ST_i, DKs, TEK]_{UK^i}$

Algorithm 5 LeKeyUpdate

Inputs: UG_K information (ID, MT_K, S_K, r_K, T) and system information (e_M, S)
Output: updated MT'_K, S'_K and r'_K
The ith user leaves the UG_K
$f(e_i^{\{leave\}K}) = 0$
$e_i^{\{leave\}K}$ is revoked from S_K
$S'_K = \{e_i^{1K}, e_i^{2K}, ..., e_i^{rK}\}/\{e_i^{\{leave\}K}\}$
$r'_K = r_K - 1;$
$e'_i = e_i^{\{leave\}K} = 4 \text{ x } Random + 1;$
$e_{M_K} = (MT_K - T)$
$e'_{M_K} = e_{M_K} - e_i^{\{leave\}K} M[i]N[i];$
$MT'_K = (e'_{M_K} + T);$

Algorithm 6 LeKeyDistribute

Inputs: $TEKs, DKs, MT$
Output: new generated keys $MT', TEKs', DKs'$
SKDC updates MT of the group UG from where the user left
SKDC $\xrightarrow{\text{unicast}}$ **KDC**: notify that user has left the UG
KDC $\xrightarrow{\text{multicast}}$ **SKDCs**: $(TEK'|DK')_{MK}$
KDC $\xrightarrow{\text{multicast}}$ **Devices**: $(TEK')_{GK}$
SKDC $\xrightarrow{\text{multicast}}$ **user groups** UG: $((TEK')_{MT})_{MK}$

a leave event is going to occur. Then, the KDC updates all of the TEK_j to which the user U_{leave} was subscribed to, broadcasting these new traffic encryption keys to all of the involved SKDCs. Finally, the SKDC enforces an access control level for the user group in question, using its ID_{UG}; $[TEK_j^{new}, \forall j | A_{j,b} = 1 \text{of} UG_K$. This way, according to ID_{UG}, the SKDC encrypts the updated TEK_j^{new} using the corresponding master key MT of that user group and encrypts the results with the master key of the SKDC. Consequently, the message is broadcasted to all corresponding users, but only the ones with a valid slave token ST can decrypt the new TEK_j^{new}.

Algorithm 7 DeJoKeyUpdate

Inputs: $KEKs, GK$
Output: new and updated keys $DK, D_j, KEKs', GK'$
KDC → **device** D_j: establish a shared secret key with the new device (D_j)
KDC $\xrightarrow{\text{multicast}}$ **old devices in DG**: update group key $GK' = h(GK)$
KDC $\xrightarrow{\text{unicast}}$ **devices**: update KEK' encrypted either by secret keys or shared KEK

Algorithm 8 DeLeKeyUpdate

Inputs: $KEKs, GK$
Output: new keys $KEKs', GK'$
KDC $\xrightarrow{broadcast}$ **All**: "leaving device j is no longer available"
KDC $\xrightarrow{\text{multicast}}$ **DG**: update GK' and $KEKs'$

5) ***IoT device membership changes (Join/Leave):*** To describe the next processes, the case that is considered is the following: a device group *DG1*, that publishes data to the user groups *UG1,UG3,UG4,UG6*, based on their group identity.

a) When an IoT device joins a group: For this event, an IoT device D_{join} intends to join a *DG*, which means a rekeying process is required, therefore, the KDC will run the **DeJoKeyUpdate** algorithm (Algorithm 7). Here, the KDC shares a secret key with the new device D_{join} that will join a device group DG_y. After, the KDC updates the necessary (affected) part of the LKH tree in which the device resides, multicasting to the existing devices a notification signalling them to update the group key *GK*. Finally, the KDC sends the PK_t and TEK of the DG_y to the new device, through unicast.

b) When an IoT device leaves a group: When a device D_{leave} leaves a device group, the KDC rearranges the LKH tree structure for that group and runs the **DeLeKeyUpdate** algorithm, shown in Algorithm 8. Here, the KDC multicasts an updated group key *GK'* to the remaining devices of the group encrypted with *KEKs*, which defines the new LKH tree structure for that group. Then, the KDC broadcasts a message to announce that the device D_{leave} is no longer a valid entity in the system.

3.2.4 Protocol Shortcomings

Although the authors guarantee that their protocol is inherently secure, guaranteeing most of the desired security properties, it does not protect against most active attacks (it only envisions collusion and masquerading attacks in its threat model) nor does it explicitly provide post-compromise security, which was determined as a key property for the chosen protocol.

Furthermore, the security protocol of DLGKM-AC is excessively intricate and hard to comprehend. Additionally, its system model is also perplexing and complicated.

3.2.5 DCGKA

This protocol's specification, described in [86], makes use of some underlying modules and services, which require an introduction before presenting the whole protocol implementation.

3.2.5.1 Building Blocks

Authenticated Causal Broadcast (ACB) This mechanism is responsible for ensuring a form of eventual consistency in the system called causal consistency. In a decentralized setting, such as this one, the same messages may arrive in different orders at different users. While all users may not receive all messages in the same order, this protocol chooses to provide a weaker ordering guarantee, called *causal broadcast*.

Definition 1 The *causal order* is a partial order $\prec$ on messages. $m_1 \prec m_2$ (m_1 causally precedes m_2) if one of the following holds:

- m_1 and m_2 were sent by the same group member, and that member sent m_1 before sending m_2;
- m_2 was sent by group member x and m_1 was received and processed by x before sending m_2;
- there exists m_3 such that $m_1 \prec m_3$ and $m_3 \prec m_2$;

Additionally, it can be said that m_1 and m_2 are concurrent, when $m_1 \nprec m_2$ and $m_2 \nprec m_1$.

Before processing m, causal broadcast requires that all preceding messages, m', are processed $\{m' | m' \prec m\}$. To implement causal broadcast, this module authenticates the sender of each message, as well as its causal ordering metadata, through a digital signature under the sender's identity key. This prevents passive attacks, such as impersonation attacks, or affecting the causally-ordered delivery.

Two types of messages are used in this protocol: *broadcast messages* are sent to every group member, while *direct messages* are sent to a specific recipient. This distinction is purely for efficiency purposes, since the protocol's security properties remain unaltered, regardless of who receives a message.

Decentralized Group Membership (DGM) In a decentralized setting, it is not always clear who the current group members are. For example, if a user A removes user B from the group, while B removes A concurrently. Some users may first process A's removal of B and then ignore B's operation (because B is no longer a group member at that point), while other users may first process B's removal of A and then ignore A's operation. These types of situations are common occurrences in distributed computing, especially in decentralized architectures, creating the need to handle them carefully.

This protocol achieves this through a *Decentralized Group Membership (DGM)* function n, that takes a set of membership change messages and their causal order relationships as arguments, and returns the current set of group members. The results of this function must be deterministic and depend only on the causal order. This function may take permissions into account (i.e., if the group has an administrator, only allow him to add or remove members).

Two-Party Secure Messaging (2SM)

Definition 1 A bidirectional *two-party secure messaging* scheme composed of three algorithms: 2SM-Init, 2SM-Send and 2SM-Receive.

Initialization: 2SM-Init(ID_1, ID_2) takes two IDs: ID_1 is the local user, and ID_2 is the other party. It returns an initial protocol state σ. The 2SM protocol must use a Public Key Infrastructure (PKI) or key server to map these IDs to public keys. In practice, the PKI should include ephemeral "prekeys", as introduced by Signal [63]. This allows users to send messages to a new group member, even if that member is currently offline.

Send: 2SM-Send(σ, m) takes a state σ and a plaintext message m, and outputs a new state σ' and a ciphertext c.

Receive: 2SM-Receive(σ, c) takes a state σ and a ciphertext c, and outputs a new state σ' and a plaintext message m.

The Signal protocol is a popular implementation of 2SM, but the authors of DCGKA deemed it insufficient for their purposes, and used a different protocol, with better security guarantees, specified in [51].

3.2.5.2 The DCGKA Abstraction

DCGKA's scheme can be broadly described by six different algorithms: initialization, group creation, member addition, member removal, PCS updates and message processing. With the exception of the initialization algorithm, all algorithms take a state γ as one of their arguments:

- **Initialization**: The function init(ID) takes the ID of the current user, and returns an initial state γ.
- **Group Creation**: The function create(γ,ID) takes a state γ and a set of user IDs, and creates a new group with those members.
- **Member Addition**: The function add(γ, ID) takes a state γ and a user ID, and adds that user to the group.
- **Member removal**: The function remove(γ, ID) takes a state γ and a user ID, and removes that user from the group.
- **PCS Update**: The function update(γ) takes a state γ and performs a key update.

- **Message processing**: The function process(γ,sender, control, dmsg) is called when a control message is received. It takes a state γ, the user ID of the message *sender* (authenticated as discussed in the paragraph related to ACB), a control message *control*, and a direct message *dmsg* (or an empty value if there is no associated direct message).

3.2.5.3 The DCGKA protocol

The full pseudocode specification of this protocol is present in Figure 3.7. The variable γ denotes the state of the system, which consists of the variables initialized in the function **init**. The notation $2sm[\cdot] \leftarrow \varepsilon$ means that 2sm is a dictionary where every key is initially mapped to the default value ε, representing the empty string.

Every control message is a triple of the form (type, seq, content). The message type must be one of the following: "create", "ack", "update", "remove", "add" or "add-ack". The *seq* field is a sequence number that consecutively numbers successive control messages from the same sender. The *content* depends on the type of the message. The **process** function unpacks the tuple and then calls one of the six functions, **process-create**, **process-ack**, **process-update**, **process-remove**, **process-add**, or **process-add-ack**, depending on the message type.

Helper functions The protocol makes use of several distinct helper functions, that directly influence the main functions.

encrypt-to uses 2SM to encrypt a *direct* message for another group member. The first time a message is encrypted to a particular recipient ID, the 2SM protocol state is initialized on line 2 and stored in $\gamma.2sm[ID]$. Then, the 2SM-Send function is called on line 4 to encrypt the message, and store the updated protocol state in γ.

decrypt-from is the reverse of **encrypt-to**. It similarly initializes the protocol state on first use, and then uses 2SM-Receive to decrypt the ciphertext, with the protocol state saved in $\gamma.2sm[ID]$.

update-ratchet generates the next update secret for the group member pertaining to the given ID. The ratchet state is stored in γ*.ratchet[ID]*; an HMAC-based key derivation function HKDF [58, 60] is used to combine the ratchet state with an input, producing an update secret and a new ratchet state.

member-view computes the set of group members at the time of the most recent control message sent by user ID. It first filters the set of group membership operations to obtain only those seen by user ID, and then calls the Decentralized Group Membership (DGM) function to compute the group's membership.

generate-seed generates a seed secret using *KGen*, a secure source of random bits, and then calls **encrypt-to** to encrypt the seed secret for each group member using the 2SM protocol. It returns the new, updated, protocol state and the set of *direct* messages to send.

```
init(ID)
1 : γ.myId ← ID
2 : γ.mySeq ← 0
3 : γ.history ← ∅
4 : γ.nextSeed ← ε
5 : γ.2sm[·] ← ε
6 : γ.memberSecret[·, ·, ·] ← ε
7 : γ.ratchet[·] ← ε
8 : return γ

process(γ, sender, controlMsg, dmsg)
1 : (type, seq, info) ← controlMsg
2 : if type = "create" then
3 :    return process-create(γ, sender, seq, info, dmsg)
4 : else if type = "ack" then etc . . .

create(γ, IDs)
1 : control ← ("create", ++γ.mySeq, IDs)
2 : (γ, dmsgs) ← generate-seed(γ, IDs)
3 : (γ, _, _, I, _) ← process-create(γ, γ.myId, γ.mySeq, IDs, ε)
4 : return (γ, control, dmsgs, I)

process-create(γ, sender, seq, IDs, dmsg)
1 : op ← ("create", sender, seq, IDs)
2 : γ.history ← γ.history ∪ {op}
3 : return process-seed(γ, sender, seq, dmsg)

process-ack(γ, sender, seq, (ackID, ackSeq), dmsg)
1 : if (ackID, ackSeq) was a create/add/remove then
2 :    op ← ("ack", sender, seq, ackID, ackSeq)
3 :    γ.history ← γ.history ∪ {op}
4 : s ← γ.memberSecret[ackID, ackSeq, sender]
5 : γ.memberSecret[ackID, ackSeq, sender] ← ε
6 : if s = ε ∧ dmsg = ε then return (γ, ε, ε, ε, ε)
7 : if s = ε then (γ, s) ← decrypt-from(γ, sender, dmsg)
8 : (γ, I) ← update-ratchet(γ, sender, s)
9 : return (γ, ε, ε, I, ε)

update(γ)
1 : control ← ("update", ++γ.mySeq, ε)
2 : recipients ← member-view(γ, γ.myId) \ {γ.myId}
3 : (γ, dmsgs) ← generate-seed(γ, recipients)
4 : (γ, _, _, I, _) ← process-update(γ, γ.myId, γ.mySeq, ε, ε)
5 : return (γ, control, dmsgs, I)

process-update(γ, sender, seq, _, dmsg)
1 : return process-seed(γ, sender, seq, dmsg)

remove(γ, ID)
1 : control ← ("remove", ++γ.mySeq, ID)
2 : recipients ← member-view(γ, γ.myId) \ {ID, γ.myId}
3 : (γ, dmsgs) ← generate-seed(γ, recipients)
4 : (γ, _, _, I, _) ← process-remove(γ, γ.myId, γ.mySeq, ID, ε)
5 : return (γ, control, dmsgs, I)

process-remove(γ, sender, seq, removed, dmsg)
1 : op ← ("remove", sender, seq, removed)
2 : γ.history ← γ.history ∪ {op}
3 : return process-seed(γ, sender, seq, dmsg)

add(γ, ID)
1 : control ← ("add", ++γ.mySeq, ID)
2 : (γ, c) ← encrypt-to(γ, ID, γ.ratchet[γ.myId])
3 : op ← ("add", γ.myId, γ.mySeq, ID)
4 : welcome ← (γ.history ∪ {op}, c)
5 : (γ, _, _, I, _) ← process-add(γ, γ.myId, γ.mySeq, ID, ε)
6 : return (γ, control, {(ID, welcome)}, I)

process-add(γ, sender, seq, added, dmsg)
1 : if added = γ.myId then
2 :    return process-welcome(γ, sender, seq, dmsg)
3 : op ← ("add", sender, seq, added)
4 : γ.history ← γ.history ∪ {op}
5 : if γ.myId ∈ member-view(γ, sender) then
6 :    (γ, s) ← update-ratchet(γ, sender, "welcome")
7 :    γ.memberSecret[sender, seq, added] ← s
8 :    (γ, I_sender) ← update-ratchet(γ, sender, "add")
9 : else I_sender ← ε
10 : if sender = γ.myId then return (γ, ε, ε, I_sender, ε)
11 : control ← ("add-ack", ++γ.mySeq, (sender, seq))
12 : (γ, c) ← encrypt-to(γ, added, ratchet[γ.myId])
13 : (γ, _, _, I_me, _) ← process-add-ack(γ, γ.myId,
14 :       γ.mySeq, (sender, seq), ε)
15 : return (γ, control, {(added, c)}, I_sender, I_me)

process-add-ack(γ, sender, seq, (ackID, ackSeq), dmsg)
1 : op ← ("ack", sender, seq, ackID, ackSeq)
2 : γ.history ← γ.history ∪ {op}
3 : if dmsg ≠ ε then
4 :    (γ, s) ← decrypt-from(γ, sender, dmsg)
5 :    γ.ratchet[sender] ← s
6 : if γ.myId ∈ member-view(γ, sender) then
7 :    (γ, I) ← update-ratchet(γ, sender, "add")
8 :    return (γ, ε, ε, I, ε)
9 : else return (γ, ε, ε, ε, ε)

process-welcome(γ, sender, seq, (adderHistory, c))
1 : γ.history ← adderHistory
2 : (γ, γ.ratchet[sender]) ← decrypt-from(γ, sender, c)
3 : (γ, s) ← update-ratchet(γ, sender, "welcome")
4 : γ.memberSecret[sender, seq, γ.myId] ← s
5 : (γ, I_sender) ← update-ratchet(γ, sender, "add")
6 : control ← ("ack", ++γ.mySeq, (sender, seq))
7 : (γ, _, _, I_me, _) ← process-ack(γ, γ.myId, γ.mySeq,
8 :       (sender, seq), ε)
9 : return (γ, control, ε, I_sender, I_me)

generate-seed(γ, recipients)
1 : γ.nextSeed ←$ KGen; dmsgs ← ∅
2 : foreach ID ∈ recipients do
3 :    (γ, msg) ← encrypt-to(γ, ID, γ.nextSeed)
4 :    dmsgs ← dmsgs ∪ {(ID, msg)}
5 : return (γ, dmsgs)

process-seed(γ, sender, seq, dmsg)
1 : recipients ← member-view(γ, sender) \ {sender}
2 : if sender = γ.myId then
3 :    seed ← γ.nextSeed; γ.nextSeed ← ε
4 : else if γ.myId ∈ recipients then
5 :    (γ, seed) ← decrypt-from(γ, sender, dmsg)
6 : else
7 :    return (γ, ("ack", ++γ.mySeq, (sender, seq)), ε, ε, ε)
8 : foreach ID ∈ recipients do
9 :    γ.memberSecret[sender, seq, ID] ← HKDF(seed, ID)
10 : senderSecret ← HKDF(seed, sender)
11 : (γ, I_sender) ← update-ratchet(γ, sender, senderSecret)
12 : if sender = γ.myId then return (γ, ε, ε, I_sender, ε)
13 : control ← ("ack", ++γ.mySeq, (sender, seq))
14 : members ← member-view(γ, γ.myId)
15 : forward ← ∅
16 : foreach ID ∈ members \ (recipients ∪ {sender}) do
17 :    s ← γ.memberSecret[sender, seq, γ.myId]
18 :    (γ, msg) ← encrypt-to(γ, ID, s)
19 :    forward ← forward ∪ {(ID, msg)}
20 : (γ, _, _, I_me, _) ← process-ack(γ, γ.myId, γ.mySeq,
21 :       (sender, seq), ε)
22 : return (γ, control, forward, I_sender, I_me)

encrypt-to(γ, recipient, plaintext)
1 : if γ.2sm[recipient] = ε then
2 :    γ.2sm[recipient] ← 2SM-Init(γ.myId, recipient)
3 : (γ.2sm[recipient], ciphertext) ←
4 :       2SM-Send(γ.2sm[recipient], plaintext)
5 : return (γ, ciphertext)

decrypt-from(γ, sender, ciphertext)
1 : if γ.2sm[sender] = ε then
2 :    γ.2sm[sender] ← 2SM-Init(γ.myId, sender)
3 : (γ.2sm[sender], plaintext) ←
4 :       2SM-Receive(γ.2sm[sender], ciphertext)
5 : return (γ, plaintext)

update-ratchet(γ, ID, input)
1 : (updateSecret, γ.ratchet[ID]) ←
2 :       HKDF(γ.ratchet[ID], input)
3 : return (γ, updateSecret)

member-view(γ, ID)
1 : ops ← {m ∈ γ.history | m was sent or acked by ID
2 :    (or the user who added ID, if m precedes the add)}
3 : return DGM(ops)
```

Figure 3.7: Full DCGKA protocol specification

Group creation A group is created in three steps:

1. A user calls the **create** function and broadcasts a *control* message of type "create", in addition to *direct* messages to the initial members.

