Expertise
and insight
for the future

Secure and Scalable Group Messaging with MLS Protocol

Adarsh Krishnan

Metropolia University of Applied Sciences
Master of Engineering
Information Technology
Master's Thesis
16th May 2022

Metropolia
University of Applied Sciences

Preface

It has been a challenging but rewarding experience to work on this project. In the fast-moving area of Computer Science and Software Engineering has been surprising to see the problem of a secure, scalable and open group messaging standard unsolved. As a result, my experience has been somewhat like a venture into the dark unknown.

I would like to thank Metropolia University of Applied Sciences for offering me this opportunity to conduct this study and in particular my course coordinator Ville Jääskeläinen and my principal lecturer Auvo Häkkinen for guiding and supporting me though this project.

Furthermore, this thesis would have been several magnitudes harder to complete without the loving help and support of my partner who has made it easy for me to conduct the study through the exciting but challenging first year of our son.

Dedicated to Edi and Henna.

Helsinki, 16th May 2022
Adarsh Krishnan

Author Title Number of Pages Date	Adarsh Krishnan Secure and Scalable Group Messaging with MLS Protocol 37 pages + 1 appendix 16 May 2022
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Auvo Häkkinen, Principal Lecturer Ville Jääskeläinen, Course coordinator

This thesis explores the shortcomings and problems of the current scenario of group messaging. The lack of an open standard for secure and scalable group messaging was understood.

This project investigates the IETF Internet-Draft produced by the MLS Working Group for the Messaging Layer Security (MLS) protocol. The MLS protocol drafted by members of the messaging industry and relevant academia to address the need for an open standard for secure and scalable group messaging.

In this project, the MLS protocol and architecture were studied to reflect how MLS could solve the need for a secure layer for group messaging that could scale with the ever-increasing number of users and messaging volumes. Furthermore, the OpenMLS implementation of the MLS protocol was studied and evaluated.

Finally, the thesis concludes by providing an entry point for implementing the MLS protocol to be taken into use in a group messaging application. It is however noted that this protocol is still a draft at the time of writing Thus, it should only be taken into production use once the MLS protocol has been researched further by academia and the industry and eventually approved by the IETF as an RFC.

Keywords	group messaging, messaging layer security, MLS

Contents

Preface
Abstract
List of Figures (Tables)
List of Abbreviations

1 Introduction 1

2 Present State of Group Messaging 3

3 Secure and Scalable Messaging 6

3.1 Security Properties 6

3.1.1 End-to-End Encryption (E2EE) 6

3.1.2 Message Integrity 6

3.1.3 Forward Secrecy (FS) 8

3.1.4 Post Compromise Security (PCS) 8

3.2 Scalability 9

3.3 Asynchronicity 9

4 MLS Protocol 10

4.1 Motivation for MLS 10

4.2 Current State of MLS 10

4.3 Scope of MLS 11

4.4 Operation of MLS Protocol 11

4.4.1 Ratchet Tree 12

4.4.2 KeyPackage 13

4.4.3 Epoch and Message Ordering 14

4.4.4 Group Creation 14

4.4.5 Group Proposal Operations 16

4.4.6 Group Commit 17

4.4.7 Group Application Messages 18

5 MLS Architecture 19

5.1 Components 19

5.1.1 Group 19

5.1.2 User 19

5.1.3 Client 20

5.1.4 Member ... *20*
5.1.5 Service Provider (SP) ... *20*
5.2 Functionalities of the Service Provider ... 21
5.2.1 Authentication Service ... *22*
5.2.2 Delivery Service ... *23*
5.3 Membership Changes ... 26
5.4 Asynchronicity ... 27
5.5 Access Control ... 27
5.6 Multi-device Support ... 27
5.7 Additional Considerations ... 28
6 Practical exploration with OpenMLS ... 29
6.1 Rust ... 29
6.2 MLS Delivery Service (DS) ... 30
6.3 OpenMLS Command Line Interface (CLI) ... 31
6.4 Analysis of Group Messaging with OpenMLS ... 32
6.4.1 Basic Group Operations ... *33*
6.4.2 Scaling Test ... *34*
6.5 Analysis of the Results ... 36
7 Conclusions ... 37
7.1 Shortcomings of MLS and OpenMLS ... 38
7.2 Considerations for the Future ... 38
References ... 39
Appendix ... 42

List of Figures

Figure 2.1: Pairwise encryption of a single message in a group. 3
Figure 3.1: Diagrammatic representation of digital signature for message integrity. 7
Figure 4.1: Comparison of the scope of MLS and TLS. 11
Figure 4.2: Example ratchet tree of a group with 3 members. 13
Figure 4.3: Visual representation of group creation with invitation to one member. 15
Figure 6.1: Example log output of the OpenMLS DS. 31
Figure 6.2: Screenshot of MLS client operations supported by the OpenMLS CLI. 32
Figure 6.3: Screenshot of a size limitation with OpenMLS DS. 33
Figure 6.4: Screenshot of a test result of two-member group with OpenMLS. 33
Figure 6.5: Screenshot of scaling test output. 34
Figure 6.6: Graph of time to send messages by number of messages. 34
Figure 6.7: Graph of time to send 10 messages by number of members. 35

List of Tables

Table 6.1: List of methods implemented by OpenMLS DS. 30

List of Abbreviations

ACL	Access Control List A list of rules or permissions associated with a resource.
API	Application Programmer Interface A programmatic interface for a system that a programmer may use to interact with it.
ART	Asynchronous Ratcheting Tree A protocol for end-to-end encrypted group messaging.
AS	Authentication Service A service that provides authentication within the MLS architecture.
CA	Certificate Authority A trusted entity that issues digital certificates.
CLI	Command Line Interface A terminal interface used to interact with the underlying system.
DS	Delivery Service A service that provides delivery of messages within the MLS architecture.
E2EE	End-to-End Encryption A messaging pattern where the message is an encrypted throughout the channel of transmission.
FS	Forward Secrecy A security property referring to security of data prior to a compromise.
IETF	Internet Engineering Task Force An open standards organization for matters related to the internet.
MLS	Messaging Layer Security A protocol for secure and scalable group messaging.
PCS	Post Compromise Secrecy A security property referring to security of data after a compromise.
PGP	Pretty Good Privacy An encryption program.
PKI	Public Key Infrastructure A set of components that provides functionality for managing public-key encryption.

RFC	Request for Comments Publication that serves as a technical guideline for a system or protocol.
S/MIME	Secure/Multipurpose Internet Mail Extensions A standard for public key encryption.
SP	Service Provider An entity that provides MLS as a service.
TCP	Transmission Control Protocol A protocol that stipulates the methodology for establishing network connections and transmitting data.
TLS	Transport Layer Security A security protocol that enables transport of encrypted messages.
TreeKEM	Tree Key Encapsulation Mechanism A group messaging protocol. An alternative to ART.
UDP	User Datagram Protocol A communication protocol to communicate over computer networks without the requirement for acknowledgements like TCP.
WebRTC	Web Real-Time Communication A messaging protocol for real-time communication like video and audio.
XMPP	eXtensible Messaging and Presence Protocol A messaging protocol for instant messaging.

1 Introduction

Communication between people has rapidly evolved over the past century and especially the past few decades. As humanity has moved from physical means like mail, to increasingly faster and sophisticated means through the use of electronic and computer systems. In the present time the underlying workings of communication operate at the speed of light.

With the growth of the internet, through which a huge bulk of the messaging happens contemporarily, messaging services and applications have truly become ubiquitous and ever growing.

Such rapid growth has brought upon several needs and challenges in the realm of communication, such as the need for rapid growth in the messaging capacity while maintaining and bringing forward more secure end-to-end channels for communication.

Additionally, there has been a growth in the usage of group messaging, that is messaging through a channel like abstraction that includes more than two parties that are communicating. The size of the group is also increasing, especially in certain settings, such as enterprise communication. In such cases, a group might comprise a large section of employees within an organization. The number of participants in the group may span somewhere in the hundreds or thousands. To establish a secure channel for communication with modern security properties, particularly end-to-end encryption between all parties by naive means involves a lot of heavy computation and the need for efficiency and scalability is therefore paramount.

One-to-one secure communication with a secure end-to-end encrypted channel with modern security properties is a well understood and applied matter. There are several open and closed source implementations provided by various companies for public use via their applications. For example, Wire and WhatsApp using the open source Signal protocol and Telegram using a proprietary, closed source protocol developed in-house.

While this is the scenario for one-to-one messaging, group messaging lacks a solution that is scalable and provides desirable modern security properties.

Given the need for a secure, end-to-end encrypted, scalable group messaging, the Internet Engineering Task Force (IETF) in collaboration with some industrial and academic organizations has produced a draft Request for Comments (RFC) document. This document outlines the Messaging Layer Security (MLS) protocol that has been written for the purpose of addressing this problem [2].

The aim of this thesis is to provide a practical framework for any stakeholder interested in the messaging technology to implement a service to solve the problem of offering group messaging.

The group messaging must be scalable. Additionally, the group messaging must be secure in the sense of providing end-to-end encrypted channel between all participants with modern security properties.

The security properties include message confidentiality and integrity, membership authentication, forward secrecy and post-compromise security. Additionally, asynchronicity is another requirement of the group messaging.

To fulfill the goal of this thesis, it is scoped to provide a technical system and architecture design that can be implemented at least as a prototype messaging service that includes applications such that scalable, asynchronous and secure group messaging as outlined in the aim of the thesis is fulfilled. For the sake of brevity, the goal of this thesis does not include delivering a fully working prototype. More specifically, this thesis intends to a practical entry point for the implementation for an MLS group messaging service.