2. Each member processes that broadcasted message and proceeds to broadcast their own "ack" *control* message.

3. Each member processes the acknowledgement message from the other members.

The **create** function calls **generate-seed** to obtain the set of *direct* messages to send. It then calls the **process-create** function to process the *control* message for this user (as if it had also received the message) before returning. From the response of **process-create**, the updated state γ and the update secret I are used, while the rest of the tuple can be ignored.

process-create is called both by the sender and each recipient of the "create" control message. It first records the information from the create message in γ.*history*, which is used to track group membership changes, and then calls **process-seed**.

process-seed first employs **member-view** to determine the set of users that were group members at the time the *control* message was sent, i.e., the recipients of the message. Then, it attempts to obtain the seed secret, where there are 3 possible scenarios:

- if the *control* message was sent by the local user, the last call to **generate-seed** will have placed the value in γ.*nextSeed*, and, as such, that variable is read and its contents are then deleted (lines 2-3);

- if the *control* message was sent by another user, and the local user is one of the recipients, then **decrypt-from** is used to decrypt the *direct* message containing the seed secret (lines 4-5);

- otherwise, an "ack" message is returned, without deriving an update secret;

Next, independent *member secrets* are derived from the seed secret, for each group member.The secret for the sender of the message is stored in the variable *senderSecret*, and those for the other group members are stored in γ.*memberSecret*; the latter are used when receiving acknowledgements from those users. Only the member secrets are stored, and not the seed secret, so that if the user's private state is compromised, the adversary obtains only those member secrets that have not yet been used. The sender's member secret is used immediately to update their KDF ratchet and compute their update secret I_{sender} (line 11), using **update-ratchet**. If the local user is the sender of the control message, the user is now finished and returns the update secret (line 12). If the seed secret was sent from another user, an "ack" control message is broadcasted, including the sender ID and sequence number of the message the user is acknowledging (line 13). The last step here is to compute an update secret for the local user, I_{me}, which is done through the function **process-ack**.

process-ack is also called by other group members when they receive the "ack" message. In this function, *ackID* and *ackSeq* are the sender and the sequence number of the acknowledged message, respectively. If this message was a group membership operation,

this is recorded in γ*.history* (lines 1-3). Next, the value of γ*.memberSecret* is read and deleted for forward secrecy purposes. Finally, on lines 8-9, the ratchet is updated for the sender of the "ack" message and the resulting update secret is returned.

PCS Update and Removing Group members The functions **update** and **remove** are similar to **create**: they also call **generate-seed** to encrypt a new seed secret for each group member. The main difference between these two functions is that the set of group members for this operation is obtained using **member-view**, but in the case of **remove**, the user being removed is excluded from the set of recipients of the new secret. Moreover, the *control* message they build will have the type "update" or "remove", respectively.

Afterwards, the **process-update** and **process-remove** functions are called, where the former will omit the update of γ*.history*, and the latter will add a remove operation to that same variable. Both will then call **process-seed**, which will execute just like during group creation.

Adding group members To add a new group member, an existing member calls the **add** function, passing the ID of the user to be added as an argument to that function. This function builds a *control* message with the type "add", that will be broadcasted to the whole group, and a *welcome* message to the newly-introduced member of the group, sent directly to it. This welcome message contains the current KDF ratchet state of the sender, encrypted with the 2SM algorithm, and the history of the group operations to date.

process-add will be called by the sender and each recipient of an "add" message, including the new group member. If the local user is the one being added to the system, the function **process-welcome** is called and returns. Otherwise, γ*.history* is extended with the new add operation. Following this, the ratchet of the sender is updated twice with **update-ratchet**, where the value returned by the first call is stored in γ*.memberSecret* as the added user's first member secret, while the second value returned becomes I_{sender}, the update secret for the sender of the "add" message. Next, if the local user is the sender, the update secret is returned. If this is not the case, however, an acknowledgement of the "add" message is required. A *control* message of the type "add-ack" is constructed, and the current ratchet state is encrypted with 2SM and sent via a *direct* message to the added user so that they can decrypt subsequent messages. Finally, **process-add-ack** is called to compute the local user's update secret, I_{me}, and return it with I_{sender}.

process-welcome is the second function called by a newly added user (the first is the init function, to set the user variables' initial values). In this function, *adderHistory* is the adding user's copy of γ*.history* sent in their welcome message, which is used to initialize the new user's history. c is the ciphertext of the adding user's ratchet state, which is decrypted on line 2 using **decrypt-from**. After γ*.ratchet[sender]* is initialized, a call to **update-ratchet** is made twice on lines 3–5 with the constant strings "welcome" and "add": the same ratchet operations as every other group member performs in **process-add**, producing the same results as well.

Finally, the new group member builds an "ack" *control* message to broadcast to the group, and calls **process-ack** to compute their first update secret I_{me}, as described previously. The only difference in this case is that the previous ratchet state for the new member is the empty string ε, as set up by **init**, so this step initializes the new member's ratchet. Every other group member, upon receiving the new member's "ack" message, will initialize their copy of the new member's ratchet in the same way. By the end of **process-welcome**, the new group member has obtained update secrets for themselves and the user who added them.

The ratchets for other group members are initialized by **process-add-ack**. This function is called by both the sender and each recipient of an "add-ack" message, including the new group member. This acknowledgement is added to $\gamma.history$. If the current user is the new group member, the "add-ack" message is joined by the direct message that was constructed in **process-add**; this direct message *dmsg* contains the encrypted ratchet state of the sender of the "add-ack" message, so it is decrypted on lines 3–5. On line 6 of **process-add-ack**, a check is made to confirm that the local user was already a group member at the time the "add-ack" was sent (which may not be the case when there are concurrent additions). If so, on line 7, a new update secret I is computed for the sender of the "add-ack" by calling **update-ratchet** with the constant string "add". In the case of the new member, the ratchet state was just previously initialized on line 5. This ratchet update allows all group members, including the new one, to derive each member's update secret for the add operation, but it prevents the new group member from obtaining any update secret from before they were added, effectively guaranteeing backwards secrecy.

3.2.5.4 Protocol shortcomings

In regards to security, DCGKA is seemingly "perfect", in the sense that it provides all of the security properties deemed relevant and that there are no evident concerns. The only main downside of the protocol is that it uses several different modules, with specific implementations and interactions, that make the overall system slightly complex. But even in regards to consistency and efficiency, the protocol seems to perform decently well, as tested by the authors.

3.2.6 Conclusion

Upon extensively researching and analysing these protocols' implementations, the one that seems the most promising for the context of this thesis is the DCGKA protocol. This protocol is open-source, and although its implementation is slightly complex, it can efficiently provide all of the security properties presented in Section 2.2. The remaining protocols explained in this section (Subsection 3.2.1 to Subsection 3.2.3) continue to be a valid choice, however: the base version of Matrix (i.e., without any changes to its source code) does not provide perfect forward secrecy (only partial) and backward secrecy, which is a fundamental security property for the system this thesis intends to develop; the

DBGK protocol is as secure as DCGKA, but displays some efficiency concerns if additional fault tolerance is required of the system; and the DLGKM-AC protocol makes use of an overly-complicated architecture and functioning, reason which it was not chosen for this purpose.

3.3 Preparation

With the DCGKA protocol now chosen, its implementation could now be adapted to this thesis' specific use case.

The DCGKA protocol specification is materialized in a full Java implementation, organized as a multi-project Gradle build. The authors' prototype implements a "fully decentralized secure group messaging protocol, including application message encryption, signatures, keys, etc.", even though the paper mostly focuses on the key agreement aspect of the protocol.

At this point, the DCGKA protocol seemed increasingly promising, however, some issues quickly arose:

- Even though the available implementation fully covers the protocol specification, it was intended purely for academic purposes, and therefore was not subject to security testing, making it unsuitable for production;
- Aside from the mentioned repository, and the paper where the protocol is described, there is no documentation and work regarding the protocol, making it somewhat difficult to debug any eventual issues or complications with testing;

Although these issues do not make the protocol impossible to make use of for this thesis, they still represent a considerable time constraint to work around, made even more difficult by the fact that none of the code was created specifically for this thesis, nor was it developed by us. This would mean that a significant portion of the time would have to be spent analyzing and researching the DCGKA protocol implementation, or building one from scratch, before even starting to develop the support for offline communication and testing. These shortcomings ultimately led to the abandonment of this protocol and the exploration of other options presented in Chapter 3, namely, the Matrix protocol.

Given the practical limitations found with the usage of the DCGKA protocol, a new analysis of the remaining protocols discussed in Chapter 3 was necessary. Of the other three protocols that were considered the best options aside from DCGKA, only one had been implemented in a production environment, which was Matrix. Moreover, the DLGKM-AC and the DBGK protocols display some shortcomings that disincentivized their use, discussed in Sections 3.2.3 and 3.2.2, respectively. The DLGKM-AC protocol uses an overly complex architecture and specification and does not provide protection against some common types of active attacks, nor does it provide post-compromise security. The DBGK protocol suffers from similar issues: it cannot guarantee PCS (although it can

protect against active attacks), with the added penalty of displaying some scalability and efficiency concerns, and not addressing a major design flaw in its specification.

As such, the Matrix protocol was the obvious choice, since, contrarily to DCGKA and the other alternatives, Matrix is an open-source project with extensive documentation and personal work already developed, which is a significant incentive towards its use. There are several Matrix server implementations to choose from (although most are still in their beta versions), and a wide range of client implementations, varying in their use cases, programming languages on which they are written, and functionality. This means that, if Matrix were to be the chosen protocol over DCGKA, a decision would have to be made regarding these implementations, deciding whether to use existing ones or develop new ones from scratch. There are also other factors to consider, such as the architectures of the machines and devices used, since they may not be compatible with the developed software.

Additionally, as explained previously and unlike DCGKA, Matrix is not able to guarantee all of the desired security properties, and, as such, the first step in understanding this protocol was analyzing and testing an end-to-end interaction between a real Matrix client implementation and a Matrix server, to later infer how those security shortcomings could be minimized or entirely nullified.

For this purpose, a Matrix server, with both a reverse proxy container in front of it and a database container behind it, was deployed in a private Docker network, alongside several containers running a client implementation, to emulate the proposed real-world use case. The server implementation used was Synapse [31], the most commonly used home server inside the Matrix ecosystem, written by the Matrix.org Foundation. Similarly, the client implementation used was Element [23], which is also the most common client among all Matrix users and written by the Matrix team.

This experiment yielded positive results regarding both efficiency and message security, confirming that Matrix is indeed a very competent alternative to use in the prototype's implementation.

However, Element's properties and use cases do not quite cover what the prototype requires (especially concerning offline communication), and as such, an entirely new client implementation had to be developed, to correctly and efficiently meet these contextual demands.

4

System Model and Architecture

After the evaluation of the selected protocol was made in section 3.3, discussing its suitability for implementation, in this chapter, the system's scope will be narrowed down (section 4.1). Then, the architecture for the designed system is proposed (section 4.2), followed by an explanation of every component participating in the system, zooming in, particularly on their communication. This description is followed by the assumed thread model (section 4.3), and what countermeasures the system design employs to contradict them. Finally, a summary of the chapter is given (section 4.4).

4.1 Use case and context-dependent constraints

As stated previously in Chapter 1, the prototype is meant to be inserted into the context of a smart city, where device heterogeneity and imbalanced Wi-Fi coverage are the most prominent constraints, creating small clusters of devices.

The aim of this system is Intrusion Detection Systems (IDS), but its main goal will not be detecting such intrusions. Its primary focus will be on secure communication of these intrusions to all participants, regardless of their online or offline status. Ensuring availability is crucial to detect and communicate intrusions system-wide, regardless of network status. Furthermore, given the system will be an IoT system, devices will often be offline, making this feature even more of a necessity.

Therefore, the implementation will prioritize the security, availability and efficiency of communications, rather than the content being communicated.

4.2 Architecture

The architecture of the proposed system will consist of a small number of groups of devices, each with 5 to 6 devices, scattered around a designated area (for instance, a city, in the context of a smart city), where each group's devices are connected to a server responsible only for that group. While not the central focus of this project, it is important to note that for the architecture to be truly decentralized, all servers within the group

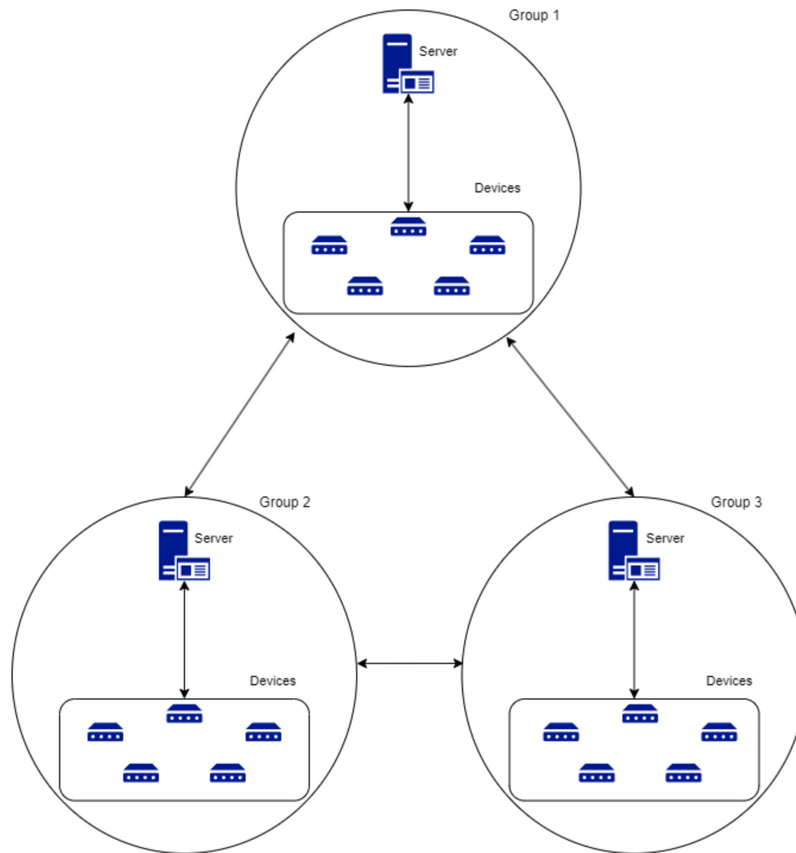


Figure 4.1: The proposed system architecture for the prototype

must communicate with each other. This ensures that there are no individual points of control over conversations or the network as a whole.

However, servers in a Matrix application only communicate if there are users connected to different servers in the same room so that they can "federate", or synchronize that room's communication history, periodically, as explained in Section 3.2.1.1.

Therefore, for the proposed architecture to be fully decentralized, there must be a chat room where there are users from different device groups, and, by extension, different homeservers. This detail is not envisioned in the prototype implementation or the testing process and shall be discussed in Section 7.1.

The mentioned architecture is displayed in Fig.4.1. This type of architecture is relatively common in decentralized group communication protocols, with some of the protocols reviewed in Section 3.1 using this type of organization as well, therefore making it a logical choice for this thesis.

4.2.1 Server

The system's server component will use a deployment similar to Fig.4.2, with each box representing a separate Docker container. Communication between the four containers

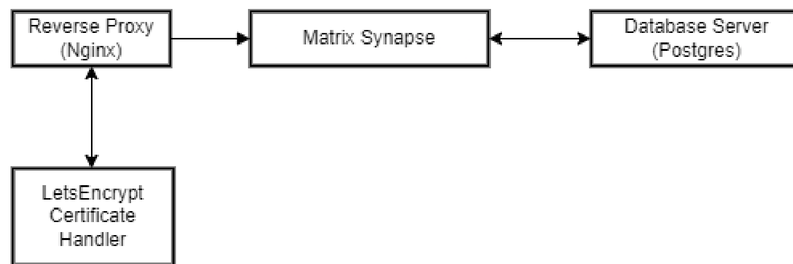


Figure 4.2: The organization of the server component in each group

will take place within a private Docker network.

Each instance of Matrix Synapse runs with a reverse proxy in front of it, and a database server behind it. In this case, the reverse proxy that was used was Nginx [69, 84], with a Let's Encrypt [1, 89] container alongside it, for TLS certificates, to allow communication through HTTPS. Furthermore, the Postgres [40] database server also runs on a separate container, continuously communicating with the Matrix server.

This is the recommended deployment for Synapse, as suggested by its authors. Clients typically connect to the server through the reverse proxy, by connecting to port 443 (the HTTPS port), which then relays the client's request to the Matrix server.

4.2.2 System Communication

Communication inside the system will mostly happen through HTTPS requests, as specified by the Matrix API [27]. The only exception to this rule is whenever a client device goes offline, which can be due to numerous causes, another client device will attempt to send to it any events that it missed while it could not communicate through the usual means.

4.2.2.1 Offline Communication

To communicate in such a way, clients will use the libp2p networking stack [47], a peer-to-peer (P2P) networking framework that enables the development of P2P applications. It consists of a collection of protocols, specifications, and libraries that facilitate P2P communication between network participants.

To discover other client hosts in the group's private network, the mDNS [15] service will be used. It allows peers to discover each other when on the same local network with zero configuration, by using a multicast system of DNS records; this allows all peers on the local network to see all query responses. Conceptually, it is a simple process: When a peer starts (or detects a network change), it sends a query to all peers. As responses arrive, the peer adds the other peers' information into its local database of peers. Figure 4.4 illustrates how the mDNS service operates in practice, with a) the query broadcast to the network by Client 1; and b) the response from all the other hosts, upon receiving that query.