This thesis and the associated research and work is carried out by the author solely and not in collaboration with any organization.

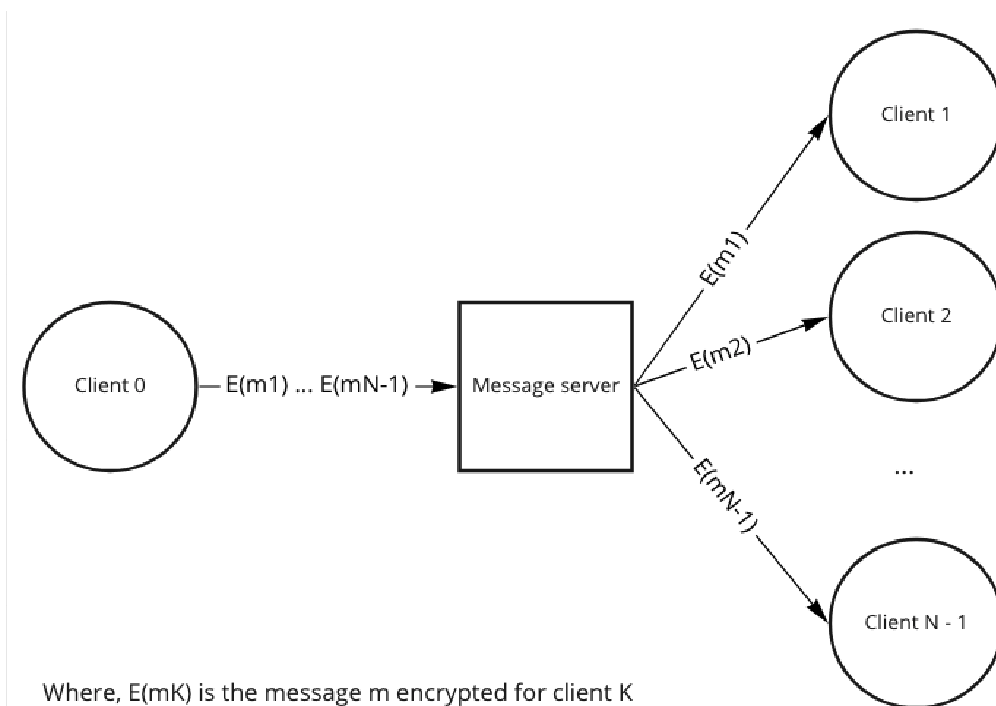
2 Present State of Group Messaging

One-to-one secure, end-to-end encrypted messaging is a well understood and adopted technology from both an industrial and an academic perspective. Several protocols and technologies have been developed and used over the years such as PGP and S/MIME. In the present day, the Signal protocol is in wide use for end-to-end encrypted messaging. While some applications and companies use different and proprietary protocols, the technology is fairly standardized.

The present scenario for group messaging is not standardized across various applications. Most companies extend the use of the regular one-to-one end-to-end encryption scheme to match the group structure.

Figure 2.1 illustrates a scenario where a message needs to be encrypted pairwise to each member within a group.

Figure 2.1: Pairwise encryption of a single message in a group.



A naive approach to this problem used by some services is to create an end-to-end encrypted channel between every pair of clients in the group. This yields a very unscalable system where any cryptographic operation to maintain the end-to-end encryption requires operations on the order of N^2 where N is the size of the group.

Some companies have approached this with solutions that are more scalable. There are several such approaches, and they are proprietary and non-standard. As an example, WhatsApp uses a server-side fan out over the Signal protocol leveraging the protocol's Sender Keys feature. In this system a joining client to a group sends out a key to all other member clients in the group. The clients then add it to their Chain Key. For the subsequent messages, a sender uses their chain key to encrypt their message [1]. This approach provides forward secrecy and each action of adding a member to the group lends itself to N operations where the size of the group is N. For larger sizes of groups, this does not scale.

The lack of standardized and open-sourced protocols for the purpose yields a scenario where there are proprietary systems that lack the level of academic and mass attention which undermines the level of security. Additionally, there is a lack of a protocol that allows scaling to very large group sizes. Generally, there is a tradeoff between maintaining the security of a group messaging service, by the virtue of modern security properties and scalability of the service.

Practically, there is a great need for both security and scalability in a messaging service. This requirement is a result of an ever-growing user-base. The need for protection of privacy and secrets is ever more important for several reasons including right to privacy being widely accepted as a human right. Also, the safety of confidential information is of paramount importance to a business's operations.

In the world there is a huge requirement for group messaging in both personal lives and messaging within companies of most industries. Particularly, large companies where the number of employees may be in the several thousands, there is a great need for efficient, scalable and secure communication for smooth operations while keeping the trade secrets safe.

There are pairwise protocols such as Signal that can already offer strong security guarantees such as Post Compromise Security (PCS). However, this is inefficient for group messaging [3]. The Internet Engineering Task Force has however been working a draft for a new protocol that builds on the older Asynchronous Ratcheting Tree (ART) and Tree Key Encapsulation Mechanism (TreeKEM) to build the new Messaging Layer Security as an open standard with strong security guarantees and efficiency for group messaging.

MLS is an ongoing effort to standardize an open standard for secure and scalable group messaging. The following sections discuss the various security properties and scalability requirements necessary for modern group messaging in addition to the MLS protocol and the associated architecture for implementing a group messaging application based on the MLS protocol.

3 Secure and Scalable Messaging

This chapter discusses the theoretical background for security, scalability and asynchronicity in the scope of messaging.

3.1 Security Properties

The following subsections discuss the key security properties that are required for secure and modern group messaging.

3.1.1 End-to-End Encryption (E2EE)

End-to-End Encryption (E2EE) refers to the scenario of messaging where only the sender and receiver of the message have access to the plaintext version of the message that is sent. Any observer of the channel that the message is sent through can read the encrypted ciphertext of the message but is unable to view the plaintext version.

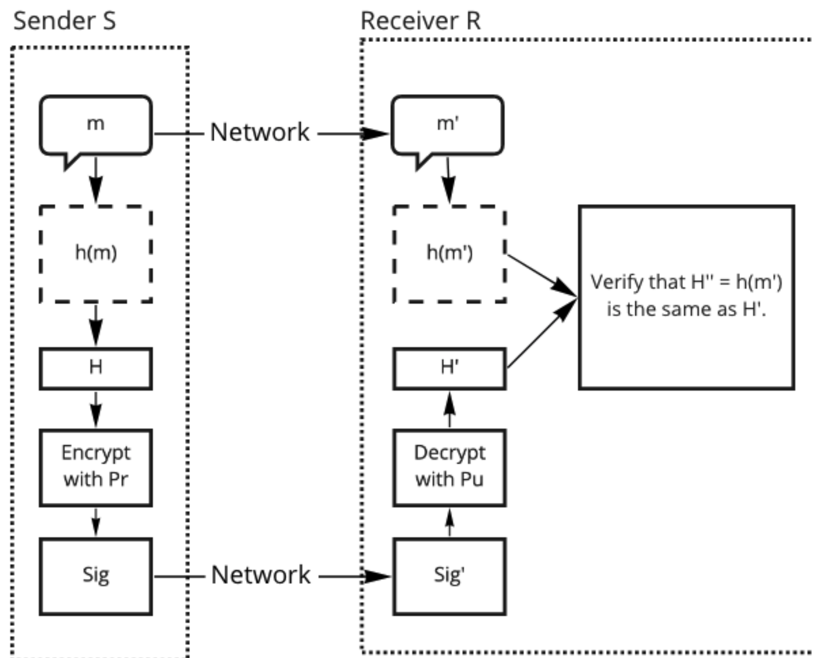
E2EE is an essential security property in secure messaging as the messages are not intended to be read by an attacker who may be able to compromise the channel where the message is transported through, and this may include the service provider of the messaging service. E2EE provides confidentiality guarantees such that any entity other than the sender and receiver have no information of the contents of the message.

3.1.2 Message Integrity

Message integrity is the security property that the receiver of a message can verify with certainty that the message was not tampered with in-transit. While it may not be possible to prevent a message to be tampered in-transit, it should be possible for the receiver to gain knowledge that the message was tampered with, and the message must not be considered to be the intended message sent by the sender [4].

To provide this security feature, it is possible to use digital signatures. Digital signatures work atop the concept of asymmetric key cryptography [5]. Figure 3.1 illustrates diagrammatically the process of using digital signatures to achieve message integrity.

Figure 3.1: Diagrammatic representation of digital signature for message integrity.



Digital signatures may be used as described in the following steps, given sender ***S*** and receiver ***R***:

- ***S*** creates a public-private encryption key pair for ***S*** such that ***S*** has the private key ***Pr*** and R has the public key ***Pu*** for ***S***.
- Generate a hash ***H*** such that ***H = h(m)*** where ***h*** is the hashing function and ***m*** is the message data.
- ***S*** encrypts ***H*** with ***Pr*** to generate signature ***Sig***.
- ***S*** sends ***m*** to ***R*** along with ***H***.
- ***R*** receives the message ***m'*** (as it may have been tampered with) and ***Sig'***.
- ***R*** decrypts ***Sig'*** using ***Pu*** to derive ***H'***.
- ***R*** derives ***H'' = h(m')***
- ***R*** verifies that ***H''*** is the same as ***H'*** which leads to the conclusion that the message was not tampered with. If ***H''*** and ***H'*** do not match ***R*** will learn that the message was tampered with.

The core working idea in this process is that only the sender ***S*** has possession of the private key ***Pr*** and that the decryption of H***,*** in order to modify it without the knowledge of ***Pr*** is computationally expensive.

It is essential in secure group messaging that the members are made aware of any attacker attempting to tamper with or impersonate a member. This also enables the group to react to a known compromise by rotating their keys and attaining Post Compromise Security (PCS).

3.1.3 Forward Secrecy (FS)

Formally, Forward Secrecy (FS) is defined as a security feature where the specific key agreement protocols give assurance that the session keys will not be compromised even if the private key of the server is compromised is taken into use [6].

From a messaging perspective, if an attacker has compromised the secret key and is able to access older messages, the messaging protocol should be able to heal the state, so the newer messages and keys are not accessible to the attacker.

In general, FS refers to the security of the past state of the system, relative to a compromise.

3.1.4 Post Compromise Security (PCS)

Post Compromise Security (PCS) is also known as future secrecy. PCS is the security property that assures the security of messages in the future after a compromise of the system [7].

PCS may be attained for example by introducing new entropy and thereby keys into the system so that newer messages are encrypted with the new keys and inaccessible to the adversary.

In general, PCS refers to the security of the future state of the system, relative to a compromise.

3.2 Scalability

In modern and enterprise group messaging, there is a need to include a large number of members in a group and therefore a group messaging service must be scalable. As an example, an organization may include a thousand members who must be part of a main group with all employees as participants. With pairwise secure messaging, this would not be scalable because a single message would need to be encrypted 999 times to encrypt it to each member. As the number of participants grows, the number of encryption and decryption operations required to be performed would linearly grow, adding to the computational and networking overhead of the system.

3.3 Asynchronicity

Users of a messaging service may possess several devices to participate in the messaging and the devices may not be online all the time. This presents the need for asynchronicity in modern group messaging such that the members can participate in the group messaging without being online always and that the group operations can tolerate and progress with such network partitions.

4 MLS Protocol

Messaging Layer Security (MLS) is a security layer being built by the MLS working group for the purpose of secure and scalable End-to-End Encrypted (E2EE) group messaging. MLS is being built in a process similar to the model followed by Transport Layer Security (TLS) 1.3 [2].

4.1 Motivation for MLS

The primary motivation for the MLS protocol was the lack of an open standard for E2EE particularly in the group messaging domain. In 2016, Wire opened discussions around this topic and thus the protocol was born. Several other contributors from both the academic and the industrial domains, such as, University of Oxford, Facebook, INRIA, MIT, Google and Twitter joined this project eventually.

More specifically, the goal of MLS is to make E2EE messaging in large groups scalable and efficient with sublinear complexities while providing formal security guarantees such as FS and PCS [8].

4.2 Current State of MLS

MLS is a Internet-Draft at the time of writing. An Internet-Draft is a working document of the IETF. The current Internet-Draft will expire on the 23rd of September 2022 [9]. At the time of writing the MLS protocol is draft version 13 with the version 14 yet to be published. The MLS working group recommends against the use of MLS in any production systems at this time [2]. The reason behind this recommendation is that MLS is still being discussed, analyzed and built.

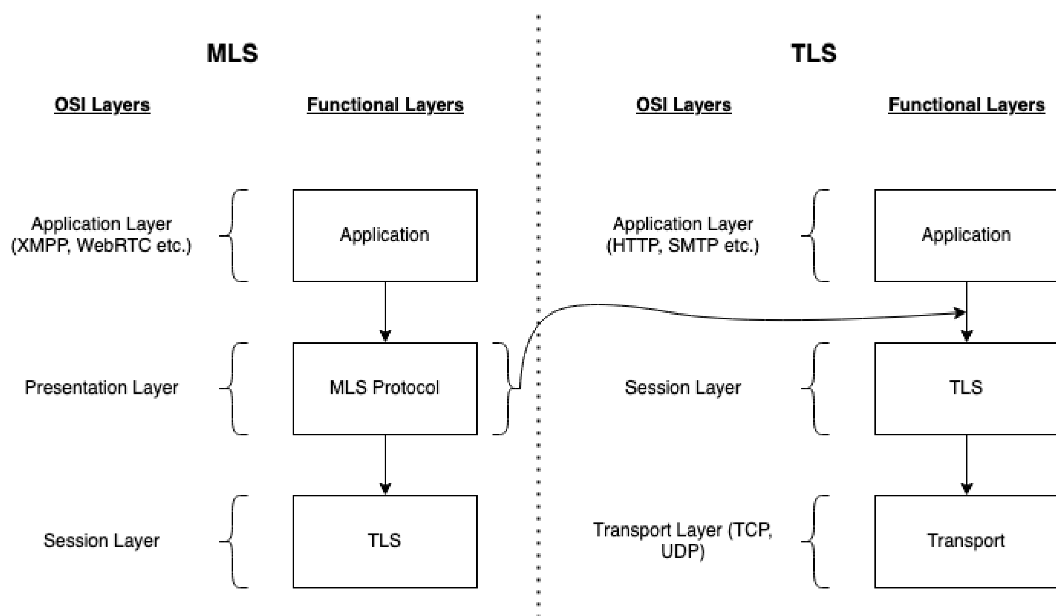
There are some MLS implementations including ones from industrial organizations. Cisco has created an open source implementation of the MLS protocol [10]. This implementation is written in C++. OpenMLS is another open source implementation of the MLS protocol written in Rust [11].

4.3 Scope of MLS

The scope of MLS is only to be a security layer between the transport level and the application level, thereby providing an E2EE channel for group communication. It is supposed to work on top of an underlying layer of transport security and encryption, such as TLS. MLS does not include any specifications for authentication, although it is compatible to work with any external authentication service implementations.

There is a parallel between the scope of MLS and TLS where TLS works atop underlying protocols, namely Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) [12]. Figure 4.1 illustrates the scope of MLS and TLS comparatively.

Figure 4.1: Comparison of the scope of MLS and TLS.



MLS protocol works on the presentation layer sandwiched between the application layer and session layer or TLS. TLS sits between the application layer and transport layer. In an application using MLS, MLS operates between the application and TLS.

4.4 Operation of MLS Protocol

The MLS protocol is designed to be implemented in an architecture as described in section 5. Shortly, a Delivery Service (DS) that facilitates a directory service for

publishing and retrieval of initial cryptographic key material, broadcast of MLS and application messages and unicast delivery of welcome messages for new members of a group is required for MLS operation. Additionally, an Authentication Service (AS) to enable authentication of the users of the MLS application is necessary. The specifics of the DS and AS are discussed further in the subsections 5.2.1 and 5.2.2 respectively.

The following subsections take a dive into the MLS application and operative objects and data structures.

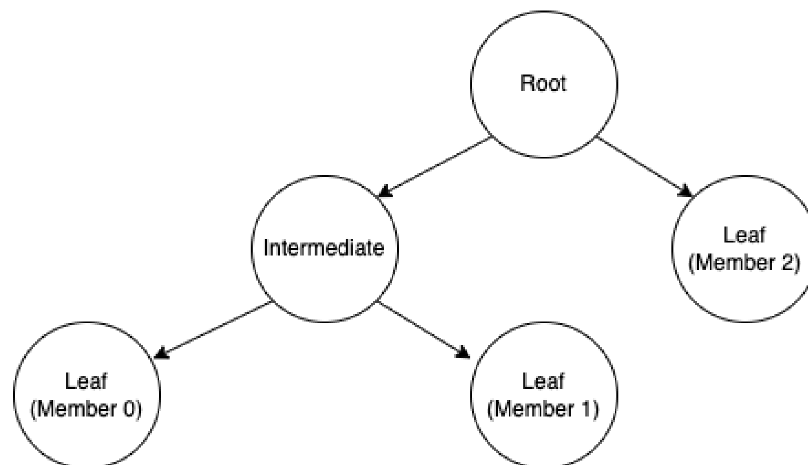
4.4.1 Ratchet Tree

The MLS protocol uses ratchet trees for creating shared secrets within a group [13]. The ratchet tree used in MLS is based on the earlier protocol Asynchronous Ratcheting Trees (ART) which is in turn an extension of the double ratchet algorithm which has been well established and studied and an accepted standard for two party E2EE communication [14]. ART is an extension of ratchet tree and Diffie-Hellman tree and is the first protocol that included the guarantees of FS and PCS security properties [15].

A ratchet tree within MLS is an arrangement of secrets such that they can be efficiently updated to reflect changes in the group state. Ratchet trees also allow efficiently removing members from a group by introducing new entropy to the subsequent group state to achieve the PCS and FS security properties. The core idea of a ratchet tree lies in the detail that there are shared keys assigned to a certain subgroup within the group and not every pairwise permutation of the group members. This allows for a computational complexity of O(log(N)) for one member encrypting a message to be sent to every other member with the group instead of the naive complexity of O(N).

Figure 4.2 is an example illustration of what an MLS ratchet tree might look like with a group comprising 3 members.

Figure 4.2: Example ratchet tree of a group with 3 members.



Leaf nodes: Contain secrets for the members they belong to.

Intermediate nodes: Contain shared secret for members under them.

Root: Contains group secret.

An MLS ratchet tree is a left balanced tree where the leaf nodes correspond to members of a group. The root node in the tree contains the group secret. The intermediate nodes contain the shared secrets for the subgroups formed by the members under them.