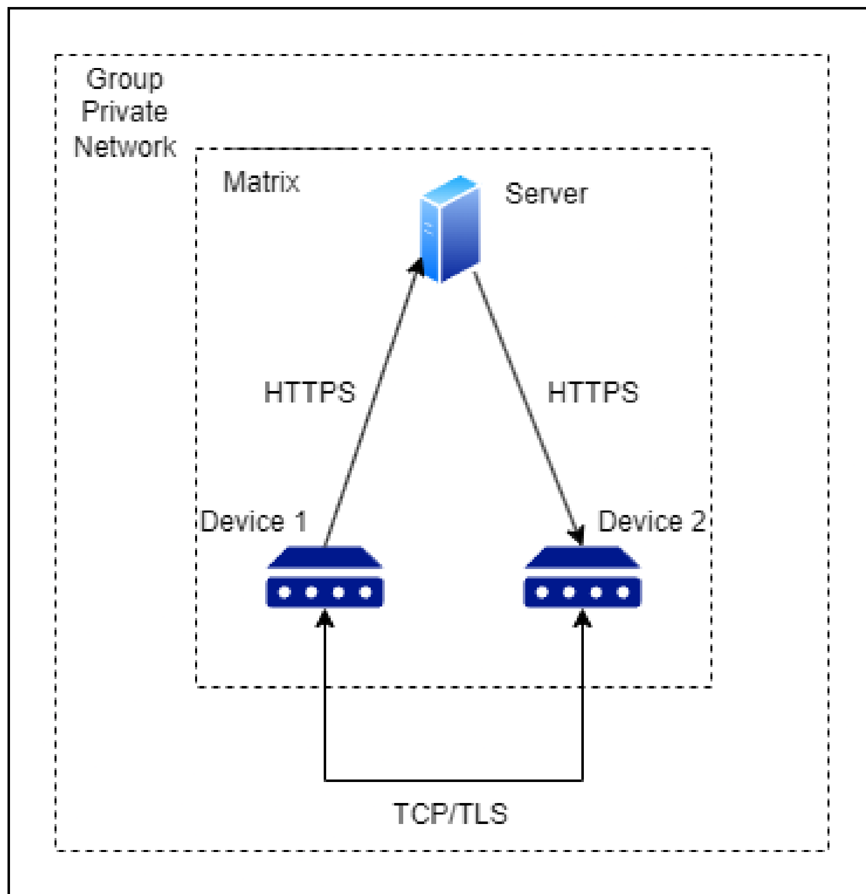


Figure 4.3: Schema defining how the Matrix clients and the server communicate

From this schema, it is clear that the mDNS host discovery protocol displays a tendency to generate a considerable amount of network traffic since every query is broadcasted to the network. To minimize the traffic, there is an mDNS cache in every client host, as seen in Fig.4.4, to store the addresses for every known peer in the network as they are discovered, thus minimizing the number of broadcasts and subsequent network traffic produced.

Once a host has been discovered, a secure channel is then established, performing either a TLS [75] or a Noise [68, 73] handshake protocol over a TCP channel (TLS in this case). During the said handshake, both peers authenticate each other's peer ID, by encoding their public key into a X.509 certificate extension. To prove ownership of the used host key, a host sends two values for its "Key Share": the public host key, and a signature performed using the private host key. The user generates the certificate using its public host key and signs it with the private host key[1].

The public host key allows the other peer to calculate the peer ID of the peer it is connecting to. Clients MUST verify that the peer ID derived from the certificate matches

[1]The public host key and the signature are ANS.1-encoded into a SignedKey data structure, which is carried in the libp2p Public Key Extension. The libp2p Public Key Extension is a X.509 extension, allocated by IANA to the libp2p project at Protocol Labs.

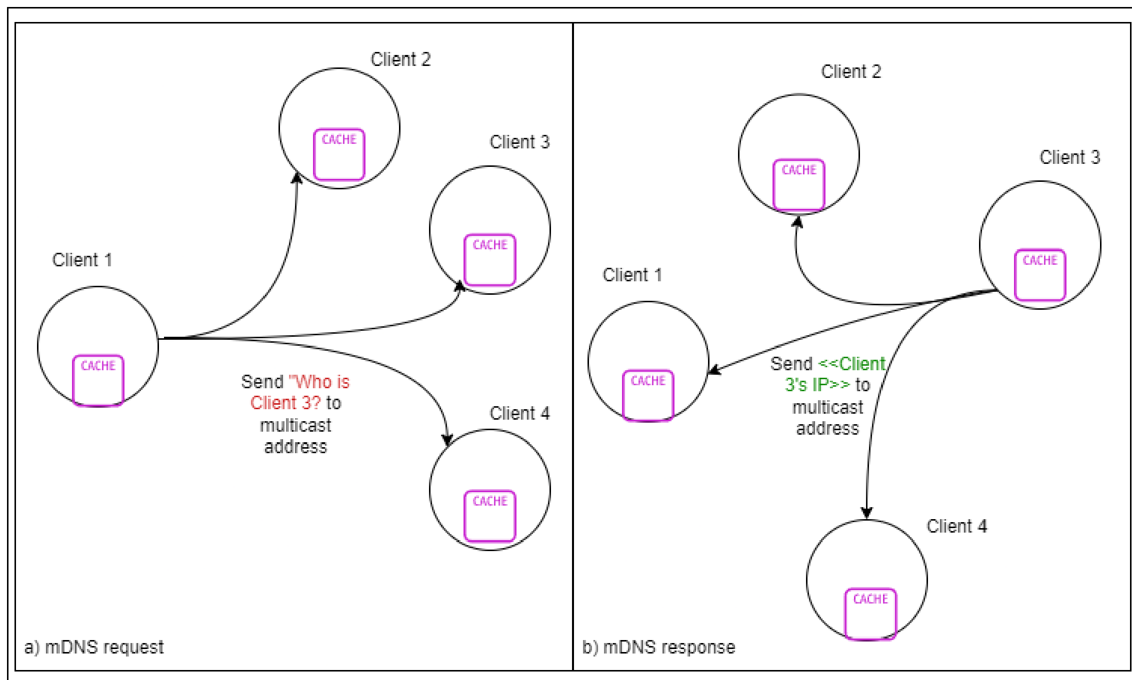


Figure 4.4: Example for an mDNS host discovery service with a) the query broadcasted to the system and b) the response for that query, returned by every other client in the network

the peer ID they intended to connect to and MUST abort the connection if there is a mismatch. In the positive case, the handshake is finished and the secure communication channel is established.

Figure 4.5: TLSv1.3 Handshake protocol used by libp2p

From this point on, the clients begin exchanging data following an execution flow similar to Matrix's: the two hosts exchange Matrix credentials, check if they have an existing Olm session between them, and encrypt the intended message with their Megolm session key.

4.3 Adversary Model

In this section, a well-defined adversary model and the possible threats that such an attacker could use against the solution are presented.

All of the biggest vulnerabilities found in Matrix [64] require the cooperation of the homeserver, and, as such, were considered out of scope for this threat model, since the implementation only focuses on the client side of the system. Similarly, all attacks that focus on the collusion of two or more homeservers are also out of the scope of this thesis.

Furthermore, the Matrix application implementation only includes the minimum necessary functionalities and communication between clients and servers occurs within a private network in the developer's control, which greatly reduces the possible attack surfaces.

However, Matrix clients in this context are still subject to some types of attacks. Each device group assumes the same threat model, where an attacker can:

- Use a replay of messages from a different context into the intended (or original and expected) context, thereby fooling the honest participant(s) into thinking they have completed the protocol run, i.e., a replay attack;

- Impersonate a user to try to receive another user's encryption keys and thus compromise an entire room's cryptography;

- Intercept the communications between two clients, when one of them is offline (a Man-in-the-middle attack), secretly relaying the communications between the two parties who believe that they are directly communicating with each other;

Replay Attack Replay attacks in this threat model can happen in two distinct situations: when, or if, they use "fallback" keys, which are cryptographic keys (Curve25519 keys, signed by the device's Ed25519 signing key) that are used to prevent one-time key exhaustion when devices are offline or unable to upload additional one-time keys; and whenever they receive a message encrypted with a Megolm session key if they do not keep track of the session's ratchet indices.

Firstly, sessions started with fallback keys are susceptible to replay attacks, since the keys are not consumed after their usage, unlike one-time keys. Therefore, to guard against this, while the client device is still online, the server informs it about the number of one-time keys remaining that can be claimed, as well as whether the fallback keys have been used. This way, while the device is online, it can make sure that there is a sufficient supply of one-time keys available, and that the fallback keys get replaced if they have been used. Additionally, the client devices should also not store the private half of fallback keys indefinitely to avoid situations where attackers can decrypt past messages sent using that fallback key. Instead, they should keep the private keys for at most 2 fallback keys: the current, unused, fallback key and the key immediately preceding it. Once the client is

reasonably certain it has received all messages that used the old fallback key, such as after an hour since the first message, it should remove that fallback key.

Secondly, clients should keep track of the ratchet indices of Megolm sessions, rejecting messages with a ratchet index that they have already decrypted. Otherwise, an attacker can try to disrupt the consistency of the room event history or impersonate another client device, among others.

Impersonation An attacker could try to send a message claiming to be from the victim without the victim having sent the message to: impersonate the victim while performing illicit activity and/or obtain privileges of the victim. To this effect, the attacker can alter the contents of an existing message from the victim or try to send a new message purporting to be from the victim with a phoney "origin" field.

To prevent these types of attacks, specific properties are included in encrypted Olm Matrix events, to stop an attacker from publishing someone else's curve25519 keys as their own and subsequently claiming to have sent messages that they did not. The *sender* field must correspond to the user who sent the event, the *recipient* to the local user, and *recipient_keys* to the local ed25519 key. Clients must confirm that the *sender_key* and the ed25519 field value under the *keys* property of an encrypted event match the keys returned by the "/keys/query" endpoint for the given user, and must also verify the signature of the keys from the "/keys/query" response. Without this check, a client cannot be sure that the sender device owns the private part of the ed25519 key it claims to have in the Olm payload. This is crucial when the ed25519 key corresponds to a verified device.

MiTM - Man in The Middle Whenever two clients engage in offline communication, i.e., outside of Matrix, the attack surface varies slightly, given that the network and communication methods are also different from the first two cases. Namely, the two clients are susceptible to man-in-the-middle attacks, when they are searching for the other client's host in the private network, through mDNS. While the two clients are searching for each other, the attacker could try to impersonate one of them in the network and lead them to share sensitive information with him.

Both the libp2p functionality and the client logic have mechanisms to prevent this possibility. As described in Section 4.2, specifically in subsection 4.2.2.1, the client logic for offline communication as a whole deploys security mechanisms to protect against this sort of attack. By using secure and authenticated communication channels with their peers, clients communicating through the private network (i.e., not through Matrix) can be assured that they are conversing with their correct counterparts.

4.4 Summary

This section narrowed the intended use case and defined the architecture for the prototype, as well as defined the threat model for the proposed system, explaining the

countermeasures employed to resist such an adversary.

Vying away from the initially chosen protocol was a tough decision, but ultimately, the correct one, as it allowed us to explore a more open and modern option with the Matrix protocol. This way, the prototype will be much more adaptable to any future circumstances, as Matrix is considerably more flexible and malleable than the other considered alternatives.

The discussed architecture is a straightforward and compact IoT system. The communication methods described in this section are highly suitable for the context of this thesis and have valuable properties for the system, enabling the desired efficiency and availability. Additionally, the adversary model and its countermeasures make the proposed system a robust and promising solution to achieve the previously-determined goals.

In the next section, the implementation details and technologies used for the prototype will be presented and explained.

5

Implementation and Prototype

In this chapter, the developed prototype for the use case and architecture defined in the previous chapter (Chapter 4) will be presented. First, there will be a discussion regarding the prototype's high-level aspects, and the overall choices made before the actual implementation (section 5.1). Then, the main challenges that occurred throughout the process of implementation in Section 5.2 will be shown and explained. Finally, a summary of the implemented prototype and the overall solution (section 5.3) concludes the discussion of the prototype's implementation.

5.1 Prototype

The developed prototype enables communication in a decentralized environment, through the usage of the Matrix protocol, with additional support for offline communication. The basis for its implementation was derived from an existing Matrix client implementation, called *gomuks* [7], with the rest of its functionality developed from scratch.

5.1.1 Client SDK

Matrix offers plenty of fully-fledged SDKs[1] for client development, for many different programming languages, such as JavaScript, Rust, GO, Java, Dart, Kotlin, Python, and C#, and there even exists a client library for MicroPython [35], with the most popular ones being the Python and JavaScript libraries. However, this being inserted in an IoT context, a somewhat lightweight solution would always be a valuable ally for the future, regarding storage capacities, energy levels and computational effort.

With that in mind, the choice for the programming language to develop the prototype was Golang, more specifically, the *mautrix-go* framework [8]. Not only is GO one of the most prominent programming languages in recent times, making it suitable for any future work to build on this prototype, but it also has a large ecosystem of partners, communities, tools and extensive documentation to aid in its usage. Furthermore, GO has built-in

[1] SDK stands for Software Development Kit, which is usually a set of tools to help the developer create applications for that specific platform.

support for concurrency, which is essential for the offline communication section of the prototype's code.

5.1.2 Interface

The decision for the client's interface was a command-line interface (or CLI, in short form). Taking into account the context in which the client will be, an interface of this type is perfectly adequate, as the prototype will never really be made for commercial use, and therefore, the users will most likely be engineers used to this type of interface, where a terminal console is the only requirement. Moreover, a CLI is simple, both in its form and implementation, further advocating in favour of its choice.

5.1.3 Storage

As stated previously, due to the requirements of the environment where the clients are going to exist, the clients' storage should be lightweight, as the storage space of the device will be limited. As such, the implementation will only feature a small database, to store cryptographic material for encrypted communications, and another, similarly small key-value store, to store the exchanged data between clients.

Additionally, each chat room has a cache to speed up accessing information about its members, state and settings.

5.2 Implementation - Code

This section will present and discuss the overall code of the prototype, i.e., how it is organized, the main challenges that arose during its development, and the solutions that were employed to solve those problems, with thorough explanations through the means of pseudocode algorithms.

5.2.1 Code Organization - Modules

The prototype implementation is available for research purposes at the following GitHub repository (https://github.com/jfcSoares/matrix-thesis), and it consists of around 4000 lines of code.

The implementation is centred around a **ClientWrapper** struct, which essentially handles every action the client needs to perform. This object makes use of several modules, of different aspects of the implementation, to aid it in executing every command that is required of it. These modules include:

- a **Matrix client** object, which implements the Matrix API (and is part of the *mautrix-go* SDK), allowing the prototype to communicate with a Matrix server through all of the API's endpoints;

- a **Syncer** module, that implements a custom syncing mechanism with the server, as described in the SDK's documentation (see the Interface definition at [66]);
- a **HistoryManager** module, which is responsible for storing event history locally;
- a **Crypto** module, which also implements the Matrix API (and is part of the *mautrix-go* SDK), but specifically in regards to its cryptography principles, allowing the encryption and decryption of communications within Matrix;
- a **Config** module, short for "configuration", that creates and manages directories to store all relevant data after it is dealt with by the other modules;
- a **Logging** module, to format and organize the client's internal logging messages;

A helpful comparison for this process of modularization could be to focus on the main module, the **ClientWrapper**, as the brain of the entire application, using the other modules as its "limbs", to perform its desired commands. Above the **ClientWrapper** module is the **Thesgo** module, which will interact with and be called upon by the "outside world", namely, the command-line interface and its user. These modules and their interactions are visually represented in Fig.5.1, with the caveat of the interactions between the modules inside the **ClientWrapper** not being shown for visual clarity.

5.2.2 Challenges

Server Synchronization Synchronization with the server is a major aspect of client implementations and must be dealt with efficiently and carefully since it affects most of the client's functionality if not done properly.

Clients must periodically synchronize with the server, to be informed about every event that happened in the rooms in which they participate. This must be done concurrently with the application's main functionality, which is to answer and complete the user's commands. This means that synchronization with the server must not impact the rest of the application's functions, nor should it be impeded by them.

Local Storage Implementing storage components in IoT systems is always a challenge, considering the device's power and storage capacity constraints. Therefore, there was a necessity for a local storage component capable of aiding the application in serving data to the user, while also being lightweight and efficient in its functions.

In essence, the implementation's storage component can be simplified into two categories: the cache, and persistent storage.

For caching, it has to be carefully managed in regards to its size, given that IoT devices generally do not have much memory to use for this purpose. Furthermore, the contents of the cache also require some attention, and a choice needs to be made regarding what type of information is kept in memory, and for how long.

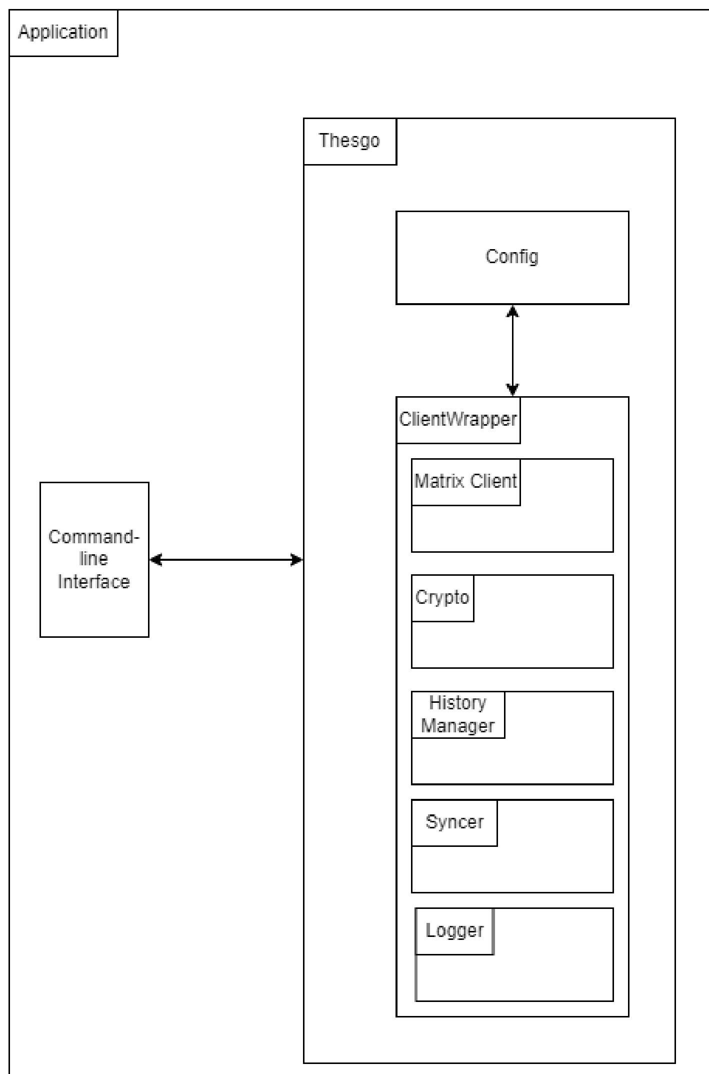


Figure 5.1: Component diagram of the application, showing the interaction between the different modules

For persistent storage, there is also limited available space on which to store data, and as such, it must be lightweight, either by storing less data or by compressing it before storage. Moreover, it should also be efficient in its operations, fetching and writing to storage quickly, to reduce general overhead on the system.