Every client in a group must eventually maintain a complete and up-to-date view of the public state of the group which includes the public keys for all the nodes and credentials for each member, i.e., leaf nodes. A private key for a node, however, is only known to a member if their leaf node falls within the subtree of that node [16]. The converse of this is not true though. Figure y depicts an example situation with the view of private keys from the point of view of the members of the group.

4.4.2 KeyPackage

A KeyPackage is an object in the MLS protocol that holds data that is published by a user to the directory service with the DS that allows for other users to add them to a group by authenticating them and reading the capabilities of their client.

The KeyPackage object contains the following information with respect to a given client:

- The MLS Protocol version and the ciphersuite that the client supports.

- Public key that other clients can encrypt a Welcome message to this client when joining a group. This is also known as an InitKey.
- Initial keyring material that must be added to the contents of a leaf node that the client joints.

A KeyPackage may be signed with the public key that it contains so that the other clients downloading it may verify the authenticity of the KeyPackage.

The KeyPackage object may also optionally include a vector that contains the extensions supported by the client.

4.4.3 Epoch and Message Ordering

An Epoch refers to a point in time. In relation to the MLS protocol, it refers to a sort of version of the group state such that using the epoch it is possible to order group MLS messages lexicographically.

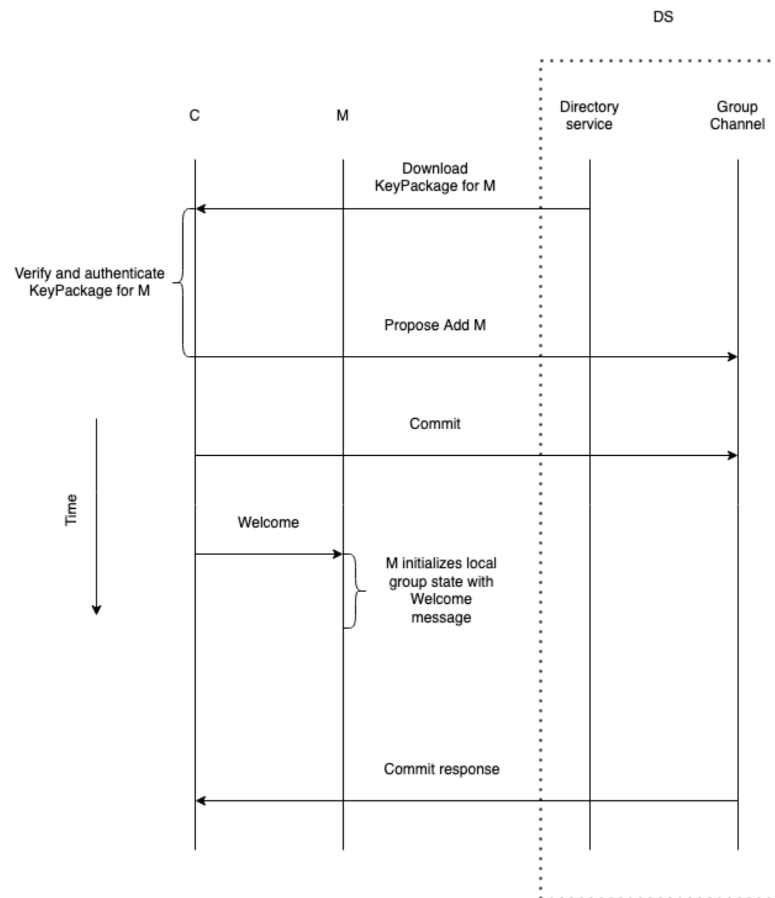
The MLS protocol requires that the Commit MLS messages, discussed further in subsection 4.4.6, are applied in a consistent order across all clients within the group so that every client maintains the same view of the ratchet tree and thereby the group state [17]. The MLS messages may be delivered in order by a DS such that all the clients receive the messages in a deterministic and consistent order based on the group epoch or some other point of information that can be used as an anchor to represent an epoch. Another potential solution would be that the clients are able to decide on an ordering based on some group consensus algorithm.

4.4.4 Group Creation

A single user may create a group and the creator of the group may add other members to the group with Add proposal MLS messages. A group may consist of a minimum of two members and there is no theoretical upper bound on the group size. The user who creates the group initializes the group state and then invite at least one additional member to create a group.

Figure 4.3 shows visually the process of creating a group and inviting one other member to the group.

Figure 4.3: Visual representation of group creation with invitation to one member.



As an illustrative example, when the creator ***C*** adds a new member ***M***, they perform the following actions (also shown visually in Figure 4.3):

- ***C*** downloads the KeyPackage for ***M*** from the directory service in the DS.
- ***C*** authenticates and verifies the KeyPackage.
- ***C*** sends an Add proposal to the Group Channel.
- ***C*** sends a Commit message to the Group Channel.
- ***C*** sends a Welcome message to ***M***. Note that this is supposed to be a unicast message, but it is still possible to send this as an E2EE encrypted message to ***M*** through the DS, as the KeyPackage contains the public key of ***M*** and the server does not contain the private key to decrypt it. The Welcome message is used by ***M*** to initialize their copy of the group state.

- ***C*** waits for response to the Commit message from the Group Channel before creating any new proposal or Commit messages.

4.4.5 Group Proposal Operations

In order to advance the group state, essentially, to add, update and remove members from a group any member of the group may send a corresponding proposal message to the group channel and thereby the entire group, followed by a Commit message to confirm the action and modify the group state.

By default, any current member of the group may send a proposal message and following Commit message to the group and modify the group state. However, the application implementing the MLS protocol may choose to limit the access to a subset of the members based on some Access Control Lists (ACLs)

The Commit message is discussed further in the subsection 4.4.6 and in the following subsections, some of the key group proposal operations and messages are discussed.

Add

An Add proposal contains the KeyPackage for a user that is to be added to the group that the proposal is sent to. If applied successfully with a Commit message following, the user that the KeyPackage is for, is added to the group.

Update

An Update proposal message contains the new contents of the leaf node that must replace the current contents of the leaf node corresponding to the member that is to be updated. The Update proposal must be followed by an eventual Commit message to modify the group state. An Update proposal is generally used to rotate the secrets and credentials for a given member which enables PCS.

Remove

A Remove proposal message is used to remove a member from a group. The proposal message contains the reference to the leaf node of the member that must be removed. A Remove message must be followed by a Commit message for the proposal to modify the group state. As the Commit message introduces new entropy into the group, this enables PCS as the member that is removed, if compromised, is unable to decrypt any new messages in the group relative to the epoch in which the member was removed.

4.4.6 Group Commit

The Group Commit message has a key function in advancing the group state. It is a message that instructs the members of the group to modify their copy of the group state in accordance with the uncommitted proposal messages that were sent prior to the Commit message [18]. The Commit message starts a new Epoch in the history of the group state. The Commit message also introduces new entropy into the group enabling the PCS security guarantee [19]. This is a result of the new entropy introduced in the group state in the new Epoch created by the Commit message which generates new group and shared secrets amongst the group members.

The members that removed, for instance, are unable to read messages following their removal as they do not have access to the new group secret. Similarly, FS guarantee is provided such that the new group secrets generated and they cannot be used to decrypt messages older than the ones sent in an old Epoch.

It is also possible to send a so-called, empty Commit message, to simply introduce new entropy and to update a member’s secrets and credentials for the purpose of providing PCS.

It is essential that the Commit messages are processed in a consistent order by each member of a group such that each member maintains a consistent view of the group state and thereby the group secrets and credentials.

4.4.7 Group Application Messages

Application messages are any conversational messages that the members may use to communicate with each other. These messages do not provide any MLS functionality other than the possibility for the members to communicate with each other. The contents of the application message must conform to the MLSCiphertext so that they are E2EE, however the message contents within the ciphertext may be arbitrary. The message contents are essentially dictated by the corresponding application layer for example in an Extensible Messaging and Presence Protocol (XMPP) based text messaging application, the contents may simply be textual conversational data or in a media-based communication such as an application using WebRTC (Web Real-Time Communication) for example, may be some video data.

5 MLS Architecture

The MLS protocol aims to provide a basis for a service implementing it the ability to provide secure and scalable messaging within groups for a set of users. In order to understand the various architectural patterns that are possible to implement the MLS protocol as a Service Provider (SP) it is important to recognize the definitions and the general setting prescribed by the MLS architecture RFC document.

5.1 Components

The main entities in the MLS architecture include the groups, the clients, the SP which comprises the Authentication Service (AS) and the Delivery Service (DS).

5.1.1 Group

A group is a set of clients that may use a set of devices to communicate with each other using the SP. The minimum size for a group is two and it may be as large as possible within the limits of practicality.

From a more formal perspective, a group in MLS is defined as a set of clients that know the shared group secret in the group key establishment phase of the protocol and made some contributions to it.

A client cannot be considered a member of a group unless the client has been added to the group and contributed to the group secret such that it has been verified.

5.1.2 User

A user is an entity, for example a physical person that might have one or more devices and uses the messaging service provided by the SP.

5.1.3 Client

The MLS architecture document defines a client as a set of cryptographic objects composed of public values such as a name (an identity), a public encryption key and a public signature key.

A client is considered to be associated with a given user if the user has access to the secret values of the client.

5.1.4 Member

A member of a group is a client that is a part of it. MLS requires that the given client must have been added to the group and have contributed to the group secret such that all other members of the group have verified it.

5.1.5 Service Provider (SP)

In the context of providing group messaging using the MLS protocol, an SP is the entity that allows for users to interact with itself, in turn enabling group messaging. According to the MLS Architecture document, the SP provides two abstract functionalities, namely the Authentication Service (AS) and the Delivery Service (DS).

These functionalities while being logically separate do not necessarily need to be split up in such a manner. For example, the functionalities may be provided by the same server.

Additionally, neither of the functionalities of the AS and DS to be implemented as a centralized server. It may be that the MLS clients are able to generate, redistribute and validate their credentials without the requirement for a centralized AS.

Similarly, the MLS clients are able to distribute their credentials pertaining to messaging and the group messages without the need for a centralized DS.

Operation independent of an external party AS and DS might be a requirement of an application, for example in applications where privacy is of high concern and therefore there is a need for a decentralized messaging system. In this sense, the SP is an optional component.

The purpose of the AS includes holding and verifying a mapping between application-specific identifiers and cryptographic material, namely, the public key material used for authentication within the MLS protocol. Furthermore, the AS must have capability to create credentials that encode these mappings and validate credentials provided by some client held by a user of the messaging service.

The DS provides the functionality of receiving and distributing messages between group members within the MLS messaging service. The DS may for example act as a broadcaster, such that a message sent by one member within the group is received by the DS and then broadcast to each recipient within the group. Furthermore, the DS is responsible for storing and delivering initial public key material that is used by MLS clients associated with their respective members to create the group secret key.

5.2 Functionalities of the Service Provider

The Service Provider (SP) is an entity that provides functionalities that implement the MLS protocol so that it offers an MLS group messaging service. MLS protocol is defined such that the clients are able to operate without an SP, in a decentralized setting.

Although, it is simple to envision an architecture with an SP, for example a SaaS application in conjunction with client applications that would be installed on devices such as laptops and smartphones of users participating in group messaging. Such an application might be used for personal messaging or even group messaging for organizational purposes where a group might be composed of employees within an organization or a department within an organization.

Regardless of a given MLS application being centralized or otherwise, the functionality that as outlined by architecture requirements of the MLS protocol, the two logically separate services, namely the AS and DS must be implemented either within the SP or as some component with the MLS clients.

In the case of a centralized setting, the AS and DS are implemented within the SP, in some arbitrary or no partition between them so long as the logically separate functionalities are available to the MLS clients [20].

The following subsections discuss in further detail, the functional requirements of the AS and DS.

5.2.1 Authentication Service

The MLS architecture document stipulates that the AS must provide two key functionalities to implement itself as an MLS AS:

- Create cryptographic credentials that encode a mapping between the identity used by the application and the signature key pairs for verification purposes.
- Enable a user to verify that the credentials used by another user is valid and legitimately issued by the AS.

An MLS client is able to authenticate MLS messages by signing them with a private key associated with its signing key pair and another client is able to verify this by checking it against the public key associated with the same signing key pair [21].

As per the MLS architecture specifications, the protocol is designed so that the AS could be considered an abstract component [21]. This implies that a part of the AS could be implemented on the MLS client end as well. For example, a Public Key Infrastructure (PKI) could be used to implement the AS functionality wherein the issuance of credentials is provided by the Certificate Authorities (CAs) in the PKI and the verification is performed by the clients.

It is possible and several existing messaging applications use the pattern of users verifying each other’s key fingerprints for the purpose of authentication. In such a case, there is no exclusive issuance by an external CA, instead a key pair (public key and private key with an asymmetric encryption strategy) is generated. The verification of the credentials in this scenario is implemented as client-side application functionality.

5.2.2 Delivery Service

The MLS architecture document outlines multiple functionalities that a DS must implement. Some important functionalities include acting as a directory service and routing MLS messages to the users via the MLS clients [22].

Directory service functionality provided by the DS helps with the discovery of clients and associated users. Furthermore, the directory service functionality is required for establishing shared group keys that enable group operations such as sending messages to other clients.

It is important to consider that the level of trust given by the users and from an MLS perspective, the groups, the specific details of the DS implementation may vary alongside the privacy guarantees. Regardless of the privacy guarantees, MLS mandates that the authentication and confidentiality guarantees must be met in order to be correctly implemented as an MLS DS.

A DS must be considered to be untrustworthy in its own accord in regard to authentication and secrecy per the MLS architecture document. The AS must be trusted for these aspects of the application.

The following subsections dive further into some important functionalities of the DS.

Key Management

Key management includes the storing and distribution of some public information related to a client. Such information includes the initial cryptographic key material known as the KeyPackage by MLS. The KeyPackage includes the functional abilities of a client namely, the MLS protocol version and the protocol extensions used, if any.

In addition to this, the KeyPackage contains cryptographic data which comprises a credential granted by the AS marking the mapping between a client's signing key and the client's identity within the MLS service. Additionally, it holds the key corresponding to the credential and a client's asymmetric public key material. The KeyPackage must be signed with the signing private key corresponding to the KeyPackage [23].

The DS must allow the users to add, remove and update their initial key material. This functionality is required to maintain FS and PCS.

Message Delivery

A very crucial function of the DS is to deliver messages between clients. The messages include both the MLS messages to support the protocol and its functions and the messages that the users send each other.

Messages may be required to be sent to a subset of the members of a group, for instance MLS messages to add a new member to a group or to all members of a group in case of a conversational message to everyone within the group. The DS may choose to use different patterns to facilitate this with a server fanout (broadcast) or by delegating it to the clients to implement some client fanout. The pattern chosen may vary across specific applications and functionalities across various MLS services as may be deemed practical for said implementation.

Message Ordering

The history of a group is linear in the MLS protocol. This presents a requirement that operations that add to and update the group state must be delivered and applied in order by all parties involved so that all parties maintain a consistent state of the group. In particular, this implies that the MLS Commit messages must be delivered by the DS and applied by all parties, in order or in collectively agreed upon order. Apart from this requirement, it is not required by MLS that other types of messages are delivered in any particular order [24].

A DS may assist in consistent ordering of the MLS Commit messages for the clients or in a case where the DS does not provide such functionality, the clients could implement some consensus strategy to agree upon consistent ordering.

An example of a DS implementing an ordering strategy could involve using a monotonically increasing epoch such that there is only one valid Commit message

associated to a given epoch and the DS is programmed to maintain the correct state by the virtue of order or messages sent to it and invalidate any incorrect orderings.

A DS may also be designed so that it always sends the Commit messages and does so in a way that they are ordered in the same way for each client.

The MLS protocol provides three key points of data within the MLSCiphertext to provide ordering:

- A group identifier to distinguish between distinct groups as the ordering is only relevant within a group.
- An epoch number that represents the state of progression in the group, more precisely, the number of changes or in other words the version of the group state associated with the group that the message is intended for.
- The content type of the message which allows a reader of the message, for example a DS to determine the requirement of ordering on the message type. This is particularly useful to distinguish Commit messages from other message types.

The MLS protocol itself supports the verification and application of these properties. This has the effect that even though a DS delivers messages in inconsistent order, MLS clients may be able to put the messages in order themselves.

Membership

The DS provides a directory service which includes maintaining and providing information about the membership of clients and their association to the users. Despite this, it is important to consider that group membership is considered to be sensitive information within MLS. MLS is designed to limit the amount of metadata that needs to be persistent to maintain group membership knowledge.

In the case server fanout is required, it is required by the server to store data pertaining to group membership. The MLS architecture document mentions that in such a case, this metadata is stored encrypted at rest. Additionally, it recommends that applications should consider using systems designed for anonymous server fanout such as Loopix

[25]. This pattern, though, exposes the risk to privacy that a DS can learn about client messaging patterns even though it is not a breach of the authentication or confidentiality guarantees provided by the MLS protocol.

It is also possible to use client fanout for message delivery where the DS does not need to store metadata pertaining to group membership. The MLS architecture document notes however that, there is risk to privacy in case an adversary is able to analyze and decipher messaging patterns by analyzing the network traffic. It may still be possible to thwart such attempts by introducing padding to messages and scrambled traffic such that the real nature of messages is not revealed to an external observer of the network channels involved.

5.3 Membership Changes

It is not hard to envision scenarios where group state needs to be changed for usability purposes. New members might join a group and existing members may update their own state which requires it to be reflected in the group state and members might be removed.

The access control for such operations is not a part of the specification of the MLS protocol or the architecture. It is left up to the applications implementing an MLS service to enforce such access control limitations over their users [26]. This is discussed further in section 5.5.

MLS protocol does not enforce any remove or update access limitations even when clients installed on multiple devices of the same user are concerned. It may be quite important in some applications that only the user owning the clients is able to update and remove said clients from a group or that only a specific user assigned to take the role of an administrator is able to perform such actions. This sort of functionality must be implemented by the application itself outside of the MLS protocol. This aspect is considered further in section 5.6.

It is important to note that given the FS and PCS guarantees offered by the MLS protocol that within the scope of the protocol members joining a group only have access to messages while they have become a member and continue to be as such. Delivery of older messages for instance must be implemented outside the MLS protocol by the

application, however this comes with a tradeoff made to the FS and PCS guarantee offered by the application.

5.4 Asynchronicity

One of the key reasons and functionalities of the MLS protocol is to enable asynchronous messaging. The key implication of this is that no two distinct clients need to be online simultaneously to participate in messaging. Members participating in conversations can update the group state and send messages without waiting for the reply of the others. It is the responsibility of the application however, to provide the transport layer to facilitate the delivery of messages asynchronously and reliably from a functional perspective.