Encryption To implement encryption for the prototype's communications, the specification present in Subsection 3.2.1.4 was used.

Matrix's cryptography involves the use of the Olm and Megolm libraries for encryption. It was important to understand how these libraries work, including the creation and management of Olm and Megolm sessions, their purposes, and internal functions.

Aside from the protocol's specification, there is also some implementation-specific documentation regarding encryption, i.e., the Matrix E2EE implementation guide [28], facilitating the overcoming of this challenge. The implementation guide is considered as

"legacy documentation", since Matrix recently updated their domain and websites (as of the 13th of September, 2023), but since no updated version of this guide has been put out on their new domain, this guide was still the best tool for this purpose.

Interface After choosing a command-line interface for the application, a decision needed to be made regarding which commands to implement, and how. The commands that required implementation are essentially the basic functionality a Matrix client can provide: sending and receiving messages, joining or leaving a room, creating a room, checking the members of a room, checking the event history of a room, logging in and logging out, as well as showing some general user account information if requested.

Regarding how they would be implemented, it effectively came down to which Go packages to use for that purpose. Command-line interfaces are fairly common, even more so with Go, which offers a variety of packages and methods to implement a custom command-line interface. Although manually implementing it for full customizability is feasible, it would likely be a much more lengthy process. Furthermore, even with a package chosen, there was still an additional problem to address: to allow the user to continuously execute commands after the previous one has finished its execution, without terminating the user's session. Due to the nature of Matrix's specification and operation, the user's sessions needed not to be terminated after executing a single command, not only in regards to efficiency and computational concerns, because this would mean that user and cryptographic data would have to be loaded every time the user executed a new command, but also in respect to the overall execution flow of the application, making it awkward for the user to not be fully certain that they are logged in.

Communicating Offline Implementing offline communication meant asking, and answering several different questions:

- How to detect if the client is offline?
- How to search for the client if he is indeed offline? What technologies or libraries to use for this purpose?
- Once the offline client has been discovered, how to authenticate it, and establish a secure communication channel with it, outside of Matrix?
- How to know what events the offline client has not received?

Answering all of these questions and controlling the variables involved was a challenging and strenuous task. There are numerous methods available for detecting if a client is offline, such as pinging the host. for instance. Additionally, there are several ways of searching for the offline client, including Bluetooth, WiFiDirect, or P2P networking. Finally, there are also multiple ways of verifying that the host that the client is connected to is indeed the offline client and that communication with it is secure.

5.2.3 Solutions

Synchronization with server The solution for this issue was surprisingly easy, and it can be grouped into two aspects: the first one was already mentioned in Subsection 5.2.1, which is the implementation of a custom **Syncer** module; the second one, aided by GO's simplicity, was the use of one of GO's main properties - built-in concurrency.

The Matrix API allows developers to implement their proprietary syncing mechanism, by implementing the Syncer interface (seen in Alg.9), which contains the *ProcessResponse*, *OnFailedSync*, and *GetFilterJSON* functions, through which the developers can choose how their clients process the server's responses to the Sync() endpoint, what actions should be performed when a synchronization call fails, and what types of events their clients are interested in (i.e., filtering the response), respectively. This is exactly what the **ThesgoSyncer** type does in the prototype's code, implemented in the mentioned Syncer module.

Algorithm 9 Functions for the *mautrix-go* Syncer Interface

```
1: // Processes the /sync response. "since" is useful to detect the very first sync
2: ProcessResponse (response, since string) error
3: ▹ OnFailedSync returns either the time to wait before retrying or an error to stop sync
   permanently
4: OnFailedSync (response, error) (time, error)
5: GetFilterJSON(userID) *Filter                    ▹ Returns the filter for a given userID
```

In regards to concurrency, the GO runtime supports the use of "lightweight threads", called *goroutines* [38]. These threads make features like server synchronization extremely easy to implement, given that one can just create a new goroutine at any point in time, and have it perform any desired operation.

Additionally, to use alongside goroutines, GO also employs a specific data type, called a *channel*. Channels are a typed conduit through which you can send and receive values with the channel operator, <-. By default, send and receive operations with channels block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

Finally, GO's built-in concurrency also has a specific control statement, the *select* statement, which allows a goroutine to wait on multiple communication operations, blocking until one of its cases can run, and then executing that case. if multiple are ready, one is chosen at random.

With these features, the synchronization code for the prototype is materialized in two GO functions, **Start** and **Stop**, shown by Algorithms10 and 11.

Once a user has logged into the client, and its data has been fetched from local storage (if present), a new goroutine starts, calling the **Start** function, alongside the main routine (the main thread), initializing the synchronization process by calling the Sync() endpoint of the Matrix API. This thread is effectively controlled by the *c.stop* channel variable; whenever the channel holds a "true" boolean value, the thread will stop its current execution and

Algorithm 10 Function called by the server synchronization goroutine - initializing the syncing process

```
 1: function START
 2:     CRYPTOONLOGIN                          ▷ Get crypto store from files if it exists
 3:     ONLOGIN                                ▷ Initialize the syncer
 4:     if client == nil then
 5:         return
 6:     end if
 7:     running = true
 8:     client.SyncMinAge = 30 * minute
 9:     while true do
10:         select:
11:         case ← stop:
12:             running = false
13:             return
14:         default:
15:             error = client.SYNC
16:             if error ≠ nil then
17:                 if error == UnknownToken then
18:                     LOGOUT
19:                 else
20:                     PRINT("Sync call error")
21:                 end if
22:             else
23:                 PRINT("Sync call successful")
24:             end if
25:         end select
26:     end while
27: end function
```

terminate itself, i.e., the synchronization thread runs infinitely until the user logs out of the client and the "stop" signal is sent to the channel.

This process is encapsulated in a *select* statement in both the **Start** and **Stop** functions, to cover the two possible cases for the value that the channel can hold. In the first function, a call is made to the Sync() endpoint every 30 minutes, as stipulated by the *StreamSyncMinAge* variable, as long as the channel holds a "false" value; otherwise, when that value is "true", it stops the execution and immediately returns from the function, stopping the sync process. In the **Stop** function, if the client is indeed active, it waits infinitely for the channel to hold a *true* value, at which point it explicitly stops the synchronization mechanism, and terminates the execution of all of the client's modules, effectively closing the user's session.

To accommodate for this syncing routine, and the implementation of the **ThesgoSyncer** module, event handlers need to be implemented and assigned for each specific type of event. This was done through the **ClientWrapper** module, assigning the event handlers to their respective event types whenever a user logs in, as seen in Alg.12, which shows pseudocode for the implementation of the OnLogin() function. The custom syncer is

Algorithm 11 Function called by the server synchronization goroutine - terminates the syncing process and the client's activity

```
1: function Stop
2:     if running == true then
3:         select
4:         case stop ← true :
5:         default:
6:         end select
7:
8:         client.StopSync
9:         error = history.Close
10:        if error ≠ nil then
11:            Print("Error closing history manager")
12:        end if
13:
14:        history = nil
15:        if crypto ≠ nil then
16:            error = crypto.FlushStore
17:            if error ≠ nil then
18:                Print("Error flushing crypto store")
19:            end if
20:        end if
21:    end if
22: end function
```

Algorithm 12 Pseudo-code for the OnLogin() function, where handlers are assigned to incoming event types

```
1: function OnLogin
2:     cryptoOnLogin
3:     client.Store = Config
4:     Syncer = NewThesgoSyncer(Config.Rooms)
5:     if Crypto ≠ nil then
6:         Syncer.OnSync(Crypto.ProcessSyncResponse)
7:         Syncer.OnEventType(StateMember, CryptoEventHandler)
8:         Syncer.OnEventType(Encrypted, HandleEncrypted)
9:     else
10:        Syncer.OnEventType(Encrypted, HandleEncryptionUnsupported)
11:    end if
12:    Syncer.OnEventType(Message, HandleMessage)
13:    ...            ▷ Lots of different event handler assignments omitted for visual clarity
14:    Syncer.OnEventType(StateMember, HandleMembership)
15:    Syncer.OnEventType(EphemeralReceipt, HandleReadReceipt)
16: end function
```

initialized and then specific event types are assigned to their corresponding handlers. For example, regular room messages are directed to the HandleMessage() function while membership events are sent to the HandleMembership() function, and so on. Now that the client could effectively receive events from the server, the next challenge was storing those same events in the device on which the client is running, which will be explained in the next section.

Storage For both categories, their implementation is intrinsically connected to the GO packages most suited for that purpose. Here, GO offers numerous packages with different implementations for various use cases, but the ones deemed the most adequate for the application's needs were: the bbolt package [50], for a lightweight and simple key-value store; the go-deadlock package [80], to avoid deadlocks when accessing data; and the encoding package [37], one of the default packages of GO, to handle data as efficiently as possible when reading and writing from storage.

The last two packages mentioned are extensively used in this part of the code, as shown by the pseudocode present in Alg.13. This snippet shows the Load() function of a *room*, where, after some initial safety checks, a lock is acquired, to then call the private function load(), which will load into memory the saved state, using the "encoding/gob" package to decode a gzip-compressed file [39]. This flow of acquiring the lock before executing any read or write operations in the cache is ever present in this section of the code.

In this context, **caching** is only relevant when referring to data within a room (except for message events, which are stored differently, as will be explained at the end of this paragraph), and, as such, each *room* type (as defined in GO) stores information as follows:

- Each *room* contains a *RoomCache* struct, that maintains room state info in a hash map and a linked list . This type of object essentially keeps each room's state information in persistent storage and memory, acting as the state store [2] for the Matrix client;
- Additionally, each room also has its own state and member cache, modelled as a "map" data structure stored in memory.;

This flow is essentially the traditional approach to storage, with some small nuances, where room data is first loaded from storage through the *RoomCache* object (if it is present), and later, as more and more room data arrives and is processed, it is put into memory, i.e., into the state and member map data structures. Furthermore, the data in the cache is periodically persisted in storage through an independent goroutine for that specific purpose.

In summary, the room list and room states are managed and stored as explained above, with the aid of caching and periodically persisting the cached data. Other types of data, however, such as Matrix events, and message events, among others, are handled differently once the application finishes processing them.

[2]"store" in this context refers to storage.

Algorithm 13 Pseudo-code exemplifying the use of the *go-deadlock* and *encoder* packages

```
1:  function Load
2:      cache.TouchNode(room)
3:      if room.Loaded then
4:          return
5:      end if
6:      if room.preLoad ≠ nil && !room.preLoad then
7:          return
8:      end if
9:      room.Lock                                  ▹ Provided by the go-deadlock package
10:     room.load
11:     room.Unlock                                ▹ Provided by the go-deadlock package
12:     if room.postLoad ≠ nil then
13:         room.postLoad
14:     end if
15: end function
16: function load
17:     if room.Loaded then
18:         return
19:     end if
20:     file = os.OpenFile(roomPath, os.RDONLY)
21:     ...                                        ▹ Error handling was omitted
22:     cmpReader = gzip.NewReader(file)           ▹ Provided by the gzip package
23:     dec = gob.NewDecoder(cmpReader)            ▹ Provided by the encoding package
24:     room.changed = false
25: end function
```

The processed data is stored in a key-value store provided by the bbolt package and managed by the **HistoryManager** module (referenced in Subsubsection 5.2.1). Similarly to the previous case, here, the go-deadlock and encoding packages are crucial to guarantee the well-functioning of the database and improve the overall efficiency of the code.

Algorithm 14 Pseudo-code for the Load() function of the **HistoryManager** module

```
function Load(room, num, ptrStart)
    events = Event[]
    Lock
    error = DB.View(function(boltTransaction)
    i = 0
    j = len(events) − 1
    while i ≤ j do
        events[i], events[j] = events[j], events[i]
        i = i + 1
        j = j − 1
    end while
    return
end function
```

In Alg.14, an example of the use of these packages and the **HistoryManager** module

is demonstrated. In this function, first, a lock is acquired, to protect all of the operations inside its body, and then, a call is made to the View() function of the database, provided by the *bbolt* package, which executes a function within the context of a managed read-only transaction. This process, if no errors are thrown, will populate the *events* array (or slice, as it is called in GO documentation), and after some post-processing, returns the calculated results.

Encryption Implementing end-to-end encryption in the application was surprisingly easy since most of the low-level details of the specification (creating Olm sessions, sharing Megolm session keys, etc.) are abstracted by *mautrix-go*'s cryptographic module, and are done automatically, "under-the-hood". The encryption used in the application is grouped under the "Crypto" module, which is mentioned in Subsection 5.2.1.

As such, the main decision to make regarding encryption was which database to use for storing cryptographic data. In this context, cryptographic data includes device lists for all known users, device keys for those users and devices, as well as Megolm and Olm session storage with information about them. Additionally, it includes the trust level for each device, among other details. To this end, an SQLite [18] database was chosen, using the *go-sqlite3* package [65], which is the database usually used alongside *mautrix-go*'s cryptographic module.

In Algorithm.15, the pseudo-code for the initCrypto() function is shown, where the database is created (and its tables are generated), and assigned as the *cryptoStore* for the Crypto module. Data stored in this database is encrypted by the given "pickle key" (the last argument in the call to the NewSQLCryptoStore() function).

In this function, the Crypto module, of the type **OlmMachine**, as the *mautrix-go* SDK named it, is also initialized, creating or loading an Olm account from storage to later use for encrypted communications.

When the initCrypto() function finishes its execution, the client can now send and receive encrypted communications. This function is called when the application is initialized.

For the client to receive encrypted messages, a specific handler needs to be implemented and assigned since encrypted messages in Matrix have their own event type, which is *m.room.encrypted*. This assignment can be seen in Alg.12, right after the confirmation that the **Crypto** module has been initialized.

In Alg.16, the pseudo-code for the event handler for encrypted room events is presented. Here, the client first attempts to decrypt the given event, by calling the Decrypt-MegolmEvent() function of the *crypto* module. This function abstracts all of its underlying details, but it essentially performs the following actions:

- It checks the local *cryptoStore* for a group session (the same as a Megolm session) with the same session ID as the one with which the event was encrypted - if one

Algorithm 15 Pseudo-code for the initCrypto() function of the Crypto module

```
function initCrypto
Ensure: error
    newStorePath = file.Join(DataDir, "crypto.db)
    db, err = NewDB(newStorePath, "sqlite3")
    if err ≠ nil then
        return err
    end if
    log = Logger
    accID = userID + deviceID
    cryptoStore = NewSQLCryptoStore(db, log, accID, deviceID, pickleKey)
    err = cryptoStore.DB.Upgrade
    if err ≠ nil then
        return err
    end if
    crypt = crypto.NewOlmMachine(client, log, cryptoStore, config.rooms)
    Crypto = crypt                          ▷ Crypto module is effectively initialized
    Crypto.Load
    if err ≠ nil then
        Print("Failed to create Olm machine")
        return err
    end if
    return nil
end function
```

Algorithm 16 Pseudo-code of the event handler for encrypted room events

```
function HandleEncrypted(source, mxEvent)
    evt, err = Crypto.DecryptMegolmEvent(context, mxEvent)
    if err ≠ nil then
        mxEvent.Type = BadEncrypt
        origContent = mxEvent.ParseContent
        mxEvent.Content = BadEncryptContent(origContent)
        HandleMessage(source, mxEvent)
        return
    end if
    if type == InRoomVerification then
        err = Crypto.ProcessVerificationInRoom
        if err ≠ nil then
            Print("Failed to process verification event)
        else
            Print("Processed verification event successfully")
        end if
    else
        HandleMessage(source, evt)
    end if
end function
```

is found, the execution continues; if not, an error is thrown in accordance with its cause;

- After obtaining the necessary Megolm session, the ciphertext is decrypted, returning the plaintext and a message index; Similar to the first case, if the decryption is successful, the execution proceeds as expected, otherwise, an error is thrown;

- Lastly, the returned message index is used to perform some safety and consistency checks, i.e., update the ratchet indices, make sure the session message limit has not been reached, etc., and store the updated information back in the database;

- Once the plaintext is returned, additional security checks are made, investigating the trust level for the device that sent the message in question. If the device is considered to be trusted, only then is the plaintext handled, confirming that its parameters are correct (if the room where it was sent exists, if it has a valid event type, etc.), and finally returning the original decrypted event.

If the decryption is successful, the decrypted event is sent to the regular event message handler. However, there is an exception to this case, where the decrypted event might pertain to an in-room device verification, in this case, a specific handler function is called, namely, the **crypto.ProcessInRoomVerification()** function.

In the opposite scenario of sending an encrypted message, as shown in Alg.17, the execution flow is comparable, where first, a check is made to the room's parameters, i.e., if it exists, if encryption is enabled inside that room, if the *crypto* module has been initialized, and if the event is not a "reaction", a specific event type, which must be dealt with differently. After the room where the event is to be sent is confirmed to be valid, the event is encrypted, by calling the EncryptMegolmEvent() function. Inside this function, the *crypto* module searches for a group session for the given room ID, and if found, packages the event into a plaintext object, with the necessary format, encrypting said plaintext with the session key obtained previously.

Afterwards, the group session is updated in the *cryptoStore*, or more specifically, in its database, and the encrypted event is returned. If an error is returned instead, the client checks the error's cause: if the error is not related in any way to the group session, either because it does not exist, or it was not shared, etc., then the SendEvent() function will return an empty string and the error; otherwise, if the cause is indeed related to the group session, then the client will try to re-share the session with all of the room's participants and encrypt the event once more. Finally, the encrypted event is sent through the Matrix API endpoint "SendMessageEvent".