5.5 Access Control

The MLS protocol does not enforce any sort of access control within the protocol. This implies that each member in a group is able to perform all MLS operations with equal privilege. This is a consequence of all members’ clients possessing access to the same cryptographic material.

Should some access control limitations be required for a certain application it is the responsibility of the application to maintain the metadata pertaining to this and enforce it on the application level.

5.6 Multi-device Support

It is easy to envision that a given user may own more than one device and use multiple to participate in a group. A new device associated with a given user may always be added as a new client to the group. It must be noted however that as a new client, the device will only have access to new messages from the point of joining within the scope of the MLS protocol. The application may however implement some other mechanism at the application layer to provide older messages at the cost of trading off the security guarantees that MLS provides.

5.7 Additional Considerations

The MLS architecture document considers a few other functionalities that may be implemented by an MLS application.

The MLS protocol provides several points for extensions to the protocol where additional data can be associated [27]. For instance, it is possible to extend the KeyPackage to advertise capabilities of the clients.

A single user may have membership to several groups within the same MLS service. The MLS protocol does not prevent this and the FS and PCS guarantees provided by the MLS protocol hold valid within each individual group. Operations performed in one group are isolated from any operations and changes to the state in another group.

The MLS protocol also aims to be compatible with federated environments. Such applications could be quite useful in situations where there is a need for inter-organizational conversations for example [28].

It is also important to consider that the subsequent versions of MLS are able to coexist as they are released and taken into use. The MLS protocol offers a possibility for negotiation mechanisms for this purpose.

6 Practical exploration with OpenMLS

OpenMLS is an implementation of the MLS protocol. It is written in the Rust programming language. The basis for the project is the MLS draft 12 and above.

OpenMLS provides a high-level API to do group operations such as creating and updating it.

OpenMLS does not implement any cryptographic primitives on its own but uses separate libraries that implement the cryptographic primitives of the MLS protocol.

This thesis proceeds to use OpenMLS to answer the business question of providing a practical entry point into implementing an MLS group messaging service. OpenMLS was chosen for this purpose because of its active development and conformity to one of the more recent drafts of the MLS protocol, namely the draft 12, at the time of writing. Additionally, OpenMLS includes a build-in DS and client application [11].

6.1 Rust

Rust is an open source, general purpose programming language. It is intended for systems programming [29]. Rust provides strong type safety in addition to strong memory safety. Rust is a compiled language and like the lower-level language C, the memory management is not garbage collected, whereas it is managed at compile time. Thus, it follows that Rust produces very performant applications [30].

Rust is a good choice for implementing a protocol intended to be a performant layer under the user facing application layer. Additionally, the type and memory safety properties provided by Rust only add to the security of an implementation of the MLS protocol.

A few tools come along with an installation of Rust, namely, cargo, rustc and rustup. Cargo is a package manager and is used for installation of external dependencies a rust

project may have. Rustc is the Rust compiler which takes the sources code as input and produces a binary that can be executed on a target system architecture. Rustup is a Rust version manager that allows to update the Rust toolkit, including cargo and rustc. Some Rust codebases use features not available in the stable channel. These features are available in nightly releases. Rustup is also useful for managing different versions of nightly releases of Rust on a system which may even exist parallel to multiple stable versions.

6.2 MLS Delivery Service (DS)

OpenMLS has an implementation of an MLS Delivery Service (DS) intended for testing purposes. Within the repository it is under the /delivery-service directory. This directory contains the server implementation under the /ds directory and the corresponding library under the /ds-lib directory.

The DS server implements some HTTP routes that facilitate the following function (Table 6.1):

Table 6.1: List of methods implemented by OpenMLS DS.

Function	**HTTP Method**	**Route**
Register clients	POST	/clients/register
Listing clients	GET	/clients/list
List of key packages	GET	/clients/get/{client_name}
Send message to group	POST	/send/message
Send welcome message	POST	/send/welcome
Retrieve messages for client	GET	/recv/{client_name}

To run the OpenMLS DS it is a prerequisite to have the latest version of Rust and the associated tooling installed. Namely, rustc, rustup and cargo.

To run the OpenMLS DS, first, the current directory must be /delivery-service/ds and then cargo build must be executed to build the executable from the source code. An example of the operation of the DS is shown in figure 6.1.

Figure 6.1: Example log output of the OpenMLS DS.

```
> RUST_LOG=DEBUG cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/mls-ds`
 INFO  mls_ds > Listening on: 127.0.0.1:8080
 INFO  actix_server::builder > Starting 12 workers
 INFO  actix_server::builder > Starting "actix-web-service-127.0.0.1:8080" service on 127.0.0.1:8080
 DEBUG mls_ds                > Storing group message: GroupMessage { msg: MlsMessageIn { mls_message:
Ciphertext(MlsCiphertext { wire_format: MlsCiphertext, group_id: GroupId { value: TlsByteVecU8 { vec:
[103, 114, 111, 117, 112, 48] } }, epoch: GroupEpoch(1), content_type: Application, authenticated_data
: TlsByteVecU32 { vec: [103, 114, 111, 117, 112, 48, 32, 65, 65, 68] }, encrypted_sender_data: TlsByte
VecU8 { vec: [149, 37, 158, 224, 121, 69, 77, 119, 57, 200, 11, 155, 55, 82, 36, 132, 130, 41, 94, 155
, 26, 238, 97, 45, 117, 23, 104, 146, 3, 250, 234, 68, 100, 200, 206, 73, 87, 109, 72, 193] }, ciphert
ext: TlsByteVecU32 { vec: [229, 154, 195, 90, 53, 136, 111, 128, 247, 27, 110, 181, 238, 70, 80, 96, 3
2, 70, 32, 129, 119, 123, 133, 209, 137, 30, 194, 31, 128, 37, 111, 204, 229, 82, 204, 246, 233, 179,
251, 193, 204, 61, 7, 194, 20, 81, 15, 194, 207, 95, 53, 135, 14, 111, 109, 66, 3, 137, 243, 108, 245,
 60, 79, 178, 8, 9, 110, 41, 178, 17, 162, 107, 140, 219, 35, 35, 164, 189, 154, 21, 124, 173, 213, 82
, 221, 11, 102, 124, 119, 146, 120, 224, 85, 93] } }) }, recipients: TlsVecU32 { vec: [TlsByteVecU32 {
 vec: [117, 115, 101, 114, 49] }] } }
 DEBUG mls_ds                > Resetting server
 DEBUG mls_ds                > Registering client: ClientInfo { client_name: "user0", key_packages: Cl
ientKeyPackages(TlsVecU32 { vec: [(TlsByteVecU8 { vec: [27, 218, 205, 19, 96, 39, 189, 167, 42, 66, 32
, 31, 14, 52, 205, 219] }, KeyPackage { payload: KeyPackagePayload { protocol_version: Mls10, ciphersu
ite: MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519, hpke_init_key: HpkePublicKey { value: [126, 98, 150
, 148, 243, 171, 122, 6, 184, 116, 251, 140, 238, 221, 18, 218, 82, 158, 17, 90, 48, 61, 68, 131, 138,
 32, 55, 42, 167, 197, 197, 51] }, credential: Credential { credential_type: Basic, credential: Basic(
BasicCredential { identity: TlsByteVecU16 { vec: [117, 115, 101, 114, 48] }, signature_scheme: ED25519
, public_key: SignaturePublicKey { signature_scheme: ED25519, value: [217, 140, 80, 82, 181, 255, 62,
```

On completion of that, executing cargo run will start the DS. One may additionally add the RUST_LOG=DEBUG environment variable to view the debug level logs to understand the operation of the DS. When the DS is executed with debug logging enabled, the request operations such as storing of a group message and registration of a client are logged.

6.3 OpenMLS Command Line Interface (CLI)

OpenMLS provides a Command Line Interface (CLI) client to act as an interface for a user of the MLS service provided by the DS. It is possible to perform a key subset of operations supported by the MLS protocol in order to perform group messaging. In the OpenMLS repository the CLI can be found within the /cli directory. Figure 6.2 shows the various client-side operations supported by the OpenMLS CLI.

Figure 6.2: Screenshot of MLS client operations supported by the OpenMLS CLI.

```
Type help to get a list of commands

help

>>> Available commands:
>>>      - update                                   update the client state
>>>      - reset                                    reset the server
>>>      - register {client name}                   register a new client
>>>      - create group {group name}                create a new group
>>>      - group {group name}                       group operations
>>>          - send {message}                       send message to group
>>>          - invite {client name}                 invite a user to the group
>>>          - read                                 read messages sent to the group (max 100)
>>>          - update                               update the client state
```

The CLI allows resetting the server state, updating the client with the latest server state, registering as a new client. Additionally, it supports simple group operations like sending messages, inviting new members and reading new messages.

To start the CLI, the working directory must be `/cli` and `cargo run` must be run. It is important to note that the CLI communicates with the OpenMLS DS and therefore the DS must be running for the CLI to function properly.

For the ease and partially automating the testing, it is also possible to use the private API provided by the CLI or even implement the tests as unit tests. In this thesis, the private API of the CLI is used within unit tests to perform the evaluation of MLS and OpenMLS.

6.4 Analysis of Group Messaging with OpenMLS

This section uses the API provided by the OpenMLS CLI and DS to evaluate some usage and scalability scenarios for this MLS implementation.

It is important to note that while it is not a restriction of the MLS protocol, the OpenMLS DS and CLI are written to be prototypes and therefore possess limitations. One limitation is that the number of members and messages held by the DS cannot exceed a certain number because of the underlying data structures not supporting a larger size. Figure 6.3 displays screenshot of the DS logs pertaining to one such issue.

Figure 6.3: Screenshot of a size limitation with OpenMLS DS.

```
thread 'actix-rt:worker:8' panicked at 'Vector length can't be encoded in the vector length a 1858798 >= 65535',
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

However, for the purpose of this thesis, it only limits the scalability tests from exceeding certain limits. This does not undermine the analysis apart from testing against very large group sizes and number of messages

The appendix provides the code for the tests used in this section.

6.4.1 Basic Group Operations

This section evaluates basic operations in an MLS service such as creating a group, inviting members, and sending messages.

Creating a Group of Two Members

A test is run by adding two clients. One of the clients creates a group and adds the other client to the group. The second client sends a message to the group. Figure 6.4 shows the screenshot of the test result.

Figure 6.4: Screenshot of a test result of two-member group with OpenMLS.

```
running 1 test
test test_two_member_group ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.06s
```

The test verified that the first client received the message.

Adding a Member

This subsection tests adding a member to a group. Specifically, it tests that the newly added member can read new messages sent to the group after the member joined but cannot read the messages sent before the member was added.

This test verified that the first client could read the old and the new message whereas the newly added member could only read the message after the member was added.

6.4.2 Scaling Test

In this subsection, scaling by number of messages sent and number of group members is tested. Figure 6.5 shows a screenshot of a sample of the output of a scaling test run.

Figure 6.5: Screenshot of scaling test output.

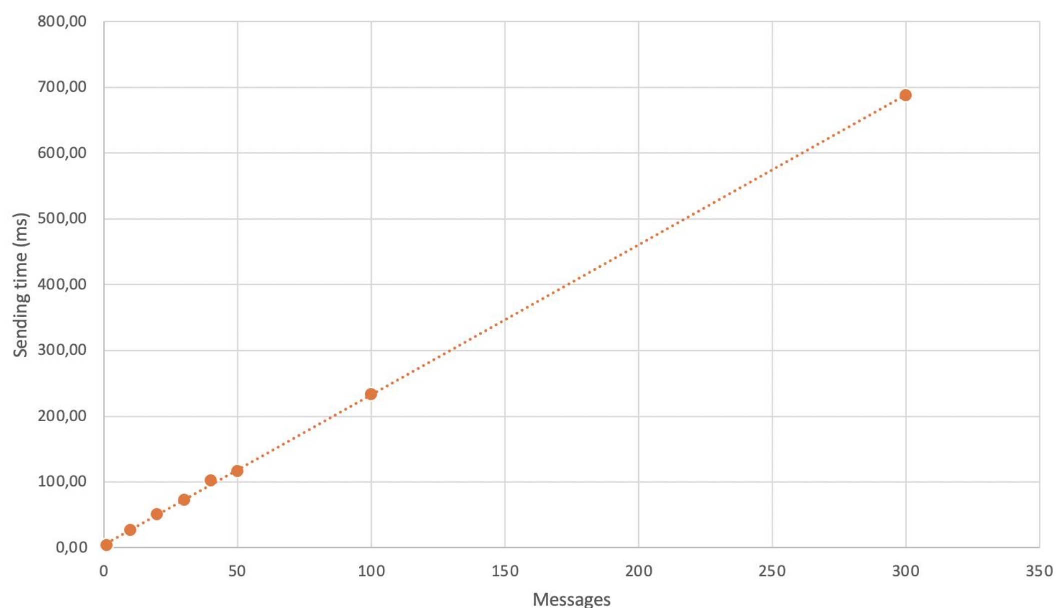
```
running 1 test
Inviting 25 members took: 2.90s
Sending 10 messages took: 24.47ms
Reading 10 messages took: 339.45ms
test statistics_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 3.48s
```

The test result includes timings for inviting a given number of members, the time taken to send a given number of messages and the time taken for clients to read them.

Figure 6.6 shows a graph of the time taken to send a scaled number of messages by the number of messages sent to a group of 10 members.

Figure 6.6: Graph of time to send messages by number of messages.

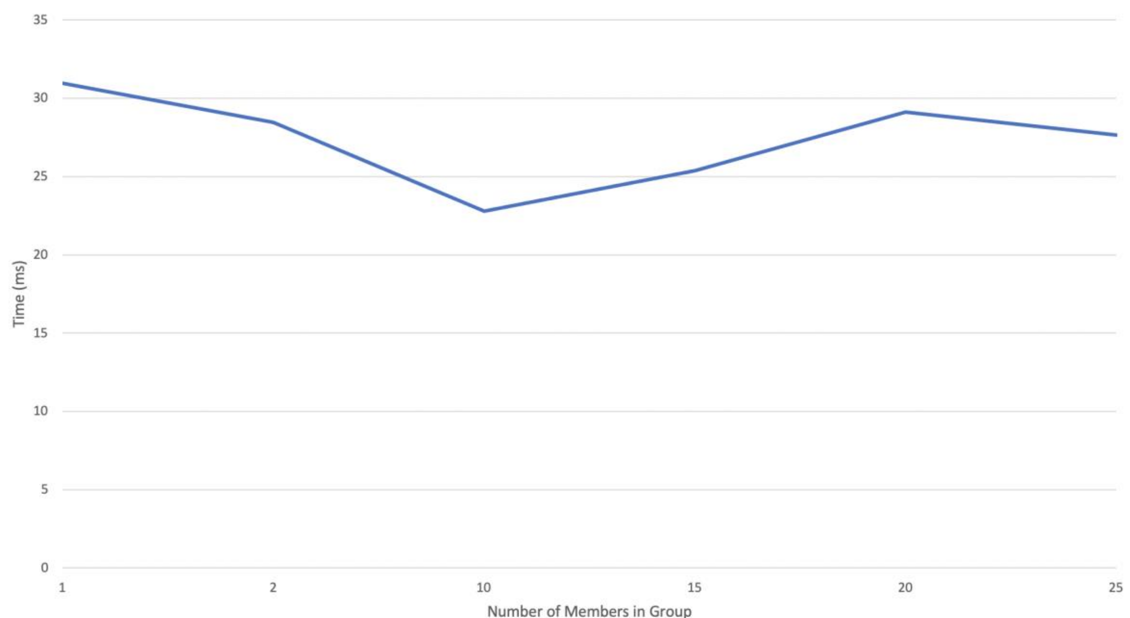


The time to send a scaled number of messages given N members is tested. It is observed that this is metric scales linearly by the number of messages. The trend is linear by the number of messages.

This result verifies that the time complexity of sending a message in MLS is O(1) as the message is only encrypted and sent to the DS once. This is unlike server fanout, where the same message must be encrypted and sent to the server N times given N clients, resulting in a time complexity of O(N).

Figure 6.7 shows the time taken to send 10 messages to a group of varying sizes.

Figure 6.7: Graph of time to send 10 messages by number of members.



Additionally, it is observed that the time taken to send 10 messages to group with increasing sizes is around the same regardless of the group size. This further concludes that the time complexity of sending a message in MLS is O(1). In other words, the protocol scales very well regardless of the group size as it only takes as long as the underlying system takes to send one message to one member.

6.5 Analysis of the Results

In this chapter, MLS protocol was evaluated with the use of OpenMLS. Basic group operations were tested, and the tests verified creation of a group and addition of members. Additionally, scaling tests confirmed the promised time complexity of MLS sending messages in a group is O(1). Or in other words, the size of the group does not affect the time taken to send messages to the entire group.

The results conclude that MLS is a scalable group messaging layer and provide OpenMLS as an entry point for implementing the MLS protocol into a group messaging application that requires scalable and secure messaging.

7 Conclusions

This thesis explored the shortcomings of the currently available group messaging systems. Pairwise secure communication was viewed as a well understood and solved matter. However, this thesis delves into the unsolved domain of scalable and secure group messaging.

While some systems that offered secure or scalable group messaging were noted, it is often a tradeoff between the two in the current scenario. In some applications, the security would be undermined, in the sense of losing some of the modern security properties that modern messaging often requires. On the other hand, some applications would require limits on the number of members or not be as performant to guarantee modern security properties.

This thesis also observes that there are also solutions offered by various service providers, but it is always a proprietary solution, and production-ready, open standard does not exist for group messaging.

The MLS protocol was created as an Internet-Draft to address the problem of the lack of an open standard for secure and scalable, end-to-end encrypted, group messaging solution. In this project, the MLS protocol and architecture were studied to evaluate how MLS would address the need for a standard for secure and scalable group messaging system.

Furthermore, this project went on to evaluate the OpenMLS implementation of the MLS protocol to provide an entry point for a potential implementation the MLS protocol a service provider may wish to use in their group messaging application.

Overall, this thesis has been successful and concludes that the MLS protocol is a promising upcoming standard for secure and scalable group messaging. Additionally,

this thesis provides an entry point into a potential implementation of an application based on the MLS protocol.

7.1 Shortcomings of MLS and OpenMLS

The MLS protocol is a draft protocol. The MLS working group is still working on it as an Internet-Draft and therefore does not recommend taking MLS into use in any production systems at the time of writing this thesis. MLS is a promising upcoming standard for secure and scalable group messaging. To implement MLS in a production system, it is recommended to wait until it is release as an IETF RFC.

OpenMLS is an implementation of the draft 12 of MLS protocol. At the time of writing, it is also consequently a draft and should not be taken into production use. However, as the MLS protocol is released as an IETF RFC and OpenMLS is developed to be conformant to the RFC and made more stable with iterative development, it would serve as a great library for implementing the MLS protocol.

The OpenMLS DS and CLI are meant as a prototype or proof of concept. Therefore, anyone implementing the MLS protocol would require writing their own DS. Additionally, the CLI client interface is not user friendly and not compatible with any platform except a terminal interface. Thus, the client interfaces must also be implemented on platforms as a specific application might require.

7.2 Considerations for the Future

MLS is a great collaborative effort between the industry and academia to bring a solution to the lack an open standard for secure and scalable group messaging. It has a promising future. However, it is important to note that it should only be taken into production use after the protocol has been further studied academically from a security perspective and otherwise. Formally, it could be considered ready for production use once the MLS working group decides to release it as an IETF RCF.

References

1 WhatsApp. WhatsApp Encryption Overview, White paper [Internet] October 2020. <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>. Accessed 20 Feb 2021.

2 MLS Working Group. Messaging Layer Security (MLS). Working group home page [Internet] <https://messaginglayersecurity.rocks/>. Accessed 3 March 2022.

3 Cremers C, Hale B, Kohbrok K. The complexities of healing in secure group messaging: Why cross-group effects matter. In Proceedings of the 30th USENIX Security Symposium. USENIX : THE ADVANCED COMPUTING SYSTEMS ASSOCIATION. 2021. p. 1847-1864

4 IBM. Data integrity and message digests. Documentation [internet] <https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=concepts-data-integrity-message-digests> Accessed 3 March 2022.

5 Easttom W. Modern Cryptography. Plano, TX (USA): Springer; 2021. p 236-238

6 Easttom W. Modern Cryptography. Plano, TX (USA): Springer; 2021. p 23

7 Madden N. API Security in Action. Shelter Island, NY (USA): Manning Publications Co; 2020.

8 Wire. Messaging Layer Security. Industry report [internet] <https://wire.com/en/resources/industry-reports/messaging-layer-security/>. Accessed 10 April 2022.

9 MLS Working Group. Messaging Layer Security. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-status-of-this-memo>. Accessed 11 April 2022.

10 Cisco. mlspp. Github repository [internet] <https://github.com/cisco/mlspp>. Accessed 2 February 2022.

11 OpenMLS. OpenMLS. Documentation home page [internet] <https://openmls.tech/>. Accessed 7 May 2022.

12 Blackhat. Messaging Layer Security: Towards a new era of secure messaging. Benjamin Beurdouche, Katriel Cohn-Gordon, Raphael Robert. Presentational document [internet]. Accessed 1 May 2022.

13 MLS Working Group. MLS Protocol: Ratchet Tree Concepts. IETF Draft [internet] <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html#section-5-1> Accessed 8 May 2022.

14 MLS Working Group. MLS Protocol: Introduction. IETF Draft [internet] <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html#section-5-1> Accessed 9 May 2022.

15 Chen K, Chen J, Zhang J. Anonymous Asynchronous Ratchet Tree Protocol for Group Messaging. 2021;21(4):1-2.

16 MLS Working Group. MLS Protocol: Ratchet Tree Views. IETF Draft [internet] <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html#section-5.2-4> Accessed 9 May 2022.

17 MLS Working Group. MLS Protocol: Sequencing of Stage Changes. IETF Draft [internet] <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html#name-sequencing-of-state-changes> Accessed 10 May 2022.

18 MLS Working Group. MLS Protocol: Commit. IETF Draft [internet] <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html#name-commit> Accessed 10 May 2022.

19 The Stack. Messaging Layer Security is the coming of age in good news for privacy lovers. Cybersecurity article [internet] <https://thestack.technology/messaging-layer-security-is-coming-of-age/> Accessed 10 May 2022.

20 MLS Working Group. MLS Architecture: General Setting. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#section-2-8> Accessed 15 May 2022.

21 MLS Working Group. MLS Architecture: Authentication Service. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-authentication-service> Accessed 15 May 2022.

22 MLS Working Group. MLS Architecture: Delivery Service. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-delivery-service> Accessed 15 May 2022.

23 MLS Working Group. MLS Architecture: Key Storage. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-key-storage> Accessed 15 May 2022.

24 MLS Working Group. MLS Architecture: Delivery of Messages. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#section-4.3-4> Accessed 15 May 2022.

25 MLS Working Group. MLS Architecture: Membership Knowledge. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#section-4.4-2> Accessed 15 May 2022

26 MLS Working Group. MLS Architecture: Membership Changes. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#section-4.3-4> Accessed 15 May 2022.

27 MLS Working Group. MLS Architecture: Extensibility. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-extensibility> Accessed 15 May 2022.

28 MLS Working Group. MLS Architecture: Federation. Architecture Document [internet] <https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html#name-federation> Accessed 15 May 2022.

29 Rust. Rust Book: The Rust Programming Language. Documentation [internet] <https://doc.rust-lang.org/book/ch00-00-introduction.html> Accessed 15 May 2022.

30 Blandy J. Why Rust?. Sebastopol, CA (USA): O'Reilly Media; 2015.

Appendix

This appendix contains the code used for performing the tests in the chapter 6 is hosted which was written as a part of this project.

Test Two Member Group