Finally, it is also worth mentioning that every room in this implementation has encryption enabled by default, but for safety, checks are always made to confirm the room's parameters, before sending a message, to avoid any inconsistencies in the rooms' event history.

Algorithm 17 Pseudo-code for the SendEvent() function, where event encryption occurs

```
function SendEvent(event)
Ensure: eventID, error
    room = GetRoom(event.RoomID)
    if room ≠ nil && room.encrypted = true && Crypto ≠ nil then
        encrypted, err = Crypto.EncryptMegolm(ctx, roomID, type, content)
        if err ≠ nil then
            if BadEncrypt(err) then
                return "", err
            end if
            Print("Failed while trying to encrypt, sharing group session again...")
            err = Crypto.ShareGroupSession(ctx, roomID, roomMemberList)
            if err ≠ nil then
                return "", err
            end if
            encrypted, err = Crypto.EncryptMegolm(ctx, roomID, type, content)
            if err ≠ nil then
                return "", err
            end if
        end if
        Type = EncryptedEvent
        Content = EncryptedContent
    end if
    resp, err = client.SendMessageEvent(RoomID, Type, Content, TransacID)
    if err ≠ nil then
        return "", err
    end if
    return resp.eventID, nil
end function
```

One additional aspect of the application's security will be discussed in the next paragraph, related to the users' device verification, which was implemented as an individual command.

As such, the next paragraph will focus on the command-line interface implementation and which commands the users can execute.

Interface Implementation To solve the aforementioned challenges regarding the application's interface, a combination of two Go packages was used, which made it fairly easy to implement:

- the famous *cobra-cli* Go package [33] was used to implement this client's interface. Cobra is a Golang library for creating powerful modern CLI applications, with Kubernetes or the GitHub CLI as examples of production-ready applications made using *cobra-cli*;

- the *cobra-shell* package [83], to implement the command-line interface's shell[3];

To maintain project modularity, and as is allowed by Cobra, commands were grouped into distinct command subgroups:

- The *user* command group, where there is every command that a user might require regarding its data, such as the **login**, **logout** and **accountInfo** commands;
- The *room* command group, where there are commands for every expected action regarding a Matrix room. This group includes commands like the creation of a new room, sending a message in a given room, verifying another user's device inside the context of a room, and leaving a room, among others;
- Above both of these, since Cobra is essentially organized as a tree, is the *root* command, which just displays information about the two command subgroups and the application's functionality. The *root* command also initialises the interface's shell.

Algorithm 18 Pseudo-code template for command creation with Cobra

```
Command = cobra.Command(){
    Use        ▷ name of the command
    Short      ▷ A short description of the command
    Long       ▷ A long description of the command
    Run        ▷ A function that contains code to execute the intended action of a command
}
```

With Cobra, the definition of commands is extremely simple, consisting only of a variable definition and the implementation of an init() function, where any desired flags can be declared and configured. The general definition of a command can be seen in Alg.18. The *Command* type has several attributes, among which are the following: The "Use" attribute, which is the name of the command, the "Short" and "Long" attributes, which are the short and long explanations of the command, respectively, and the "Run" function, which should be implemented if the command has an action associated with it.

This way, every command is standardized, facilitating code readability and efficiency. Once a command is defined, it is initialized through the init() function. This function defines any desired flags and configuration settings for the command. The initialization for a command is present in Alg.19, where both available types of flags are set and configured: a local *boolean* flag, named "toggle", that can only be called alongside this specific command; and two distinct *boolean* persistent flags, named "clear-cache" and "clear-data", which can be used alongside this command, and any of its sub-commands.

The organization and overall simplicity of Cobra CLIs not only make it easier to develop them but also add more commands as necessary, while significantly accelerating the development process.

[3]A **shell** in this context refers to an interactive shell, where the user's session remains open until he logs out or the terminal is closed, allowing the continuous execution of commands.

Algorithm 19 Pseudo-code for a command's initialization function

```
function INIT
    // Flags that are passed down to the command's child commands
    PersistentFlags.BoolVarP(var, name, shortform, default, description)

    // Flags that are only related to the command defining them
    Flags.BoolVarP(flagVar, name, shortform, default, description)

    noFlag = Flags.MarkFlagRequired(flag)
    if noFlag then
        // Do Something, show "Help" for instance
    end if
end function
```

Connecting to the back-end Although the definition of the commands was easy, the commands themselves could not yet perform any action, since there was no connection to the application's other modules. A new variable type called "Thesgo" was created to group the application's modules and mask the application's functionality. The definition of said type is shown in Alg.20, where the defined interface has four different functions: two getter functions, that return the Matrix **ClientWrapper** and the **Config** objects, respectively, and two regular functions, that start or stop the client's execution when called.

Algorithm 20 Type Thesgo

```
type Thesgo {
    function Matrix
        return MatrixContainer
    end function

    function Config
        return Config
    end function

    function Start
    end function

    function Stop(save boolean)
    end function
}
```

This way, any object that implements this interface will encapsulate all of the application's functionality in itself, making it easier to use for the user interface.

To this end, as can be seen in Alg.21, the interface's *root* command has a function that takes a variable of the aforementioned type as an argument and passes it through to both its sub-command groups, with each one defining a variable to hold the passed argument, finally connecting the interface with the application.

Algorithm 21 Pseudo-code for the implementation of the *root* command

```
function addSubCommandGroups
    AddCommand(shell)                    ▷ Adds an interactive shell
    AddCommand(userCMD)                  ▷ Adds the user commands as a subgroup
    AddCommand(roomCMD)                  ▷ Adds the room commands as a subgroup
end function

// Sets a variable in each command package (subgroup) pointing to the main application
object (Thesgo)
function SetLinkToBackEnd(Thesgo)
    user.SetLinkToBackend(Thesgo)
    rooms.SetLinkToBackend(Thesgo)
end function

function init
    Flags.BoolP(toggle, "t", false, "Help for toggle")
    addSubCommandGroups
end function
```

Additionally, it calls the private function addSubCommandGroups() upon initialization to add both sub-command groups and initialize the interactive shell.

With a fully-fledged CLI now implemented, the next and final challenge of the implementation was enabling the clients to communicate through other means, when they cannot communicate regularly through Matrix, i.e., through *offline* communication.

Offline Communication Offline communication between clients is controlled by the messages' read receipts (acknowledgements that a certain user, and by extension, the client, has seen the message). Part of the server's response to the /sync endpoint is a list of read receipts ordered by the event ID to which they are related. By checking the list of users that acknowledged each event, and cross-checking it with the list of members of the room where the event was sent, an inference can be made about which users are offline and require the event through different means. Given that clients automatically send the read receipt for an event as soon as they receive it, an assumption can be made such that, if a user did not issue a read receipt, it is because it is offline and did not receive the message.

Therefore, in the event handler for read receipts, whenever a user does not acknowledge a certain event, data is sent to a separate thread, identifying the user that did not acknowledge the event, and the event itself. This is represented by the pseudo-code in Algorithm 22. When this separate thread receives the information in question, it will try to connect to the offline client and initiate the execution flow for offline communication.

The execution follows the protocol shown in Fig.5.2, where first, the client's libp2p host uses the mDNS service discovery to find the offline client's host in the private network where they reside. After receiving a positive reply to the mDNS query from the offline

Algorithm 22 A breakdown of the steps involved in parsing and handling event read receipts

```
function parseReadReceipt(room, evt)
    members = JoinedMembers(evt.roomID)
    for all eventID, receipt in evt.Content do
        ...                                          ▷ Logic related to room timestamps

        actualEvent = GetEvent(room, eventID)

        // Only send events that the client sent originally
        if actualEvent.Sender == client.UserID then
            ackUsers = receipts["read"]              ▷ Get all users that ACK'ed the event
            for all user in members do               ▷ Compare them against room members
                if ackUsers[user] = nil then
                    offline.users = append(offline.users, user)
                end if
            end for

            // If at least one user did not send a receipt for this event
            if len(offline.users) > 0 then
                offline.eventID = eventID
                offline.roomID = evt.RoomID
                sendOff ← offline                    ▷ Send data to goroutine
            end if
        end if
    end for
end function
```

host[4], the online client's host opens a connection to it, performing a TLS handshake to secure and authenticate the connection. Once a secure communication channel has been established, a data stream is opened in that connection, where the two hosts will exchange the given event.

Since the user's Matrix identity key can not be used to identify its libp2p host, given that its private key is not exportable, additional authentication regarding Matrix is required of the two hosts. As such, they then exchange Matrix credentials, confirming that the offline host's Matrix user is indeed the one in need of the event, initiating the event encryption process.

A first attempt is made, encrypting the event as in a standard scenario, and sending it to the data stream. If the offline host can decrypt the event, it returns an ACK, and the connection is terminated; otherwise, it asks for the Megolm session key for the encrypted event, at which point the online host will share that key, and the offline host will be able to decrypt the event and terminate the connection.

[4]In this paragraph, an "offline" user or host means that it is disconnected exclusively from the Internet and, therefore, Matrix, and not any other network.

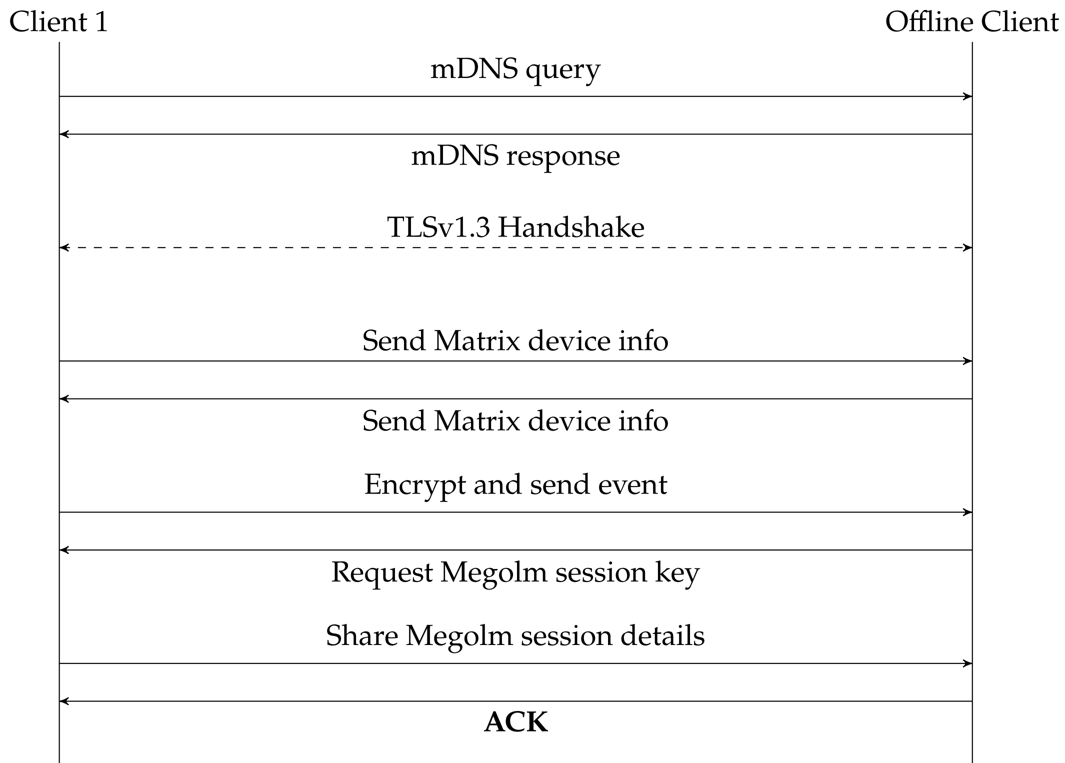


Figure 5.2: Protocol developed for client offline communication for the case where the offline client does not have the Megolm session key in storage

5.3 Summary

In this chapter, a detailed description of the implementation of the developed prototype was given, which is intended as a solution for decentralized communication, with support for an offline mode of operation. This prototype's objective, after testing, is to confirm that any system using this prototype can securely communicate data to its counterparts, regardless of internet connection status, maintaining overall consistency among all of the system's participants' data.

The Matrix protocol was successfully employed in this thesis' use case, further advocating in favour of the protocol's claim that it can be used to develop IoT systems' communication. To support the usage of the protocol in this implementation, several GO packages were used, as well as some internal GO features, that significantly aided the design and overall solutions for the presented implementation challenges.

To implement server synchronization, heavy usage was made of Go's built-in concurrency to create a separate execution thread, or goroutine, to periodically synchronize with the server. This goroutine is launched whenever a client logs in to the device, or the device starts up (in case a session was stored). This way, the client can execute any commands that the user may desire, without explicitly stopping the application's functionality to synchronize with the server.

The implementation of storage capabilities and encryption was largely based on the packages used for those purposes: the *bbolt* package, the *go-deadlock* package, and the *encoding* package, for storage; and the *mautrix-go*'s cryptographic module for encryption, with the aid of the *go-sqlite3* package to store any needed cryptography-related material. These packages fulfil the requirements for these modules, as they allow for lightweight and efficient storage for events and cryptographic material, with a manageable cache for room-related data.

Similarly, the interface implementation was also heavily influenced by the Go packages chosen. By offering a standardized library for the implementation of command-line interfaces, the *cobra-cli* Go package, alongside the *cobra-shell* package, made for a simple, but effective user interface for the developed application, allowing both the execution of isolated commands, and the continuous execution of multiple commands in a shell.

Finally, the biggest challenge of the implementation, and perhaps its main feature, was enabling offline communication. Most of the questions posed in the Communicating Offline paragraph were answered by the usage of the *libp2p* package: by using the general capabilities of the library, the host could be initialized, have it search for other hosts through mDNS and establish secure and authenticated communication channels with it. To know if a certain client is offline, Matrix's read receipts were used. If a given user, and by extension, the client, did not acknowledge a certain event, that host is assumed to be offline and that it will require that event through other means. This was the most effective way of implementing this functionality since it informs a client of all of the other client's connection status, and what events they are missing. Furthermore, this all happens in parallel to the client's regular execution flow, similar to the server synchronization implementation, where a separate thread is assigned to perform these functions.

The prototype implementation acts as a proof-of-concept for the system the proposed use case envisions, enabling secure communication in a decentralized setting, with added support for offline operation, increasing the system's general data consistency among all its members.

6 Experimental Evaluation

In this section, the testing and experimental evaluation process for the system, and the prototype, will be presented. First, a description of the testing environment (section 6.1) is given, followed by the methodology used for those test cases, i.e., the sequence of executed commands, presented and discussed in Section 6.2. Afterwards, a thorough and precise definition of the test cases that are being emulated (section 6.3). Finally, the results are presented and discussed in Section 6.4, finishing the chapter by summarizing all of the aspects of the experimental evaluation, and comparing it to the desired objective (section 6.5).

6.1 Testing environment

Due to practical limitations, testing the described use case "as-is" (see Section 4.1) was not feasible. This is because this would require several server deployments (one for each group of devices), with multiple client devices per group, which would require significant computational resources, as well as time, to set up the entire system and its components. So, the decision was to test a single group of devices (with **two** client devices) connected to a server, since the application's functionality is not affected whether there are multiple device groups or not (messages are sent and received the same way).

To this end, a system functioning on a private network was required, to limit foreign traffic from traversing the network, as well as a method to limit the client's communication with the outside world, i.e., the Internet, to test offline communication. The solution for this was a containerized deployment with Docker, to allow the creation of a private network for the containers and easily block a specific container's communication with Matrix.

The system deployment was tested on a Windows 10 host (Version 10.0.19045 Build 19045), with Docker Desktop [45] (version 4.23.0) running Docker engine (v24.0.6) with a WSL2 backend.

6.1.1 Server

The server component was set up with two Docker Compose(v.3.3) files. As described in Subsection 4.2.1, one of them defines and configures the reverse proxy and its certificate manager, and the other configures the server and its database. The four containers are then connected to and communicate with each other through a private Docker network, also defined in those same Docker Compose files. The result of this server configuration is visible in Fig.4.2.

For security reasons, the server container does not allow any external access through its ports. This is per the recommendations of Matrix Synapse's developers. Instead, the reverse proxy is responsible for publishing ports 443 and 80 for HTTPS and HTTP requests, respectively. These ports are used by clients to communicate with the server, as the traffic is redirected through them.

To emulate the real use case as closely as possible, i.e., with multiple servers with Federation (server-server communication) enabled, a public DNS domain was required for the server. As such, one was registered at DuckDns.org (a free dynamic DNS provider hosted on AWS) with the external IP address of the home network on which the system was tested. This will impact the communication flow for the test cases that will be explained ahead, in Section 6.3.

6.1.2 Clients

For the client component of the system, a Docker image was created from an Ubuntu operating system, which then pulls the prototype's source code from GitHub, with the application executable already compiled and ready to run.

The client containers are inserted into the same Docker network where the other containers reside, following the proposed use case. They are also run in interactive mode so that commands can be executed as the user wants, in the order that he wants, instead of automatically or through a script.

6.1.3 Network

To better control each container's available network interfaces and how the system's components communicate, the containers were made to not connect to the default Docker bridge network, *docker0*, and instead, only connect to the private network created for that specific purpose.

This private network is also a bridge network, which is an isolated network that runs on a single Docker Engine installation.

6.2 Methodology

To establish a common basis between all of the test cases, a simple sequence of commands was devised, taking into account both the need to test for the offline capacities, as well as the normal functionalities of the application.

1. Create a room containing only the two users logged in to the two containers;
2. Check the room's members once they are both in the room (to test general functionality);
3. Send a message in that room;
4. Check the room's history, confirming that the expected message has been delivered;

Every test case will follow this exact pattern, with the caveat that, for the tests regarding offline communication, one of the clients will not see the message sent in Item **3** immediately. Instead, it will wait for the other client to send that event outside of Matrix, only calling the history command afterwards (because first, the command attempts to retrieve events from storage; if retrieval fails, it requests them from the server).

This way, a common ground is defined for all tests, acting as a well-defined starting point for the test cases.

It is also worth mentioning that the encryption parameters for Megolm sessions, i.e., the number of messages and/or the amount of time until a session is replaced, were not changed. The default and recommended parameters were used for the room where the test cases were performed, meaning that the tests did not employ the maximum security possible. To test with the best possible security Matrix can offer, sessions would have to be replaced after each message, to guarantee backward secrecy; this not only is impractical in relation to the testing process but also somewhat debatable, since a real-world solution may want to configure these parameters at their own discretion. Similarly, to improve Matrix's forward secrecy guarantees, there should be an option to discard historical conversations, by winding forward any stored ratchet values or discarding sessions altogether. This feature was not implemented originally, but a patch to the original code could easily be made to provide this option to the user.

6.3 Test Cases

To test the prototype's functionalities in various network conditions, three different test cases were designed for the system:

1. A regular execution, where the network is stable, and clients can easily communicate through Matrix;

```
> thesgo room -n "!QptuTqzPTrcrWXxxMv:lpgains.duckdns.org" members
@test2:lpgains.duckdns.org
@test1:lpgains.duckdns.org
> thesgo
         exit   Exit the interactive shell.
         help   Help about any command
         room   Commands for every expected action regarding rooms.
         user   User is a command group for user-related commands
```

Figure 6.1: Example of the execution of a command in the client containers, with the interface shell also displayed

2. A case where one client is stable and able to communicate with the server through Matrix, and another client is offline, where it should be possible for them to communicate through other means, aside from Matrix;
3. A final case, where both clients are unstable, and often disconnect and connect to Matrix;

The first case intends to test the solution's suitability to a regular network context, where everything is functioning as intended, and clients communicate normally through Matrix. The second test case intends to specifically test the offline communication capabilities of the prototype, by impeding one of the clients from communicating with the Matrix server. The third and last test's purpose is the emulation of a situation fairly common in IoT systems, where a device's internet connection is severely unstable, and the device is continuously alternating between an online and offline status.

These test cases, although not implemented in a real-world use case, should cover most, if not all, of the functionalities and contexts proposed for the prototype.

6.3.1 Test Case: Regular Communication

This test case is the easiest of the three to test, given that it follows the normal pattern of execution. The test's execution follows the schema defined in Fig.6.2, where the two clients execute commands in the container's terminal, which then calls the application's back end to perform the necessary HTTP requests to the server, which flow through the created Docker network. An example of this is present in Fig.6.1, where the *members* command was called for a room with ID "!QptuTqzPTrcrWXxxMv", and the output is shown below the command. Additionally, the figure also shows the interactive shell that was developed, with support for command auto-completion.

6.3.2 Test Case: One Offline Client

This test case was slightly more difficult to test, given that there were multiple ways to emulate one of the clients going offline. To this end, a few approaches were tested:

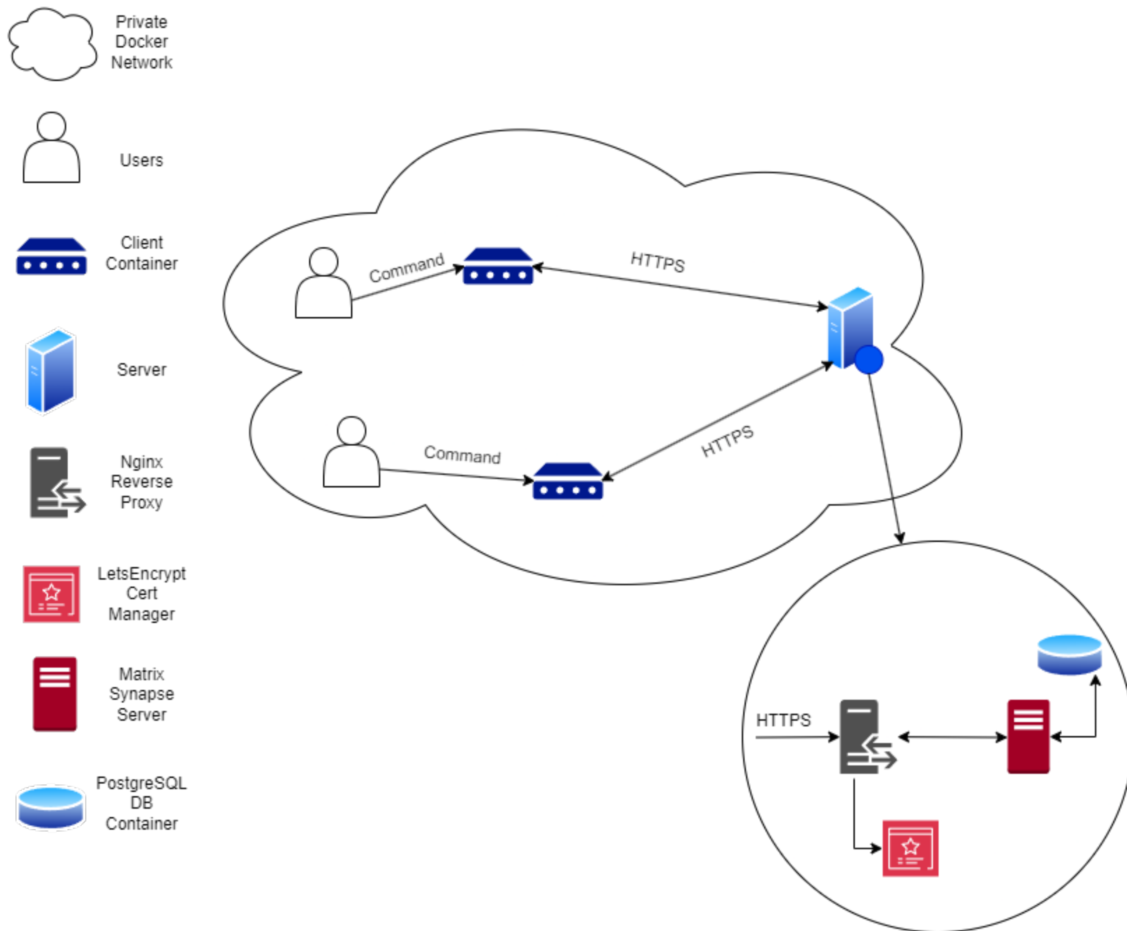


Figure 6.2: Diagram visualizing the tests for the regular communication pattern of the prototype

- Creating a separate, private and externally isolated Docker network containing only the two client containers. This way, to emulate one of them being offline, the client would be disconnected from the other network, where the other client and the server reside, but remain connected to this new externally isolated network to only be able to communicate with the other client;
- Defining some internal iptables [77] rules in one of the client containers, to block traffic to the server;
- Keeping both client containers in the same private Docker network as the server, but only keeping one user fully logged in. The other user in the other container would log in, but close the session, maintaining its information in storage. This way, it is able to re-initialize the client, keep the shell alive, and receive data while technically being offline.

The first approach, although it was the one most truthful to the context it intended to

test, ended up not working as expected. The client's libp2p host was not able to recognize the externally isolated network's interface in order to use it for communication, which, in turn, made this test case's methodology ineffectual.

The second approach, which showed similar promise to the first one, also was not able to produce any results. Even though the *iptables* Linux package exists, and is available for use, it did not function properly inside the container for unknown reasons, and the rules for any ports and outgoing or incoming connections could not be established.

Finally, the third approach was successful in stopping one of the client containers from reaching the server and allowing the container that was still online to communicate with its counterpart outside of Matrix. while continuing its normal execution. A visualization of this second test case is present in Fig.6.3.

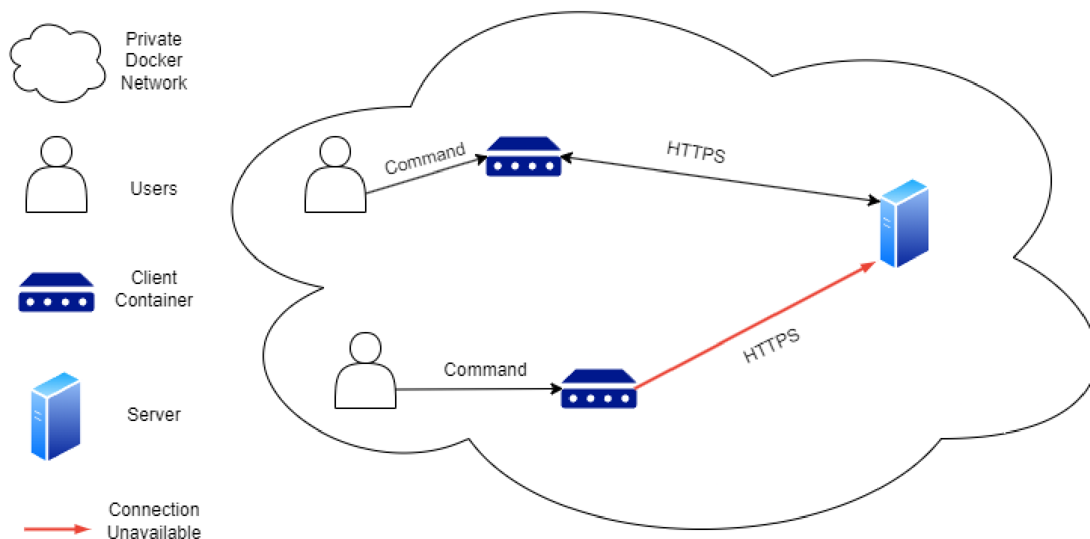


Figure 6.3: Diagram for the second test case, where one of the client containers cannot connect to the server

6.3.3 Test Case: Both clients have unstable connections

This test case was the hardest one to test, given that the two clients' connection status needed to be intermittently and simultaneously changed with the third approach mentioned in the previous section. Figure 6.4 shows the communication pattern of the clients and the server in this test case.

Since the test was made in a certain way, the relevance of its results may be disputed. This will be discussed in the following section.

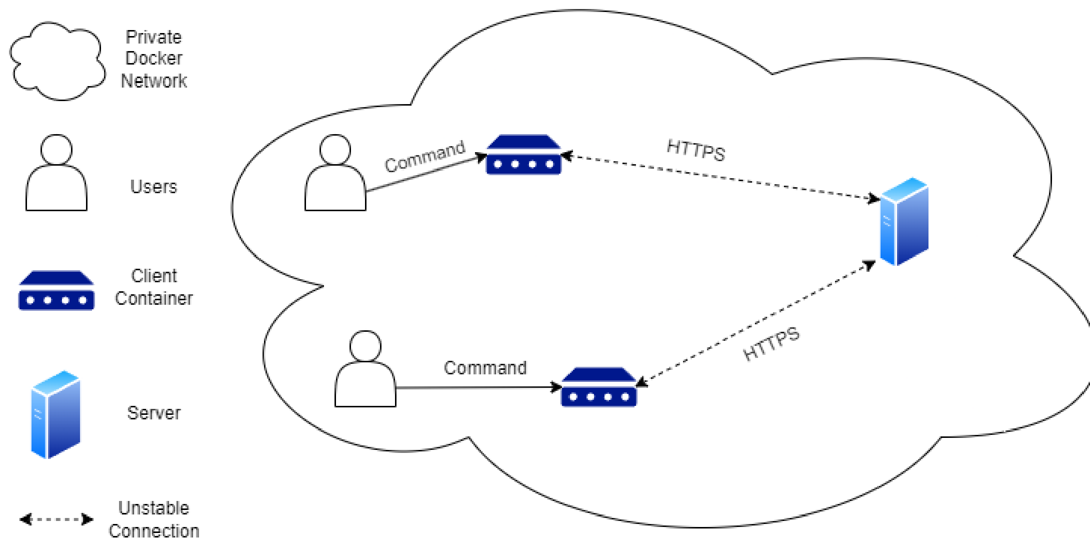


Figure 6.4: Diagram for the test case where the two clients have unstable internet connections

6.4 Results

The results will be presented and discussed individually for each test case. Due to lack of time (the testing process started already very close to the deadline), no statistics such as throughput, scalability, etc., were evaluated. Therefore, these results will only pertain to the prototype's capability of achieving the expected outcomes for each test case.

Table 6.1: Summary of the tests' results

Test Case	Yes	Conditional	No
Regular Communication	✓	-	-
One Offline Client	✓	-	-
Both clients with unstable connections	-	✓	-

Table 6.1 summarizes the results for all three test cases: a checkmark (✓) under the 'Yes' column means that the test case was successful, i.e., the prototype achieved the result that was expected; a checkmark under the 'Conditional' column means that the test was successful in some executions, but not in others; and a checkmark under the 'No' column means that the test case was not successful and that the prototype did not return the expected results from that test.

6.4.1 Regular Communication

The first test case, which tested the regular communication pattern of the prototype through Matrix was extremely successful. Every implemented command returned the expected result when connected to the Matrix server, displaying the correct output in several executions of the commands, even if executed in different orders. Figure 6.1 is an example of the correct and expected behaviour from the application for the *members* command of a given room.

Not only were the tests successful in terms of correctness, but they also displayed positive signs concerning the efficiency of their execution, with some of the commands almost instantly displaying their outputs, maybe due to caching when executing commands referring to a room or simply through efficient storage.

Additionally, when sending and receiving messages, their encryption and decryption also functioned smoothly, as confirmed by some manual debugging with print messages.

The fact that the prototype behaved almost perfectly (there were some runtime bugs in a few corner cases) in this test scenario, proves that Matrix is indeed a valid option for this thesis' proposed use case, propagating messages to the system in a simple, quick and efficient way. This was an encouraging prospect towards the proposed objectives of this thesis, leaving only the offline communication goal to be proven possible.

6.4.2 One Offline Client

The second test case, which tested the implementation of the offline communication protocol, was also very successful, perhaps even more so than expected. Due to the nature of what was being tested, the expectation was that there would be some bugs or even failures in the practical application of the designed protocol, but that was not the case. There is still room for improvement, of course, and much more optimization for this feature, but once the client sent a message while the other one was offline, and it parsed the read receipts received from the server, every step of the proposed protocol for this context was executed perfectly.

The online client successfully recognized that the offline client did not send a read receipt for the message sent, it collected that data and sent it to the alternate goroutine responsible for offline communication. Then, and once again, as confirmed by manual debugging, the goroutine began the mDNS host discovery process in the network, successfully finding and connecting to the offline client and starting the message exchange protocol specified in Fig. 5.2.

The successful execution of this test, together with the successful execution of the first test case, meant that the last remaining objective of this thesis had been met, which was the implementation of a solution capable of offline communication.

6.4.3 Both clients with unstable connections

The third and last test case was the only one where the prototype faltered somewhat. Due to the test's nature, and a lack of optimization in the prototype's implementation, the test's results were often not the ones expected. This may be because the clients were regaining a connection to the server "too fast", not allowing sufficient time for the offline protocol to finish its execution before the client received data from the server. To solve this in the future, if the offline client regains a connection to the server during the protocol's execution, it may signal to the other client that it is now online and that it has received from the server every event that it had missed, thus stopping the protocol's execution.

Another issue that became apparent when testing this situation was a design flaw which had already been noted: after the offline communication protocol finishes and the offline client has received the event, it has no way of acknowledging said event to the server, at least with the current implementation. This is to be expected, given the nature of the context it is in, but a mechanism could be developed to store the message's read receipt in memory, and then send it once the client comes back online. Another alternative could be the addition of a new step to the protocol, and the online client sending the receipt in place of the client that is currently offline.

These would be very valid additions to the prototype's code that were not included in its original version, ultimately leading to less-than-optimal results for this test case.

6.5 Summary

Overall, the prototype's experimental evaluation was a successful one: the desired objectives for this thesis were met with the developed prototype, proven true by the successful results of the first two test cases (subsections 6.4.1 and 6.4.2). By confirming that this solution can successfully communicate both online and offline, the belief in the correctness of the implementation can be turned into a certainty, proving that Matrix can be used for a decentralized and secure communication system, with the support of the libp2p library for offline communication.

Of the three test cases, the prototype performed worse in a use case typical of an IoT context, where clients are constantly connecting and disconnecting from the server, which, in itself, is a concerning outcome. However, there are some relatively simple fixes for this issue, already suggested in Subsection 6.4.3, which, in theory, either mitigate or entirely nullify the problem.

Although the prototype performed admirably in the other two test cases (sections 6.3.1 and 6.3.2), even concerning efficiency, there are still some further optimizations that can be made in the future, suggesting that there is room for a solution like this one to be extremely successful in a real-world use case. These improvements will be discussed in the next chapter: Conclusions.

7 Conclusions

As IoT technologies and digital solutions continue to expand, so do cyber-attacks on these systems. Furthermore, IoT systems deal with an enormous variety of fields of work, that often manage sensitive information, for instance, in financial fields or healthcare. Therefore, securing these types of systems is becoming more and more of a requirement, to protect not only their users but also the system's integrity.

To this end, Intrusion Detection Systems (IDS) are becoming increasingly relevant, given that they monitor other systems and detect any anomalies or privacy violations that occur. Lately, there has been an increase in work and research on Intrusion Detection Systems (IDS). This has mostly focused on creating new methods for detecting adversaries, as well as on validation and deployment strategies. The meteoric rise of AI has indubitably aided these new proposed solutions for IDS, which inevitably means that attackers will also become stronger and develop new methods of attack.

However, this thesis focused not on detecting intrusions or privacy violations, but on finding the most effective way to communicate them to the rest of the system. To keep up with the real-time, evolving, and imbalanced scenario of a smart city's IoT ecosystem, the goal was the development of a secure and decentralized solution that works in harmony with Intrusion Detection Systems, ensuring any detected irregularities are quickly and safely shared throughout the system. Moreover, given that network unavailability is an extremely common scenario in an IoT context, the developed work in this thesis also aimed to provide additional availability to the system, enabling communication between client devices while they are offline, making sure that any detected security-related problems within the system are **always** communicated to every participant, regardless of their connection status.

For this thesis, multiple algorithms and literature regarding this topic were researched. Each one's security properties and how they achieve them were subject to a thorough analysis to determine the most suitable option for the proposed use case.

Hence, based on the Matrix protocol, a decentralised solution that can efficiently and securely communicate any desired information throughout the system was developed, supporting offline communication outside of Matrix, with the help of the *libp2p* library.

The solution, in its regular "mode" of functioning, uses HTTP requests as its form of communication, switching to a P2P form of communication whenever one client is offline, utilizing mDNS to look for other hosts in the network, and TLS together with TCP to effectively communicate with other hosts in that context. There are many other protocols for decentralized communication, which were studied and looked into, however, most of them either have not been implemented in practice or are not able to guarantee all of the security properties deemed indispensable for these purposes. As such, the belief is that the developed solution is one of a kind in that regard, given that it was able to put into practice the Matrix protocol to enable decentralized and efficient communications, while still guaranteeing all of the security properties thought necessary for the developed system (even if it does require additional configurations to guarantee some of them).

7.1 Limitations

Although the proposed solution met the desired objectives, it was still lacking in some areas of its implementation and testing. These are not critical limitations, as they do not undermine any of the work or conclusions presented for this thesis, but they are small adjustments that should have been made, to offer a more overall robust and well-defined solution in the end.