```
#[test]
fn test_two_member_group() {
    // Reset the server state.
    backend::Backend::default().reset_server();

    const GROUP_NAME: &str = "test_group";

    // Client two clients.
    let mut client_1 = user::User::new("client_1".to_string());
    let mut client_2 = user::User::new("client_2".to_string());

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    // Client 1 creates a group.
    client_1.create_group(GROUP_NAME.to_string());

    // Client 1 adds Client 2 to the group.
    client_1.invite("client_2".to_string(), GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    const MESSAGE: &str = "Hello, world!";

    // Client 2 sends a message.
    client_2.send_msg(MESSAGE, GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    assert_eq!(
        client_1.read_msgs(GROUP_NAME.to_string()).unwrap(),
        Some(vec![MESSAGE.into()])
    );
}
```

Test Add Member

```
#[test]
fn test_add_member() {
    // Reset the server state.
    backend::Backend::default().reset_server();

    const GROUP_NAME: &str = "test_group";

    // Client two clients.
    let mut client_1 = user::User::new("client_1".to_string());
    let mut client_2 = user::User::new("client_2".to_string());
    let mut client_3 = user::User::new("client_3".to_string());

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    // Client 1 creates a group.
    client_1.create_group(GROUP_NAME.to_string());

    // Client 1 adds Client 2 to the group.
    client_1.invite("client_2".to_string(), GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    const MESSAGE: &str = "Hello, world!";

    // Client 2 sends a message.
    client_2.send_msg(MESSAGE, GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();

    // Client 1 adds Client 3 to the group.
    client_1.invite("client_3".to_string(), GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();
    client_3.update(None).unwrap();

    const NEW_MESSAGE: &str = "Hello, world! 2";

    // Client 2 sends a message again.
    client_2.send_msg(NEW_MESSAGE, GROUP_NAME.to_string()).unwrap();

    // Update the clients to update them to latest server state.
    client_1.update(None).unwrap();
    client_2.update(None).unwrap();
    client_3.update(None).unwrap();

    // Client 1 sees both messages.
    assert_eq!(
        client_1.read_msgs(GROUP_NAME.to_string()).unwrap(),
        Some(vec![MESSAGE.into(), NEW_MESSAGE.into()])
    );

    // Client 2 only sees the new message.
    assert_eq!(
        client_3.read_msgs(GROUP_NAME.to_string()).unwrap(),
        Some(vec![NEW_MESSAGE.into()])
    );
}
```

Statistics Test

```
#[test]
fn statistics_test() {
    use std::time::Instant;

    backend::Backend::default().reset_server();

    const MESSAGE_1: &str = "Hello, world!";
    const N_MESSAGES: i32 = 10;
    const N_MEMBERS: i32 = 25;
    let mut main_client = user::User::new("main_client".to_string());

    // Create clients
    let mut clients: Vec<user::User> = Vec::new();
    for i in 0..N_MEMBERS {
        let client = user::User::new(format!("client_{}", i));
        clients.push(client);
    }

    // Update clients
    main_client.update(None).unwrap();
    for client in &mut clients {
        client.update(None).unwrap();
    }

    // Create group
    main_client.create_group("test_group".to_string());
    main_client.update(None).unwrap();

    let invite_start = Instant::now();
    // Invite everyone
    for i in 0..N_MEMBERS {
        main_client.invite(format!("client_{}", i), "test_group".to_string()).unwrap();
    }

    // Update clients
    main_client.update(None).unwrap();
    for client in &mut clients {
        client.update(None).unwrap();
    }
    println!("Inviting {} members took: {:.2?}", N_MEMBERS, invite_start.elapsed());

    let send_message_start = Instant::now();
    // Send messages
    for _i in 0..N_MESSAGES {
        main_client.send_msg(MESSAGE_1, "test_group".to_string()).unwrap();
    }
    println!("Sending {} messages took: {:.2?}", N_MESSAGES,
send_message_start.elapsed());

    // Update clients
    let read_message_start = Instant::now();
    main_client.update(None).unwrap();
    for client in &mut clients {
        client.update(None).unwrap();
    }

    for client in &mut clients {
        client.read_msgs("test_group".to_string()).unwrap();
    }
    println!("Reading {} messages took: {:.2?}", N_MESSAGES,
read_message_start.elapsed());
}
```

These tests are also hosted in a GitHub repository which may be followed about any new updates: https://github.com/adarshk7/openmls-tests