Federation As mentioned in Chapter 4, in the Architecture section, for a Matrix solution to be fully decentralized, there are two requirements:

- All servers within the group must communicate with each other, to ensure that there are no individual points of control over conversations or the network as a whole;
- There must be users connected to different servers in the **same** room so that they can "federate", or synchronize that room's communication history periodically;

Therefore, connecting these requirements to the presented use case, for the proposed architecture to be fully decentralized, there must be a chat room where there are users from different device groups, and, by extension, different homeservers. This way, whenever an event would be sent in that chat room, different servers would receive a copy of that event, thus communicating with each other and no one holding individual control over a conversation or the network entirely.

Due to an overlook, this was not considered when implementing the presented solution. This means that the current implementation lacks some functionality to support this architectural detail. Thankfully, Matrix functions exactly in the same way, whether there are users from multiple servers participating in a room, or just from one server, meaning that the functionality that was described and explained throughout Chapter 5 still holds, even with this additional detail for the system's architecture. To provide for this change, some small adjustments to the proposed architecture and the implementation would have

to be made in the future, for instance, choosing a random client device from each group and creating a room with all of them, where they forward any relevant internal messages from their device group, i.e., intrusions or privacy violations, to that room.

Additionally, for Matrix servers to communicate with each other, they should have registered DNS domains with their respective IP addresses, which, in the proposed architecture, would all be controlled by the system's developers. These domains would essentially reject all incoming traffic that is not from the fellow known Matrix servers of the other device groups, or the client devices, thus maintaining the desired isolation for the system, and enabling the system to function as originally intended.

Device Verification For a user to **trust** another user's device, they must both participate in what Matrix calls "out-of-band verification", where the two devices exchange some sort of information that confirms that they are who they say they are, and that their device is legitimate. This functionality enables users to ensure that the cryptographic identity they are communicating with correctly maps to the intended user. This is intended to prevent man-in-the-middle attacks in cases of first use in contrast to a trust-on-first-use approach. For the proposed system, this verification was done "manually", i.e., since all of the devices were inherently known to the system, it was arbitrarily stated that they were all trusted among themselves. However, in cases where this is not an adequate solution, Matrix uses the Short Authentication String (SAS) protocol. A brief, but in-depth explanation of this protocol will be present at Annex II. In short, with this protocol, two users compute a shared secret. They subsequently derive a short authentication string using the shared secret and details of the connection: any attempts to modify the connection between the two parties should result in them computing different strings. To detect this, the parties compare their short authentication strings through an authenticated out-of-band channel (usually Olm). They then share their cryptographic identities, using the shared secret for verification.

Statistical Results The experimental evaluation started very late concerning the deadline for this thesis, which meant that the results that were presented were somewhat limited since a statistical analysis was not possible. Given the context of this thesis, a statistical analysis of the system, regarding scalability, performance, power usage, etc., would have provided additional and valuable information regarding the quality of the developed solution. Specifically, plots relating the scalability of the system's performance to the number of devices per group, for instance, or the overall power consumption of the client devices, given that it is an IoT system, would have been a valuable ally towards proving the correctness of the solution.

7.2 Main Contributions

This thesis' main contribution was the development of a prototype solution for decentralized and secure communication in the context of a smart city with multiple types of devices and unstable Wi-Fi coverage. The prototype has multiple uses for the scientific community. It can work alongside an IDS for its intended purpose, but it can also be extended to any system that needs a decentralized architecture and additional network fault tolerance, not just in IoT. Unfortunately, its experimental evaluation is somewhat lacklustre, and, as such, a contribution with a full and structured statistical analysis of the system's performance was not possible and is not present, and the only guarantee it can give is that it fulfils its desired purpose in most of the test cases presented.

7.3 Future Work

The work developed in this thesis allowed the identification of some directions for future work, namely:

- Extend the prototype implementation, further optimizing it for an IoT context, for instance: limit and control the size of the databases used following the storage requirements of a specific type of device, improve the efficiency of the code related to offline communication since it is currently severely under-optimized (better host authentication, more efficient and direct host discovery service, perhaps some improvements to the offline communication protocol used as well), improve the command-line interface, and general bug fixing;

- Implement the solution as the communication basis of an Intrusion Detection System. In a setting where there are two different IDSs, and every component functions adequately, this solution could act as the main communication bridge between them, allowing for an efficient and secure alert system, where defence mechanisms could be deployed much more swiftly to counteract the detected intrusion. In this context, each IDS would be connected to a different Matrix server, managing and observing the behaviour of multiple client devices, and through the process of Federation, any detected intrusion or privacy violation would be almost instantly communicated with minimal delay to the other servers, and by extension, the other IDS. Thus, the proposed solution is believed to make an Intrusion Detection System much more capable of adapting to the constantly evolving nature of IoT environments, ensuring that the protection they provide is much more reliable and securely propagated throughout the system;

- Although it was deemed out of scope for this thesis, in the future, a server implementation for the proposed architecture would greatly benefit the solution, by optimizing it for any context that requires a decentralized and secure solution. The

server that was utilised was Matrix Synapse, which is the most used and standard implementation of a Matrix server, and as such, may not be appropriate for some edge cases that are common in IoT systems. Furthermore, with a custom server implementation, the entire system would be under the control of the developer, which is an obvious necessity in a production environment;

- Deploy the proposed system in a real-world scenario and conduct a comprehensive and lengthy study regarding its performance, scalability, power usage, and any other relevant statistical variable, to fully grasp the viability of the solution in a practical setting;
- Develop a fully-fledged API with the same operations as the ones the implemented CLI supports but for any applications;

Bibliography

[1] I. S. R. G. (ISRG). *Let's Encrypt*. URL: https://letsencrypt.org/ (visited on 2023-09-08) (cit. on p. 46).

[2] M. Abdmeziem and F. Charoy. "Securing IoT-based Groups: Efficient, Scalable and Fault-tolerant Key Management Protocol". In: *Ad Hoc & Sensor Wireless Networks* (2019-11) (cit. on p. 27).

[3] M. R. Abdmeziem, D. Tandjaoui, and I. Romdhani. "A Decentralized Batch-Based Group Key Management Protocol for Mobile Internet of Things (DBGK)". In: *2015 IEEE International Conference on Computer and Information Technology- Ubiquitous Computing and Communications- Dependable, Autonomic and Secure Computing- Pervasive Intelligence and Computing*. IEEE, 2015, pp. 1109–1117. DOI: 10.1109/CIT/IUCC/DASC/PICOM.2015.166 (cit. on pp. 12, 24).

[4] J. Alwen, S. Coretti, and Y. Dodis. *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*. Cryptology ePrint Archive, Paper 2018/1037. 2018. URL: https://eprint.iacr.org/2018/1037 (cit. on pp. 8, 9).

[5] J. Alwen et al. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: (2020), pp. 248–277 (cit. on p. 11).

[6] J. Alwen et al. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *IACR Cryptol. ePrint Arch.* 2019 (2020), p. 1189 (cit. on p. 10).

[7] T. Asokan. *gomuks*. URL: https://github.com/tulir/gomuks (visited on 2023-09-22) (cit. on p. 52).

[8] T. Asokan. *mautrix-go*. URL: https://github.com/mautrix/go (visited on 2023-09-08) (cit. on p. 52).

[9] M. Bellare et al. *Ratcheted Encryption and Key Exchange: The Security of Messaging*. Cryptology ePrint Archive, Paper 2016/1028. https://eprint.iacr.org/2016/1028. 2016. URL: https://eprint.iacr.org/2016/1028 (cit. on p. 8).

[10] A. Bienstock, Y. Dodis, and P. Rösler. *On the Price of Concurrency in Group Ratcheting Protocols*. Cryptology ePrint Archive, Paper 2020/1171. 2020. URL: https://eprint.iacr.org/2020/1171 (cit. on p. 11).

[11] C. Bormann et al. *Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*. RFC 6775. 2012-11. DOI: 10.17487/RFC6775. URL: https://www.rfc-editor.org/info/rfc6775 (cit. on p. 23).

[12] E. A. Brewer. *Towards robust distributed systems (abstract) In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'00, 7.. ACM, New York*. 2000 (cit. on p. 15).

[13] J. Callas, A. Johnston, and P. Zimmermann. *ZRTP: Media Path Key Agreement for Unicast Secure RTP*. RFC 6189. 2011-04. DOI: 10.17487/RFC6189. URL: https://www.rfc-editor.org/info/rfc6189 (cit. on p. 97).

[14] Y. Challal and H. Seba. "Group Key Management Protocols: A Novel Taxonomy". In: *Information Technology - IT* 2 (2005-01) (cit. on p. 4).

[15] S. Cheshire and M. Krochmal. *Multicast DNS*. RFC 6762. 2013-02. DOI: 10.17487/RFC6762. URL: https://www.rfc-editor.org/info/rfc6762 (cit. on p. 46).

[16] K. Cohn-Gordon, C. Cremers, and L. Garratt. "On Post-compromise Security". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 164–178. DOI: 10.1109/CSF.2016.19 (cit. on p. 6).

[17] K. Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy (EuroSI&P)*. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27 (cit. on p. 8).

[18] S. Consortium. *SQLite*. URL: https://www.sqlite.org/index.html (visited on 2023-09-14) (cit. on p. 62).

[19] Dammak et al. "Decentralized Lightweight Group Key Management for Dynamic Access Control in IoT Environments". In: *IEEE Transactions on Network and Service Management* 17.3 (2020), pp. 1742–1757. DOI: 10.1109/TNSM.2020.3002957 (cit. on pp. 13, 29).

[20] F. B. Durak and S. Vaudenay. *Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity*. Cryptology ePrint Archive, Paper 2018/889. https://eprint.iacr.org/2018/889. 2018. DOI: 10.1007/978-3-030-26834-3_20. URL: https://eprint.iacr.org/2018/889 (cit. on p. 8).

[21] M. Dworkin et al. *Advanced Encryption Standard (AES)*. en. 2001-11. DOI: https://doi.org/10.6028/NIST.FIPS.197 (cit. on p. 94).

[22] M. J. Dworkin. *SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. Tech. rep. Gaithersburg, MD, USA, 2001. URL: https://csrc.nist.gov/pubs/sp/800/38/a/final (cit. on p. 94).

[23] *Element*. URL: https://element.io/ (visited on 2023-09-04) (cit. on p. 43).

[24] A. Fatani et al. "IoT Intrusion Detection System Using Deep Learning and Enhanced Transient Search Optimization". In: *IEEE Access* 9 (2021), pp. 123448–123464. DOI: 10.1109/ACCESS.2021.3109081 (cit. on p. 1).

[25] M. Foundation. *End-to-End Encryption implementation guide*. 2019. URL: https://matrix.org/docs/guides/end-to-end-encryption-implementation-guide (cit. on p. 10).

[26] M. Foundation. *Matrix APIs*. URL: https://github.com/matrix-org (visited on 2022-11-25) (cit. on p. 10).

[27] M. Foundation. *Matrix Client-Server API*. URL: https://spec.matrix.org/v1.8/client-server-api/ (cit. on p. 46).

[28] M. Foundation. *Matrix E2EE implementation guide*. URL: https://matrix.turbo.fish/docs/legacy/e2e-implementation/ (cit. on p. 55).

[29] M. Foundation. *Matrix Specification*. URL: https://spec.matrix.org/v1.5/ (cit. on p. 15).

[30] M. Foundation. *Olm - A Cryptographic Ratchet*. URL: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md (visited on 2023-01-25) (cit. on p. 16).

[31] M. Foundation. *Synapse*. URL: https://github.com/matrix-org/synapse (visited on 2023-09-04) (cit. on p. 43).

[32] R. P. Foundation. URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi-5.html (visited on 2024-01-12) (cit. on p. 2).

[33] S. Francia. *Cobra*. URL: https://github.com/spf13/cobra (visited on 2023-09-18) (cit. on p. 65).

[34] R. C. Gangwar. "Journal of Theoretical and Applied Information Technology Secure and Efficient Decentralized Group Key Establishment Protocol for Robust Group Communication". In: URL: https://api.semanticscholar.org/CorpusID:11485316 (cit. on p. 5).

[35] D. George. *MicroPython*. URL: https://micropython.org/ (visited on 2023-09-08) (cit. on p. 52).

[36] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (2002-06), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on p. 15).

[37] Google. URL: https://pkg.go.dev/encoding/gob (visited on 2023-09-11) (cit. on p. 60).

[38] Google. *A Tour of GO*. URL: https://go.dev/tour/concurrency/1 (visited on 2023-09-11) (cit. on p. 57).

[39] Google. *gzip*. URL: https://pkg.go.dev/compress/gzip (visited on 2023-09-12) (cit. on p. 60).

[40] P. G. D. Group. *PostgreSQL*. URL: https://www.postgresql.org/ (visited on 2023-09-08) (cit. on p. 46).

[41] *HMAC: Keyed-hashing for message authentication*. 1997-02. URL: https://www.ietf.org/rfc/rfc2104.txt (cit. on p. 94).

[42] R. Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652. 2009-09. DOI: 10.17487/RFC5652. URL: https://www.rfc-editor.org/info/rfc5652 (cit. on p. 94).

[43] J. Hur, Y. Shin, and H. Yoon. "Decentralized Group Key Management for Dynamic Networks Using Proxy Cryptography". In: (2014). URL: https://www.researchgate.net/publication/221453988_Decentralized_group_key_management_for_dynamic_networks_using_proxy_cryptography (cit. on p. 12).

[44] "IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques". In: *IEEE Std 1363a-2004 (Amendment to IEEE Std 1363-2000)* (2004), pp. 1–167. DOI: 10.1109/IEEESTD.2004.94612 (cit. on p. 6).

[45] D. Inc. *Docker Desktop: The #1 containerization software for developers and teams*. URL: https://www.docker.com/products/docker-desktop/ (visited on 2023-09-29) (cit. on p. 72).

[46] L. Inria et al. "Dynamic Group Communication Security". In: *Proceedings. Sixth IEEE Symposium on Computers and Communications*. Los Alamitos, CA, US: IEEE Computer Society, 2001-07, p. 0049. DOI: 10.1109/ISCC.2001.935354. URL: https://doi.ieeecomputersociety.org/10.1109/ISCC.2001.935354 (cit. on p. 12).

[47] IPFS. *libp2p*. URL: https://docs.libp2p.io/concepts/introduction/overview/ (visited on 2023-09-20) (cit. on p. 46).

[48] A. Ivan and Y. Dodis. *Proxy Cryptography Revisited*. New York University, New York, NY 10012. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/Proxy-Cryptography-Revisited-Anca-Ivan.pdf (cit. on p. 13).

[49] J. Jaeger and I. Stepanovs. *Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging*. Cryptology ePrint Archive, Paper 2018/553. https://eprint.iacr.org/2018/553. 2018. URL: https://eprint.iacr.org/2018/553 (cit. on p. 8).

[50] B. Johnson. *bbolt*. URL: https://pkg.go.dev/go.etcd.io/bbolt@v1.3.7 (visited on 2023-09-11) (cit. on p. 60).

[51] D. Jost, U. Maurer, and M. Mularczyk. *Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging*. Cryptology ePrint Archive, Paper 2018/954. https://eprint.iacr.org/2018/954. 2018. URL: https://eprint.iacr.org/2018/954 (cit. on p. 36).

[52] D. Jost, U. Maurer, and M. Mularczyk. "Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging". In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Y. Ishai and V. Rijmen. Cham: Springer International Publishing, 2019, pp. 159–188. ISBN: 978-3-030-17653-2 (cit. on p. 8).

[53] E. R. Karthikeyan Bhargavan Richard Barnes. "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. A protocol proposal for Messaging Layer Security (MLS)". In: (2018) (cit. on p. 10).

[54] A. Khraisat and A. Alazab. "A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges". In: *Cybersecurity* (2021-12). DOI: 10.1186/s42400-021-00077-7 (cit. on p. 1).

[55] A. Khraisat et al. "Survey of intrusion detection systems: techniques, datasets and challenges". In: *Cybersecurity* 2.1 (2019), pp. 1–22 (cit. on p. 1).

[56] E. Kim, D. Kaspar, and J. Vasseur. *Design and Application Spaces for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*. RFC 6568. 2012-04. DOI: 10.17487/RFC6568. URL: https://www.rfc-editor.org/info/rfc6568 (cit. on p. 23).

[57] D. H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. 2010-05. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869 (cit. on p. 94).

[58] H. Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. Cryptology ePrint Archive, Paper 2010/264. https://eprint.iacr.org/2010/264. 2010. URL: https://eprint.iacr.org/2010/264 (cit. on p. 37).

[59] H. Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by T. Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648. ISBN: 978-3-642-14623-7 (cit. on p. 94).

[60] H. Krawczyk and P. J. Eronen. "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)". In: *RFC* 5869 (2010), pp. 1–14 (cit. on p. 37).

[61] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[62] M. Marlinspike and T. Perrin. *The Double Ratchet Algorithm*. URL: https://signal.org/docs/specifications/doubleratchet/ (cit. on pp. 8, 9, 16).

[63] M. Marlinspike and T. Perrin. *The X3DH Key Agreement Protocol*. 2016. URL: `https://www.cs.rit.edu/~spr/COURSES/CRYPTO/x3dh.pdf` (cit. on pp. 9, 36).

[64] B. D. (o. S. Martin Albrecht (University of London) Sofía Celi (Brave Software) and D. J. (of London). "Practically-exploitable Cryptographic Vulnerabilities in Matrix". In: (2022) (cit. on pp. 10, 49).

[65] mattn. *go-sqlite3*. URL: `https://pkg.go.dev/github.com/mattn/go-sqlite3@v1.14.17` (visited on 2023-09-14) (cit. on p. 62).

[66] *mautrix-go*. URL: `https://pkg.go.dev/maunium.net/go/mautrix#Syncer` (visited on 2023-09-09) (cit. on p. 54).

[67] *Naming a package*. URL: `https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html` (cit. on p. 16).

[68] Y. Napora. *noise-libp2p - Secure Channel Handshake*. URL: `https://github.com/libp2p/specs/tree/master/noise` (visited on 2023-09-21) (cit. on p. 47).

[69] F. Nginx. *F5 Nginx*. URL: `https://www.nginx.com/` (visited on 2023-09-08) (cit. on p. 46).

[70] E. Omara et al. "The Messaging Layer Security (MLS) Architecture". In: (2020) (cit. on p. 10).

[71] V. Pardeshi et al. "Health monitoring systems using IoT and Raspberry Pi — A review". In: *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. 2017, pp. 134–137. DOI: `10.1109/ICIMIA.2017.7975587` (cit. on p. 2).

[72] M.-H. Park et al. "Key Management for Multiple Multicast Groups in Wireless Networks". In: *IEEE Transactions on Mobile Computing* 12.9 (2013), pp. 1712–1723. DOI: `10.1109/TMC.2012.135` (cit. on p. 28).

[73] T. Perrin. *The Noise Protocol Framework*. URL: `https://noiseprotocol.org/noise.html` (visited on 2023-09-21) (cit. on p. 47).

[74] B. Poettering and P. Rösler. "Towards Bidirectional Ratcheted Key Exchange". In: *Advances in Cryptology – CRYPTO 2018*. Ed. by H. Shacham and A. Boldyreva. Cham: Springer International Publishing, 2018, pp. 3–32. ISBN: 978-3-319-96884-1 (cit. on p. 8).

[75] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018-08. DOI: `10.17487/RFC8446`. URL: `https://www.rfc-editor.org/info/rfc8446` (cit. on p. 47).

[76] P. Rösler, C. Mainka, and J. Schwenk. "More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema". In: IEEE, 2018, pp. 415–429 (cit. on p. 9).

[77] M. N. Rusty Russell. URL: https://linux.die.net/man/8/iptables (visited on 2023-09-23) (cit. on p. 76).

[78] *Secure Hash Standard (shs)*. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (cit. on pp. 25, 94).

[79] A. Sforzin et al. "RPiDS: Raspberry Pi IDS — A Fruitful Intrusion Detection System for IoT". In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. 2016, pp. 440–448. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0080 (cit. on p. 2).

[80] S. Sobol. *Online deadlock detection in go (golang)*. URL: https://github.com/sasha-s/go-deadlock (visited on 2023-09-11) (cit. on p. 60).

[81] N. I. of Standards and Technology. *Minimum Security Requirements for Federal Information and Information Systems*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 200. Washington, D.C.: U.S. Department of Commerce, 2006. DOI: 10.6028/nist.fips.140-2 (cit. on p. 5).

[82] K. Stine et al. *Volume II: Appendices to Guide for Mapping Types of Information and Information Systems to Security Categories*. Tech. rep. U.S. Department of Commerce, 2008. DOI: 10.6028/NIST.SP.800-60v2r1r (cit. on p. 5).

[83] B. Strauch. *cobra-shell*. URL: https://github.com/brianstrauch/cobra-shell (visited on 2023-09-18) (cit. on p. 66).

[84] I. Sysoev. *Nginx Reverse Proxy*. URL: https://nginx.org/en/docs/http/ngx_http_proxy_module.html (visited on 2023-09-08) (cit. on p. 46).

[85] N. Unger et al. "SoK: Secure Messaging". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 232–249. DOI: 10.1109/SP.2015.22 (cit. on p. 8).

[86] M. Weidner et al. *Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees*. Cryptology ePrint Archive, Paper 2020/1281. https://eprint.iacr.org/2020/1281. 2020. URL: https://eprint.iacr.org/2020/1281 (cit. on pp. 13, 35).

[87] M. A. Weidner. "Group Messaging for Secure Asynchronous Collaboration". In: (2019) (cit. on p. 11).

[88] Whatsapp. *Whatsapp Encryption Overview*. 2017. URL: https://perma.cc/QD7M-GPG5 (cit. on p. 9).

[89] J. Wilder. *docker-letsencrypt-nginx-proxy-companion*. URL: https://github.com/jwilder/docker-letsencrypt-nginx-proxy-companion (visited on 2023-09-08) (cit. on p. 46).

[90] C. K. Wong, M. Gouda, and S. Lam. "Secure group communications using key graphs". In: *IEEE/ACM Transactions on Networking* 8.1 (2000), pp. 16–30. DOI: 10.1109/90.836475 (cit. on p. 27).

I

Matrix Encryption Primitives

I.1 Algorithms

Matrix cryptography makes use of the following cryptographic algorithms, which will be referenced in other sections:

- sort($x_1, x2, ..., x_n$) - returns a sorted copy of the list [$x_1, x2, ..., x_n$];
- HMAC-SHA-256(k, m) - a Hash-based Message Authentication Code (HMAC) constructed with the SHA-256 hash function [78] taking as input a key k and a message m [41];
- HKDF-SHA-256(s, k, c, l) - a Hash-based Key Derivation Function (HKDF) constructed with SHA-256 where s is the salt, k is the secret key material, c is the info/context and l is the output length in bytes [57, 59];
- AES(k, m) - an AES [21] taking a key k and a message block m of size 128 bits. Matrix uses AES-256, i.e., keys of length 256 bits;
- AES-CTR(iv, k, m) is AES in counter (CTR) mode [22], where iv is the *nonce*, k is an AES encryption key and m is a message;
- AES-CBC(iv, k, m) is AES in cipher block chaining (CBC) mode [22], where iv is the *nonce*, k is an AES encryption key and m is a message. Matrix uses PKCS7 [42] padding to split plaintexts into blocks for CBC mode in the Olm and Megolm protocols;

I.2 Megolm

Megolm **sessions** consist of the following components: (a) Group signing and verification keys (gsk, gpk) using Ed25519; (b) Megolm ratchet R and (c) Megolm ratchet index i. While the nature of the ratchet in Megolm deviates from Olm and Signal, and is one of the innovations of the Megolm protocol, its details do not matter for our purposes. Thus,

we may simply think of (i, R) as some symmetric key material. The group verification key gpk is used as the unique identifier for sessions. As such, it is also referred to as the *session identifier*. Megolm sessions are also referred to as the *room key*. A Megolm session is split into an outbound and inbound session. The inbound session allows the holder to decrypt and verify messages that were encrypted and signed by the outbound session. The outbound session contains $\mathfrak{S}_{gsk} := (i, R, gsk, gpk)$ while the inbound session contains $\mathfrak{S}_{gpk} := (i, R, gpk)$. Messages are encrypted using AES in CBC mode with HMAC applied to the ciphertext, so it is an encrypt-then-MAC construction. This authenticated ciphertext is then signed with gsk and the result is sent to the homeserver for distribution.

Algorithms 23, 24 and 25 describe how Megolm sessions are initiated, as well as how messages are encrypted and decrypted, respectively.

Algorithm 23 Megolm Init function

function MEGOLM.INIT(1^n)
  $i \leftarrow 0$
  $R \leftarrow \{0, 1\}^{1024}$
  $(gsk, gpk) \leftarrow$ Ed25519.KGen(1^n)
  ver $\leftarrow 0x03$
  $\sigma_{mg} \leftarrow$ Ed25519.Sign(gsk, (ver,i, R, gpk))
  $\mathfrak{S}_{gsk} \leftarrow$ (ver,i, R, gsk, gpk)
  $\mathfrak{S}_{gpk} \leftarrow$ (ver,i, R, gpk)
  return $\mathfrak{S}_{gsk}, \mathfrak{S}_{gpk}, \sigma_{mg}$
end function

Algorithm 24 Megolm Encrypt function

function MEGOLM.ENCRYPT($\mathfrak{S}_{gsk}, m$)
  (ver,i, R, gsk, gpk) $\leftarrow \mathfrak{S}_{gsk}$
  $k_e \,||\, k_h \,||\, k_{iv} \leftarrow$ HKDF(0, R,"MEGOLM KEYS",80)
  $c \leftarrow$ AES-CBC(k_{iv}, k_e, m)
  $\tau \leftarrow$ HMAC(k_h, (ver,i, c))[0:8]
  $\sigma \leftarrow$ Ed25519.Sign(gsk,(ver,i, c, τ))
  $c' \leftarrow$(veri, c, τ, σ)
  $i, R \leftarrow$ Megolm.RatchetAdvance(i, R)
  $\mathfrak{S}_{gsk} \leftarrow$ (ver,i, R, gsk, gpk)
  return ($\mathfrak{S}_{gsk}, c'$)
end function

Algorithm 25 Megolm Decrypt function

function MEGOLM.DECRYPT($\mathfrak{S}_{gpk}, c$)
  (ver,i, R, gpk) ← $\mathfrak{S}_{gpk}$
  (ver$'$, i', c', τ, σ) ← c
  if !Ed25519.Verify(gpk, σ,(ver,i', c', τ)) **then**
    return ($\mathfrak{S}_{gpk}, \perp$)
  end if
  (i, R) ← Megolm.RatchetAdvance$^{i'-i}(i, R)$
  $k_e \mid\mid k_h \mid\mid k_{iv}$ ← HKDF(0, R',"MEGOLM KEYS",80)
  if $\tau \neq$ HMAC(k_h,(ver,i, c')[0:8] **then**
    return ($\mathfrak{S}_{gpk}, \perp$)
  end if
  m ← AES-CBC.Dec($k_i v, k_e, c'$)
  $\mathfrak{S}_{gpk}$ ←(ver, i, R, gpk)
  return $\mathfrak{S}_{gpk}, m$
end function

II

Matrix Out-of-band Verification - SAS protocol

We briefly describe Matrix's SAS protocol. The SAS protocol builds upon the ZRTP key agreement handshake [13]. It uses an ephemeral X25519 key exchange to compute a shared secret. They subsequently derive a short authentication string using the shared secret and details of the connection: any attempts to modify the connection between the two parties should result in them computing different strings. To detect this, the parties compare their short authentication strings through an authenticated out-of-band channel. They then share their cryptographic identities, using the shared secret for verification.

Figures II.1 and II.2 describe the SAS protocol for out-of-band verification. When clients source keys from their homeserver, we represent this through a call to the algorithm HS.QueryKey. It takes as input a string representing the key type, followed by a series of indices to identify the particular key. For example, HS.QueryKey("dpk",A, i) returns $dpk_{A,i}$. Once the shared secret has been generated, each party compiles a list of the keys they wish to have signed into an m.key.verification.mac message (Fig.II.3). The shared secret is used to compute a message authentication code (MAC) for each key, calculated over its public part and details from the SAS protocol execution. A second MAC is computed over a list of key identifiers, corresponding to the list of keys for which MACs have been included. These MACs are added to the message, and ensure that only parties in possession of the shared secret can request keys for signing.

Out-of-band verification is used in two cases. (1) Two users are verifying each other: The protocol is executed between a device from each user (each of which holds the user's secret cross-signing keys), and each includes their master cross-signing key mpk in the *m.key.verification.mac*, which the other device will sign using their user signing key usk. (2) A user is verifying one of their own devices: The protocol is executed between the verifying device (which holds the cross-signing secret keys) and the new device. The device being verified uses the *m.key.verification.mac* message to send their device identity key dpk to the verifying device. The verifying device uses the device self-signing key ssk to sign the new device's identity key dpk and Olm identity key ipk.

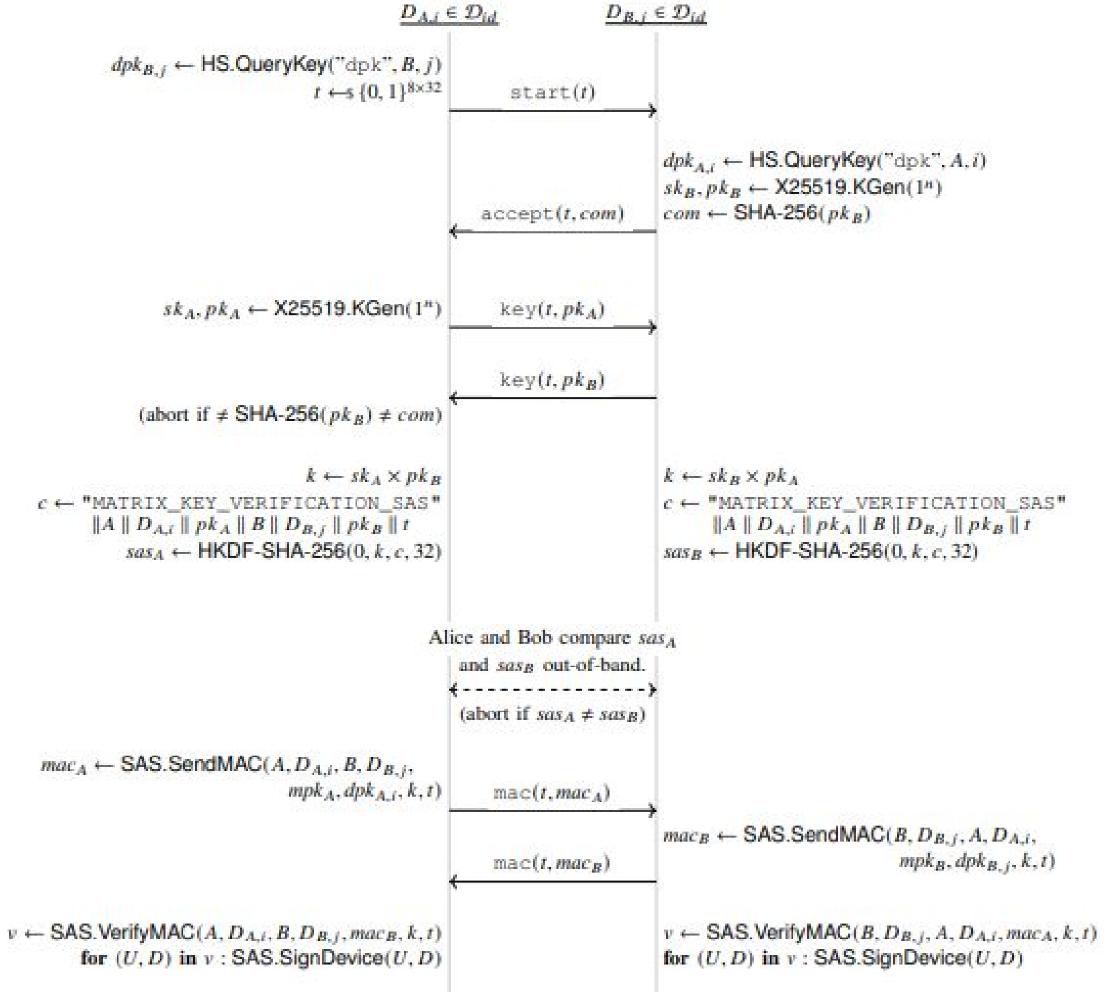


Figure II.1: A sequence diagram summarising the SAS protocol. The contents of *m.key.verification.mac* are described in Fig.II.3, while pseudocode descriptions of SAS.SendMAC, SAS.VerifyMAC and SAS.SignDevice are in Fig.II.2. Message types in the above diagram have had the prefix *m.key.verification.* removed.

SAS.CalcMAC(k, m, c)

$k' \leftarrow$ HKDF-SHA-256$(0, k, c, 32)$
$mac \leftarrow$ HMAC-SHA-256(k', m)
return mac

SAS.SendMAC$(A, D_{A,i}, B, D_{B,j}, mpk, dpk, k, t)$

$c \leftarrow$ "MATRIX_KEY_VERIFICATION_MAC" \
  $\| A \| D_{A,i} \| B \| D_{B,j} \| t$
$id_{\text{dev}} \leftarrow$ "ed25519:" $\| D_{A,i}$
$mac_{\text{dev}} \leftarrow$ SAS.CalcMAC$(k, dpk, c \| id_{\text{dev}})$
$id_{\text{cs}} \leftarrow$ "ed25519:" $\| mpk$
$mac_{\text{cs}} \leftarrow$ SAS.CalcMAC$(k, mpk, c \| id_{\text{cs}})$
$ms \leftarrow ((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$
$ks \leftarrow$ SAS.CalcMAC$(k$, sort$(id_{\text{dev}}, id_{\text{cs}}), c \|$ "KEY_IDS"$)$
return (ms, ks)

SAS.SignDevice$(A, D_{A,i})$

// Check whether $D_{A,i}$ is a cross-signing identity
$mpk \leftarrow$ HS.QueryKey("mpk", A)
if $D_{A,i} = mpk$ **then**
  return UserVerified(A, mpk)
// Otherwise, $D_{A,i}$ refers to a device
else
  return DeviceVerified$(A, D_{A,i})$

SAS.VerifyMAC$(A, D_{A,i}, B, D_{B,j}, mac, k, t)$

$((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}), ks) \leftarrow mac$
$c \leftarrow$ "MATRIX_KEY_VERIFICATION_MAC" \
  $\| B \| D_{B,j} \| A \| D_{A,i} \| t$
$ks' \leftarrow$ SAS.CalcMAC$(k$, sort$(id_{\text{dev}}, id_{\text{cs}}), c \|$ "KEY_IDS"$)$
if $(ks' \neq ks)$ **then**
  return $\emptyset$
$v \leftarrow \emptyset$
for (id, mac) **in** $((id_{\text{dev}}, mac_{\text{dev}}), (id_{\text{cs}}, mac_{\text{cs}}))$
  "ed25519:" $\| D_{B,j} \leftarrow id$
  // Check if this is a device verification request
  $dpk \leftarrow$ HS.QueryKey("dpk", $B, D_{B,j}$)
  if $dpk \neq \perp$ **then**
    $D \leftarrow x$
    if $mac =$ SAS.CalcMAC$(k, dpk, c \| id)$ **then**
      $v \leftarrow v \cup \{(B, D)\}$
  // Check if this is a cross-signing verification request
  elseif $(x =$ HS.QueryKey("mpk", $B)$
      $\cap\ mac =$ SAS.CalcMAC$(k, x, c \| id))$ **then**
    $mpk \leftarrow x$
    $v \leftarrow v \cup \{(B, mpk)\}$
return v

Figure II.2: Algorithms to generate, verify and process m.key.verification.mac messages in the SAS protocol. UserVerified signs the given user's master cross-signing key with the current device's user-signing key. Similarly, DeviceVerified signs the given device's fingerprint and Olm identity keys with the current device's self-signing key. These signatures are uploaded to the homeserver.

```
{"mac": {"ed25519:<device_id>":
    SAS.CalcMAC(k, dpk, c || "ed25519:<device_id>"),
  "ed25519:<mpk>":
    SAS.CalcMAC(k, mpk, c || "ed25519:<mpk>"),},
"keys": SAS.CalcMAC}(
   k, sort("ed25519:<device_id>", "ed25519:<mpk>"),
   c || "KEY_IDS")
```

Figure II.3: The format of an *m.key.verification.mac* message for a user with cross-signing setup

2024
Key Agreement for Decentralized Secure Intrusion Detection
João Soares
NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY